Оглавление

ЧАСТЬ III

КОМПОНЕНТНО-ОРИЕНТИРОВАННАЯ РАЗРАБОТКА 21

Глава 20. Ключевые элементы VCL и информация о типах времени выполнения 22
Глава 21. Создание пользовательских компонентов в Delphi 62
Глава 22. Сложные методики работы с компонентами 130
Глава 23. COM-ориентированные технологии 197
Глава 24. Расширение оболочки Windows 314
Глава 25. Создание элементов управления ActiveX 387
Глава 26. Использование интерфейса Open Tools API 454
Глава 27. Разработка приложений CORBA в Delphi 490

ЧАСТЬ IV

РАБОТА С БАЗАМИ ДАННЫХ 539

Глава 28. Создание локальных приложений баз данных 540
Глава 29. Разработка приложений архитектуры клиент/сервер 604
Глава 30. Расширения баз данных VCL 650
Глава 31. Компоненты WebBroker открывают двери в Internet 700
Глава 32. Разработка приложений MIDAS 732

ЧАСТЬ V

БЫСТРАЯ РАЗРАБОТКА ПРИЛОЖЕНИЙ БАЗ ДАННЫХ 777

Глава 33. Inventory Manager: пример разработки приложения с архитектурой клиент/сервер 778
Глава 34. Диспетчер клиента: разработка приложения по технологии MIDAS 828
Глава 35. Разработка настольного приложения: сбор сведений об ошибках 850
Глава 36. Приложение с использованием компонентов WebBroker: сбор сведений об ошибках 877

ЧАСТЬ VI

ПРИЛОЖЕНИЯ 901

Приложение А. Сообщения об ошибках и исключения 902
Приложение Б. Коды ошибок Borland Database Engine 935
Приложение В. Рекомендуемая литература 963
Приложение Г. Описание содержимого прилагаемого компакт-диска 966

Содержание

ЧАСТЬ III

КОМПОНЕНТНО-ОРИЕНТИРОВАННАЯ РАЗРАБОТКА 21

Глава 20. Ключевые элементы VCL и информация о типах времени выполнения 22

Что такое компонент 23
Типы компонентов 24
Стандартные компоненты 24
Пользовательские компоненты 24
Графические компоненты 25
Невизуальные компоненты 25
Структура компонентов 25
Свойства 25
Свойства: доступ к полям компонента 26
Методы доступа к свойствам 26
Типы свойств 27
Методы 28
События 28
Назначение событию кода во время разработки программы 28
Динамическое назначение кода событию 28
Работа с потоками данных 30
Отношения владения 30
Отношения наследования 31
Иерархия визуальных компонентов 31
Класс TPersistent 32
Методы класса TPersistent 32
Класс TComponent 33
Свойства класса TComponent 33
Методы класса TComponent 34
Класс TControl 34
Класс TWinControl 35
Свойства класса TWinControl 35
Методы класса TWinControl 36
События класса TWinControl 36
Класс TGraphicControl 36
Класс TCustomControl 36
Другие классы 37
Классы TStrings и TStringsLists 37
Класс TCanvas 39
Информация о типах времени выполнения (RTTI) 40
Модуль TypInfo.pas – определитель RTTI 41

Получение информации о типах 43
Получение информации о типах объектов 47
Получение информации о происхождении объекта 48
Получение информации о типах свойств объекта 48
Проверка наличия некоторого свойства у объекта 49
Получение информации о типах указателей на методы 50
Получение RTTI-информации для упорядоченных типов 54
RTTI-информация для целочисленных типов 55
RTTI-информация для перечислимых типов 56
RTTI-информация для множеств 57
Присваивание значений свойствам с помощью RTTI 59
Резюме 61

Глава 21. Создание пользовательских компонентов в Delphi 62

Основные концепции разработки компонентов 63
Решение о необходимости создания компонента 63
Этапы разработки компонента 64
Выбор класса-предка 64
Создание модуля компонента 66
Создание свойств 67
Типы свойств 67
Создание событий 76
Происхождение событий 76
Определение свойств-событий 77
Создание методов 80
Никакой взаимозависимости! 80
Открытость метода 81
Конструкторы и деструкторы 81
Переопределение конструкторов 81
Поведение компонента во время разработки 82
Переопределение деструкторов 83
Регистрация компонента 83
Тестирование компонента 85
Создание пиктограммы компонента 87
Примеры разработки компонентов 87
Расширение возможностей компонентов-оболочек для классов Win32 87
Компонент TddgExtendedMemo – расширение компонента TMemo 88
TddgTabbedListBox – расширение компонента TListBox 91
Компонент TddgRunButton – создание свойств 97
Создание и выполнение процесса 102
Методы класса TddgRunButton 102
Компонент-контейнер TddgButtonEdit 103
Проектные решения 103
Выведение свойств вложенных объектов на поверхность 104
Вывод на поверхность событий вложенных объектов 104
Компонент TddgDigitalClock – создание событий компонента 107
Добавление форм в палитру компонентов 110
Пакеты компонентов 113
Зачем использовать пакеты 113
Сокращение размера кода 113

Дробление приложений и уменьшение их размеров 113
Хранение компонентов 114
Когда не нужно использовать пакеты 114
Типы пакетов 114
Файлы пакетов 115
Активизация использования пакетов в приложении 115
Установка пакетов в интегрированную среду разработки Delphi 115
Разработка пакетов 116
Редактор пакетов 116
Сценарии разработки пакетов 117
Сценарий 2. Только пакет разработки компонентов 120
Версии пакетов 121
Директивы компилятора, предназначенные для пакетов 121
Дополнительная информация о директиве {$WEAKPACKAGEUNIT} 121
Соглашения об именах пакетов 122
Пакеты надстроек 122
Генерирование форм надстроек 123
Резюме 129
Глава 22. Сложные методики работы с компонентами 130
Псевдовизуальные компоненты 131
Расширенные подсказки 131
Создание потомка класса THintWindow 131
Эллиптическое окно 133
Активизация потомка класса THintWindow 134
Использование компонента TddgHintWindow 134
Анимированные компоненты 135
Компонент бегущих титров 135
Написание кода компонента 135
Работа с изображением в памяти 135
Отображение компонента 137
Анимация полосы титров 138
Тестирование компонента TddgMarquee 147
Создание редакторов свойств 149
Создание объекта-потомка редактора свойств 149
Редактирование свойства в виде текста 150
Регистрация нового редактора свойства 154
Редактирование свойства в специальном диалоговом окне 155
Пример диалогового редактора свойств: расширение возможностей компонента TddgRunButton 156
Редакторы компонентов 159
Компонент TComponentEditor 159
Свойства 159
Методы 160
Стандартный редактор компонентов TDefaultEditor 160
Пример простого компонента 160
Пример редактора для простого компонента 161
Регистрация редактора компонентов 162
Работа непубликуемых компонентов с потоками данных 164

Определение свойств 164
Пример использования функции DefineProperty() 165
Компонент TddgWaveFile: пример использования функции DefineBinaryProperty() 167
Категории свойств 174
Классы категорий 175
Пользовательские категории 176
Списки компонентов: классы TCollection и TCollectionItem 179
Определение класса типа TCollectionItem: компонент TRunBtnItem 181
Определение класса типа TCollection: компонент TRunButtons 182
Реализация классов TddgLaunchPad, TRunBtnItem и TRunButtons 183
Реализация компонента TRunBtnItem 188
Реализация компонента TRunButtons 189
Реализация компонента TddgLaunchPad 189
Редактирование списка компонентов TCollectionItem в диалоговом окне редактора свойств 190
Резюме 196

Глава 23. COM-ориентированные технологии 197
Основы COM 198
COM: Component Object Model 198
COM – ActiveX – OLE 199
Терминология 199
Достоинства ActiveX 200
Стандарты OLE 1 и OLE 2 200
Структурированное хранилище 201
Единообразная передача данных 201
Потоковые модели 201
Стандарт COM+ 202
Object Pascal и COM 202
Интерфейсы 202
Интерфейс IUnknown 203
Использование интерфейсов 205
Интерфейсы и идентификаторы интерфейсов 208
Псевдоним метода 208
Тип возвращаемого значения HResult 209
COM-объекты и фабрики классов 210
Классы TComObject и TComObjectFactory 210
Внутренние COM-серверы 211
Функция DllRegisterServer() 212
Функция DllUnregisterServer() 212
Функция DllGetClassObject() 212
Функция DllCanUnloadNow() 213
Создание экземпляра внутреннего COM-сервера 213
Внешние COM-серверы 214
Регистрация 214
Создание экземпляра внешнего COM-сервера 214
Агрегирование 215
Распределенная модель COM 215

Автоматизация 215
Интерфейс IDispatch 216
Информация о типе 217
Позднее и раннее связывание 217
Регистрация 218
Создание сервера автоматизации 218
Создание внешнего сервера автоматизации 219
Создание внутреннего сервера автоматизации 232
Создание контроллеров автоматизации 237
Управление внешним сервером 237
Управление внутренним сервером 242
Использование более сложных технологий автоматизации 244
События автоматизации 245
Что представляют собой события COM 245
События в Delphi 245
События в автоматизации 246
События автоматизации в Delphi 247
События с несколькими стоками 252
Коллекции автоматизации 256
Реализация в Delphi 258
Новые типы интерфейсов в библиотеке типов 265
Обмен двоичными данными 266
Языковая поддержка для COM 270
Варианты 270
Массивы вариантов 271
Автоматизация с поздним связыванием 272
Тип данных WideString 272
Интерфейсы 273
Диспинтерфейсы 274
Сервер транзакций Microsoft (MTS) 275
Зачем нам MTS 275
Что такое MTS 276
Что лучше: знать или не знать о своем состоянии? 277
Управление временем жизни объектов 278
Пакеты 278
Безопасность 279
Ах, да, еще и транзакции! 279
Ресурсы 280
Технология MTS в Delphi 280
Мастера MTS 280
Структура MTS 282
Игра в крестики-нолики: пример приложения 284
Инсталляция сервера 295
Приложение клиента 296
Отладка MTS-приложений 301
Класс TOleContainer 301
Простейший пример приложения 302
Внедрение нового OLE-объекта 302
Внедрение или связывание существующего OLE-файла 303
Пример помощнее 304

Создание дочерней формы 305
Сохранение в файл и чтение из файла 306
Использование буфера обмена для копирования и вставки OLE-объекта 307
Резюме 313

Глава 24. Расширение оболочки Windows **314**
Вывод пиктограммы на панель задач 315
Интерфейс API 315
Обработка сообщений 317
Пиктограммы и подсказки 318
Обработка щелчков мышью 319
Скрытие приложения 321
Пример приложения 328
Панели инструментов рабочего стола Windows 329
Интерфейс API 330
Компонент TAppBar: форма окна AppBar 331
Использование компонента TAppBar 340
Ярлыки Windows 343
Создание экземпляра интерфейса IShellLink 344
Использование интерфейса IShellLink 345
Создание ярлыка 347
Получение и запись информации ярлыка 348
Пример приложения 353
Расширения оболочки 361
Мастер COM-объектов 362
Обработчики перемещений 363
Метод CopyCallback() 363
Реализация объекта TCopyHook 365
Регистрация 365
Обработчики контекстных меню 368
Интерфейс IShellExtInit 369
Интерфейс IContextMenu 370
Регистрация 372
Обработчики пиктограмм 378
Флаги пакета 378
Интерфейсы обработчика пиктограмм 379
Регистрация 381
Резюме 386

Глава 25. Создание элементов управления ActiveX **387**
Зачем создавать элементы управления ActiveX 388
Создание элемента управления ActiveX 388
Мастер создания элемента управления ActiveX 389
Параметры элемента управления ActiveX 391
Инкапсуляция элементов управления VCL 391
Среда разработки ActiveX 417
Контейнерные элементы управления 418
Окно отражения 418
Время разработки и время выполнения 418
Лицензирование элемента управления 419

Страницы свойств 420
Стандартные страницы свойств 420
Пользовательские страницы свойств 422
Активные формы ActiveForm 432
Добавление свойств к формам ActiveForm 432
Элементы управления ActiveX в Web 439
Связь с Web-броузером 440
Распространение в Web 449
Вкладка Project 450
Вкладки Packages и Additional Files 450
Вкладка Code Signing 452
Общие советы 452
Резюме 453

Глава 26. Использование интерфейса Open Tools API 454
Интерфейсы Open Tools 455
Использование интерфейса Open Tools API 457
Мастер Dumb 457
Мастер Wizard 460
Интерфейс мастера 461
Пользовательский интерфейс мастера 464
Одним выстрелом – двух зайцев: EXE и DLL 470
Мастер DDG Search 471
Мастера форм 482
Резюме 489

Глава 27. Разработка приложений CORBA в Delphi 490
Брокеры запросов объектов 491
Интерфейсы 491
Заглушки и каркасы 492
ORB-брокер VisiBroker 493
VisiBroker: службы поддержки времени выполнения 493
Служба Smart Agent (инструмент osagent) 493
Демон активизации объектов 493
Хранилище интерфейсов 493
VisiBroker: инструменты администрирования 494
Поддержка архитектуры CORBA в среде Delphi 494
Классы поддержки архитектуры CORBA 496
Мастер CORBA-объектов 498
Редактор библиотек типов IDE Delphi 504
Создание CORBA-решений в среде Delphi 5 506
Построение CORBA-сервера 506
Вызов мастера CORBA-объектов 506
Использование редактора библиотек типов 507
Реализация методов интерфейса IQueryServer 507
Запуск CORBA-сервера 522
Построение CORBA-клиента с использованием раннего связывания 522
Создание CORBA-клиента 523
Подключение к CORBA-серверу 524
Построение CORBA-клиента с использованием сценария позднего связывания 525

Кросс-язык CORBA 528
Ручной маршалинг для CORBA-сервера, написанного на языке Java 528
Компилятор Idl2Pas фирмы Inprise 531
Развертывание ORB-брокера VisiBroker 537
Резюме 538

ЧАСТЬ IV

РАБОТА С БАЗАМИ ДАННЫХ 539

Глава 28. Создание локальных приложений баз данных 540
Работа с наборами данных 541
Архитектура компонентов баз данных библиотеки VCL 542
Компоненты доступа к данным BDE 542
Открытие набора данных 543
Навигация по набору данных 543
Свойства BOF, EOF и циклическая обработка 544
Закладки 544
Пример навигации 545
Компонент TDataSource 549
Работа с полями 549
Значения полей 550
Типы полей данных 551
Имена и номера полей 551
Манипулирование полем данных 552
Редактор полей 553
Добавление полей 554
Потомки компонента TField 554
Поля и инспектор объектов 555
Вычисляемые поля 556
Подстановочные поля 557
Перетаскивание полей мышью 558
Работа с BLOB-полями 559
Класс TBlobField и типы поля 559
Пример использования BLOB-поля 560
Обновление набора данных 565
Изменение состояния набора данных 565
Фильтры 566
Фильтрация набора данных 566
Методы FindFirst/FindNext 567
Поиск записи 568
Использование компонента TTable 568
Поиск записей 568
Метод FindKey() 569
Методы SetKey()..GotoKey() 569
Поиск ближайшего соответствия 569
Использование индексов 570
Метод SetRange() 570
Метод ApplyRange() 571
Главная/подчиненная таблицы 571
События компонента TTable 573

Создание таблицы в программе 573
Модули данных 574
Пример приложения с поиском, фильтрацией и выделением диапазона данных 575
Модуль данных 575
Главная форма 575
Форма выделения диапазона данных 578
Форма поиска по ключу 580
Форма фильтрации 582
Другие типы наборов данных: Tquery и TStoredProc 585
Компонент TQuery 586
Компонент TStoredProc 586
Таблицы в текстовом файле 586
Формат файла схемы 587
Файл данных 588
Использование текстовых таблиц 589
Ограничения 589
Импорт текстовых таблиц 590
Подключение с помощью ODBC 590
Где найти драйвер ODBC 591
Пример использования ODBC: подключение к MS Access 591
Объекты данных ActiveX (ADO) 596
Кто есть кто среди стратегий доступа к данным Microsoft 596
Компоненты ADOExpress 597
Компоненты обеспечения соединения 598
Компоненты доступа ADO 598
Компоненты обеспечения совместимости 598
Установка соединения с источником данных ADO 599
Пример: установка соединения средствами ADO 602
Установка ADO 603
Резюме 603
Глава 29. Разработка приложений архитектуры клиент/сервер 604
Почему именно клиент/сервер? 605
Архитектура клиент/сервер 606
Клиент 606
Сервер 607
Бизнес-правила 607
Размещение программ бизнес-правил 608
Защита данных 608
Целостность данных 609
Централизованное управление данными 609
Распределение работ 609
Модели клиент/сервер 610
Двухуровневая модель 610
Трехуровневая модель 611
Архитектура клиент/сервер и локальные базы данных 611
Работа с наборами данных и с отдельными записями 611
Защита данных 612
Методы блокировки записей 612

Целостность данных 613
Транзакции 613
Язык SQL и его роль в технологии клиент/сервер 614
Разработка приложений клиент/сервер в Delphi 614
Разработка серверной части 615
Объекты базы данных 615
Определение таблиц 616
Типы данных 616
Создание таблицы 616
Индексы 617
Вычисляемые столбцы 617
Внешние ключи 618
Значения по умолчанию 618
Использование доменов 618
Определение бизнес-правил с помощью представлений, хранимых процедур и триггеров 619
Определение представлений 619
Определение хранимых процедур 620
Поддержка целостности данных с помощью хранимых процедур 622
Определение триггеров 624
Привилегии и права доступа к объектам базы данных 625
Предоставление доступа к таблицам 626
Предоставление доступа к представлениям 627
Предоставление доступа к хранимым процедурам 627
Отмена прав доступа пользователей 627
Разработка клиентской части 627
Использование компонента TDatabase 627
Подключение на уровне приложения 630
Управление защитой 630
Управление транзакциями 634
Какой компонент использовать – TTable или TQuery? 637
Компонент TTable и SQL-команды 637
Использование компонента TQuery 638
Динамический SQL 639
Выполнение хранимых процедур 646
Использование компонента TStoredProc 646
Получение результирующего набора хранимой процедуры из компонента TQuery 648
Резюме 649
Глава 30. Расширения баз данных VCL 650
Использование BDE 651
Модуль BDE 651
Функция Check() 651
Курсоры и дескрипторы 652
Синхронизация курсоров 652
Таблицы dBASE 653
Физический номер записи 653
Просмотр удаленных записей 654
Проверка записи на удаленность 655

Восстановление ранее удаленной записи 656
Упаковка таблицы 656
Таблицы Paradox 657
Порядковый номер 657
Упаковка таблицы 658
Ограничение результирующих наборов данных компонента TQuery 664
Прочие полезные функции BDE 665
Обобщающие функции SQL 665
Быстрое копирование таблицы 667
Пользователи сеанса Paradox 668
Создание элементов управления VCL для работы с данными 670
Расширение возможностей компонента TDataSet 674
Давным-давно... 674
...и теперь 675
Создание потомка компонента TDataSet 676
Абстрактные методы компонента TDataSet 676
Методы буферизации записи 678
Методы работы с закладками 682
Навигационные методы 683
Методы редактирования 684
Прочие методы 685
Необязательные методы работы с номером записи 688
Резюме 699

Глава 31. Компоненты WebBroker открывают двери в Internet 700
Расширения Web-серверов: ISAPI, NSAPI и CGI 702
Интерфейс CGI 702
Интерфейсы ISAPI и NSAPI 703
Создание Web-приложений с помощью Delphi 704
Классы TWebModule и TWebDispatcher 704
Классы TWebRequest и TWebResponse 708
Создание динамических HTML-страниц 711
Компонент TPageProducer 711
Классы TDataSetTableProducer и TQueryTableProducer 714
Поддержка информации о пользователях с помощью cookies 720
Перенаправление на другой Web-узел 724
Считывание информации из HTML-форм 725
Формирование потоков данных 727
Резюме 731

Глава 32. Разработка приложений MIDAS 732
Механизм построения многоуровневого приложения 733
Преимущества многоуровневой архитектуры 734
Централизованная поддержка бизнес-логики 734
Архитектура “тонкого” клиента 734
Автоматическое согласование ошибок доступа 735
Модель “портфеля” 735
Отказоустойчивость 735
Балансировка загрузки 735

Классические ошибки 736
Типичная архитектура приложения MIDAS 736
Сервер 737
Способы создания экземпляров 737
Выбор модели потоков 738
Выбор способа доступа к данным 739
Публикация служб 739
Клиент 739
Выбор соединения 739
Соединение компонентов 741
Использование технологии MIDAS для создания приложений 741
Установка сервера 741
Удаленный модуль данных (RDM) 741
Провайдеры 742
Регистрация сервера 743
Создание клиента 743
Извлечение данных 743
Редактирование данных в клиентском приложении 744
Отмена внесенных изменений 745
Возврат к исходной версии 745
Транзакции клиента: свойство SavePoint 745
Согласование данных 746
Использование стандартного диалогового окна согласования ошибок 747
Дополнительные параметры, используемые для повышения устойчивости приложения 749
Методы оптимизации клиентской части приложения 749
Ограничение размеров пакета данных 749
Использование модели "портфеля" 750
Передача на сервер динамических SQL-команд 751
Методы оптимизации серверного приложения 751
Согласование разногласий записей 751
Прочие параметры сервера 752
Поддержка связи "главная–детальная" 752
Вложенные наборы данных 753
Примеры из реальной жизни 753
Объединения 753
Обновление одной таблицы 753
Обновление нескольких таблиц 754
Приложения MIDAS и Web 755
Простой HTML 756
Компоненты InternetExpress 759
Дополнительные возможности наборов данных клиента 763
Вложенные наборы данных 764
Связь "главная–детальная" в клиентской части приложения 764
Двухуровневые приложения 770
Установка MIDAS-приложений 772
Предоставление лицензии 772
Настройка DCOM 773
Файлы, необходимые для установки приложения 774
Соглашения по установке приложений в Internet 774
Резюме 776

ЧАСТЬ V

БЫСТРАЯ РАЗРАБОТКА ПРИЛОЖЕНИЙ БАЗ ДАННЫХ 777

Глава 33. Inventory Manager: пример разработки приложения с архитектурой клиент/сервер 778

Проектирование внутреннего интерфейса 779
Определение доменов 780
Определение таблиц 782
Таблица CUSTOMER 782
Таблица PART 782
Таблица SALES 783
Таблица ITEMS 783
Определение генераторов 784
Определение триггеров 784
Определение хранимых процедур 785
Предоставление прав доступа 787
Централизованный доступ к базе данных: реализация бизнес-правил 788
Методы Login/Logout 799
Методы работы с таблицей CUSTOMER 800
Методы работы с таблицей PART 800
Методы работы с таблицей SALES 800
Методы работы с временной таблицей 801
Оповещение пользователей класса TDataModule о событиях, связанных с компонентами доступа к данным 802
Создание пользовательского интерфейса 802
Класс TMainForm – главная форма приложения 803
Форма TCustomerForm – ввод данных о покупателе 808
Форма TPartsForm – ввод данных о содержимом склада 812
Форма TSalesForm — просмотр данных о продажах 817
Форма TNewSalesForm — ввод данных о продажах 817
Диалоговое окно поиска данных о покупателях 822
Резюме 827

Глава 34. Диспетчер клиента: разработка приложения по технологии MIDAS 828

Проектирование приложения сервера 829
Проектирование клиентского приложения 832
Модуль данных клиента 832
Начальное формирование соединения 839
Согласование разногласий 840
Установка и определение режима работы клиента 841
Сохранение режима работы клиента 841
Фильтрация записей 842
Главная форма клиентского приложения 842
Резюме 849

Глава 35. Разработка настольного приложения: сбор сведений об ошибках 850

Общие требования к приложению 851
Возможность размещения в World Wide Web 851
Ввод данных о пользователе и регистрация 851
Обработка, просмотр и фильтрация записей об ошибках 852

Способы работы с записями об ошибках 852
Другие функции пользовательского интерфейса 852
Модель данных 852
Разработка модуля данных 852
Инициализация приложения и регистрация 863
Генерирование ключей для таблиц Paradox 864
Процедуры работы с данными об ошибках 864
Просмотр и фильтрация ошибок 865
Добавление пользователей 865
Добавление сведений о действиях (примечаний) 867
Разработка интерфейса пользователя 869
Главная форма 869
Другие функции пользовательского интерфейса 876
Подготовка приложения к работе в Web 876
Резюме 876

Глава 36. Приложение с использованием компонентов WebBroker: сбор сведений об ошибках 877
Макеты страниц приложения 878
Внесение изменений в модуль данных 879
Настройка компонента TDataSetTableProducer: объект dstpBugs 879
Настройка компонента TWebDispatcher: объект wbdpBugs 880
Настройка компонента TPageProducer: объект pprdBugs 880
Кодирование функций ISAPI-сервера: добавление экземпляров объекта TActionItem 881
Вспомогательные процедуры 881
Вводная страница 882
Получение и проверка регистрационного имени пользователя 884
Просмотр сведений об ошибках 887
Просмотр всех ошибок 888
Просмотр данных об ошибках, внесенных конкретным пользователем 890
Форматирование ячеек таблицы и отображение подробной информации об ошибках 892
Добавление сведений о новой ошибке 893
Ввод сведений о новой ошибке 893
Верификация введенной информации об ошибке 896
Резюме 899

ЧАСТЬ VI

ПРИЛОЖЕНИЯ 901

Приложение А. Сообщения об ошибках и исключения 902
Уровни обработки 903
Ошибки времени выполнения 904
Исключения 904
Системные ошибки программного интерфейса Win32 910

Приложение Б. Коды ошибок Borland Database Engine 935
Коды ошибок Borland Database Engine 936

Приложение В. Рекомендуемая литература **963**
Программирование на Delphi 964
Разработка компонентов 964
Программирование в Windows 964
Объектно-ориентированное программирование 964
Проектирование программного обеспечения и разработка пользовательского интерфейса 964
COM/ActiveX/OLE 965

Приложение Г. Описание содержимого прилагаемого компакт-диска **966**
Что содержится на прилагаемом компакт-диске 967
Инструкции по установке программ 967
Описание продуктов от независимых поставщиков 967
1stClass 967
Abbrevia (пробная версия) 968
Adobe Acrobat Reader 4.0 968
Advantage Database Server 5.5 для NT или NetWare 968
Async Professional (пробная версия) 969
Communicator 4.7 970
Dalis-SQL 1.5 970
EarthLink TotalAccess 2.3.2 970
Essentials (пробная версия) 971
Hawk Eye 4.0 (пробная версия) 971
InfoPower 2000 972
Internet Explorer 5.0 972
IntraBob v3.01 972
OnGuard (пробная версия) 972
Orpheus (пробная версия) 973
PowerTCP Internet Toolkit Evaluation 974
ReportBuilder Pro 4.21 (пробная версия) 974
Rubicon 2.07 (пробная версия) 974
Пробная версия StarTeam Workstation 975
SysTools (пробная версия) 975
WinZip 7.0 (SR-1) 976
Предметный указатель 966

Компонентно-ориентированная разработка

ЧАСТЬ

КЛЮЧЕВЫЕ ЭЛЕМЕНТЫ VCL И ИНФОРМАЦИЯ О ТИПАХ ВРЕМЕНИ ВЫПОЛНЕНИЯ 22

СОЗДАНИЕ ПОЛЬЗОВАТЕЛЬСКИХ КОМПОНЕНТОВ В DELPHI 62

СЛОЖНЫЕ МЕТОДИКИ РАБОТЫ С КОМПОНЕНТАМИ 130

COM-ОРИЕНТИРОВАННЫЕ ТЕХНОЛОГИИ 197

РАСШИРЕНИЕ ОБОЛОЧКИ WINDOWS 314

СОЗДАНИЕ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ ACTIVEX 387

ИСПОЛЬЗОВАНИЕ ИНТЕРФЕЙСА OPEN TOOLS API 454

РАЗРАБОТКА ПРИЛОЖЕНИЙ CORBA В DELPHI 490

Глава

20 Ключевые элементы VCL и информация о типах времени выполнения

Что такое компонент 23

Типы компонентов 24

Структура компонентов 25

Иерархия визуальных компонентов 31

Информация о типах времени выполнения (RTTI) 40

Резюме 61

Появление библиотеки классов *Object Windows Library* (*OWL*) в составе языка Turbo Pascal кардинально упростило процесс программирования в Windows. Объекты этой библиотеки автоматизировали многие операции, прежде требовавшие кропотливого труда по созданию программного кода. Программистам больше не было нужды использовать длинные операторы `case` для обработки сообщений или создавать специальные программы для работы с классами Windows — OWL делала это за них. Требовалось лишь овладеть новой методологией — объектно-ориентированным программированием.

Библиотека визуальных компонентов (*Visual Component Library* — *VCL*), впервые представленная в Delphi 1, была наследницей OWL. Она базировалась на схожей объектной модели, реализация которой, тем не менее, значительно изменилась. VCL Delphi 5 отличается от своих предшественниц, входивших в состав Delphi 1, 2, 3 и 4, расширенными функциональными возможностями.

VCL разработана специально для визуальной среды Delphi. Вместо того чтобы создавать программный код окна или диалога и описывать его поведение, достаточно модифицировать поведение и характеристики готовых компонентов в процессе визуальной разработки приложения.

Уровень знаний, необходимых для работы с VCL, зависит от способа ее использования. Delphi-программистов можно разделить на разработчиков приложений и создателей визуальных компонентов. Первые создают приложения непосредственно в визуальной среде Delphi (возможность, предоставляемая далеко не всеми средами программирования). С помощью VCL они разрабатывают графический интерфейс и некоторые другие элементы приложения, например, связывающие его с базами данных. Создатели же компонентов расширяют VCL путем написания новых компонентов. Такие компоненты доступны в виде продуктов сторонних компаний.

Независимо от того, собираетесь вы создавать приложения или компоненты Delphi, следует хорошо изучить библиотеку визуальных компонентов. Разработчик приложений должен знать, какие методы, события и свойства доступны ему в каждом компоненте. Весьма полезно также разобраться в функционировании объектной модели VCL. Наиболее распространенная проблема Delphi-разработчиков заключается в том, что они избегают использовать тот или иной компонент VCL только потому, что не до конца понимают, как он работает. Разработчикам компонентов следует глубже разбираться в специфике VCL — в ее управлении сообщениями, внутренних обозначениях, соотношениях компонентов, правах собственности, редакторах свойств и т.п. Это совершенно необходимо для того, чтобы определить, следует ли писать новый компонент или достаточно расширить уже существующий.

Эта глава целиком посвящена библиотеке VCL. В ней обсуждается иерархия компонентов и поясняется назначение ключевых уровней этой иерархии. Здесь же описаны наиболее распространенные свойства, методы и события, присущие компонентам разных уровней иерархии. И, наконец, в главе содержится описание методов работы с *информацией о типах времени выполнения* (*RTTI*).

Что такое компонент

Компоненты — это строительные блоки приложения, используя которые, разработчик создает пользовательский интерфейс и включает в приложение некоторые невизуальные элементы. Для разработчиков приложений компонент — это нечто, взятое из палитры компонентов и помещенное в форму. После помещения в форму можно манипулировать различными свойствами компонента и добавлять обработчики событий, чтобы придать ему специфический вид или задать определенное поведение. С точки зрения их непосредственных создателей, компонент — это объект, реализованный в виде кода, написанного на языке Object Pascal. Такие объекты могут, например, инкапсулировать поведение элементов системы

(элементы управления Windows 95). Другие могут представлять абсолютно новый визуальный или даже невизуальный элемент, и в этом случае поведение компонента полностью задается написанным вами кодом.

По своей сложности компоненты могут значительно отличаться друг от друга. Есть компоненты совсем простые, есть имеющие довольно развитую функциональность. Нет никаких ограничений на степень сложности компонента. Вы можете использовать простой компонент-метку `TLabel` или компонент, инкапсулирующий функциональность целой электронной таблицы.

Для работы с VCL необходимо знать существующие типы компонентов, а также понимать иерархию компонентов и назначение каждого уровня этой иерархии. Эта информация содержится в следующем разделе.

Типы компонентов

Существует четыре базовых типа компонентов, которые можно не только использовать, но и создавать в Delphi: стандартные, пользовательские и графические элементы управления, а также невизуальные компоненты.

На заметку

Часто термины *компонент* и *элемент управления* (control) употребляются в одном и том же контексте, хотя их значения не всегда совпадают. Понятие "элемент управления" относится к отдельным элементам графического пользовательского интерфейса. В Delphi элементы управления всегда являются компонентами, поскольку происходят от класса `TComponent`. Компоненты — это объекты, которые могут отображаться в палитре компонентов и изменяться в процессе разработки формы. Компоненты всегда происходят от класса `TComponent`, но не всегда являются элементами управления, т.е. элементами графического интерфейса.

Стандартные компоненты

Delphi предоставляет стандартные компоненты, инкапсулирующие поведение элементов управления Windows 95/98, например компоненты `TRichEdit`, `TTrackBar` и `TListView`. Они доступны во вкладке Win32 палитры компонентов. Каждый такой компонент можно назвать *оболочкой* (wrapper) обычного элемента управления Windows 95/98, которая облегчает его применение в программах, написанных на языке Object Pascal. Если у вас есть исходный код VCL, то вы можете посмотреть, как Borland реализует этот подход на программном уровне в файле `ComCtrls.pas`.

Совет

Для того чтобы разобраться с принципами работы VCL, нужно иметь исходный код этой библиотеки, особенно если планируется создание новых компонентов. Вероятно, лучший способ научиться разрабатывать компоненты — посмотреть, как это делает компания Borland. Поэтому настоятельно рекомендуем вам приобрести в компании Borland библиотеку времени выполнения (Runtime Library — RTL).

Пользовательские компоненты

Пользовательскими обычно называют компоненты, не являющиеся частью стандартной библиотеки компонентов Delphi. Другими словами, это такие компоненты, которые пишутся вами или другими программистами для добавления в существующий набор компонентов. Мы еще вернемся к вопросу создания таких компонентов в этой главе.

Графические компоненты

Графические компоненты позволяют применять или создавать визуальные управляющие элементы, которые не могут получить фокус ввода. Обычно они используются для отображения определенной информации, но при этом, в отличие от стандартных и пользовательских компонентов, они не расходуют ресурсов Windows. Им не нужен дескриптор (что, с другой стороны, не позволяет им получить *фокус ввода*). Примерами графических компонентов являются `TLabel` и `TShape`. Они не могут служить контейнерами компонентов, т.е. не могут владеть расположенными поверх них компонентами. Вот некоторые примеры графических компонентов: `TImage`, `TBevel` и `TPaintBox`.

Дескрипторы

Дескриптор (handle) представляет собой 32-разрядное число, указывающее на определенный экземпляр объекта в системе Win32 (в данном случае имеются в виду объекты Win32, а не Delphi). Существуют различные типы объектов в Win32: объекты ядра, объекты USER и объекты GDI. Термин “объекты ядра” применяют к событиям, объектам отображения файлов и процессам. Объекты USER — это редактируемые элементы управления, списки и кнопки. Объекты GDI — это изображения, кисти, шрифты и тому подобные оформительские инструменты.

В среде Win32 каждое окно имеет свой уникальный дескриптор, используемый во множестве функций API в качестве параметра. Delphi инкапсулирует большинство функций Win32 API и обеспечивает работу с дескрипторами. Если вы захотите использовать функции Windows API, для выполнения которых необходим дескриптор, воспользуйтесь потомками классов `TWinControl` и `TCustomControl`, так как оба этих компонента имеют свойство `Handle`.

Невизуальные компоненты

Как следует из названия, невизуальные компоненты не обладают визуальными характеристиками. Такие компоненты позволяют инкапсулировать функциональность в объект и модифицировать определенные характеристики создаваемого компонента в окне инспектора объектов путем присвоения значений его свойствам и подключения обработчиков событий. В качестве примеров можно привести классы `TOpenDialog`, `TTable` и `TTimer`.

Структура компонентов

Как уже было сказано, компоненты представляют собой классы Object Pascal, инкапсулирующие функциональность и поведение элементов, добавляемых разработчиком в приложение для придания ему необходимого поведения и установки нужных визуальных характеристик. Все компоненты имеют определенную структуру, которая обсуждается далее в этой главе.

На заметку

Различайте термины *компонент* и *класс*. Компонент — это класс, с которым можно работать в среде Delphi. Класс — это структура Object Pascal, подробно описанная в главе 2, “Язык программирования Object Pascal” (том I).

Свойства

Со свойствами вы уже познакомились в главе 2, “Язык программирования Object Pascal” (том I). Свойства предоставляют пользователю интерфейс с внутренними полями компонентов. Как правило, пользователь не имеет прямого доступа к этим полям, поскольку они объявлены в разделе `private` определения класса компонента.

Свойства: доступ к полям компонента

Свойства предоставляют доступ к полям компонента либо прямым способом, либо с помощью специальных методов доступа. Рассмотрим следующее определение свойства:

```
TCustomEdit = class(TWinControl)
private
  FMaxLength: Integer;
protected
  procedure SetMaxLength(Value: Integer);
...
published
  property MaxLength: Integer read FMaxLength write SetMaxLength default 0;
...
end;
```

Свойство MaxLength предоставляет доступ к полю FMaxLength. Определение свойства состоит из его имени, типа, объявлений read и write и необязательного значения default. Объявление read определяет способ, позволяющий получить значение поля. Свойство MaxLength получает значение поля FMaxLength непосредственно. Объявление write указывает метод присвоения полю требуемых значений. В нашем случае в свойстве MaxLength для этого используется метод SetMaxLength(). Конечно же, свойство может содержать и метод чтения значения поля, в этом случае свойство MaxLength могло быть объявлено следующим образом:

```
property MaxLength: Integer read GetMaxLength write SetMaxLength default 0;
```

Метод чтения значения поля GetMaxLength() можно было бы объявить так:

```
function GetMaxLength: Integer;
```

Методы доступа к свойствам

Методы доступа имеют только один параметр того же типа, что и у самого свойства. Целью метода write является присвоение значений внутреннему полю, на которое ссылается свойство. Промежуточный уровень в виде метода для присвоения значений полю предназначен как для защиты поля от недопустимых данных, так и для реализации при необходимости различных побочных эффектов. Например, рассмотрим реализацию метода SetMaxLength():

```
procedure TCustomEdit.SetMaxLength(Value: Integer);
begin
  if FMaxLength <> Value then
  begin
    FMaxLength := Value;
    if HandleAllocated then SendMessage(Handle, EM_LIMITTEXT, Value, 0);
  end;
end;
```

Этот метод сначала проверяет, не пытается ли пользователь присвоить полю прежнее значение. Если нет, то внутреннему полю FMaxLength присваивается необходимое значение и затем вызывается функция SendMessage() для передачи сообщения Windows EM_LIMITTEXT в окно, инкапсулируемое элементом TCustomEdit. Это сообщение ограничивает размер текста, вводи-

мого в редактируемый элемент управления. Вызов SendMessage() в методе доступа свойства write называется побочным эффектом присвоения полю значения.

Побочные эффекты — это любые действия, вызванные присвоением свойству значения. При присвоении значения свойству MaxValue побочный эффект заключается в том, что инкапсулированному редактируемому элементу управления передается максимальный размер вводимого поля. Естественно, побочные эффекты могут быть значительно сложнее.

Основным преимуществом использования свойств для доступа ко внутренним полям компонента является то, что разработчик компонента может изменить реализацию доступа к полям, а пользователю компонентов при этом не придется ничего изменять в своем коде.

Метод чтения может изменить тип возвращаемого значения поля, на которое ссылается свойство.

Другая важная причина использования свойств — возможность их редактирования в процессе разработки. Свойство, объявленное в разделе published объявления компонента, также отображается в окне инспектора объектов, и пользователь компонента может модифицировать его значение.

Гораздо больше о свойствах, их применении и методах доступа вы узнаете в главе 21, "Создание пользовательских компонентов в Delphi".

Типы свойств

К свойствам применимы стандартные правила, используемые в отношении типов Object Pascal. Важным моментом является то, что тип свойства определяет, каким образом это свойство будет редактироваться в окне инспектора объектов. Типы свойств перечислены в табл. 20.1; для получения более подробной информации воспользуйтесь интерактивной справочной системой.

Таблица 20.1. Типы свойств

Тип свойства	Интерпретация свойства Object Inspector
Простой	Числовые, символьные и строковые свойства отображаются в окне Object Inspector как числа, символы и строки соответственно. Пользователь может вводить и редактировать их значения
Перечислимый	Перечислимые типы (включая Boolean) выводятся в том же виде, что и в исходном коде. Пользователь может циклически перебирать значения двойным щелчком в столбце Value либо выбирать их в раскрывающемся списке
Множество	Свойства этого типа отображаются в окне инспектора объектов именно в виде множества. Редактируя его, пользователь рассматривает каждый элемент как имеющий тип Boolean: если элемент присутствует, то он равен True, а если нет — то False
Объект	Свойства-объекты часто имеют собственный редактор свойств. Впрочем, если у свойства, являющегося объектом, в свою очередь, есть свойства, объявленные как published, окно инспектора объектов позволяет расширить за их счет список свойств исходного объекта, а затем редактировать их в обычном порядке. Свойства-объекты должны происходить от класса TPersistent
Массив	Свойства типа массива обязаны иметь собственные редакторы. В окне Object Inspector нет встроенных возможностей для редактирования таких свойств

Методы

Поскольку компоненты — это всего лишь объекты, у них могут быть свои методы. Методы объектов уже рассматривались в главе 2, "Язык программирования Object Pascal" (том I), поэтому не будем особо останавливаться на них. Ниже, в разделе "Иерархия визуальных компонентов", вы найдете информацию о некоторых основных методах компонентов различных уровней иерархии VCL.

События

События возникают в результате выполнения каких-либо действий, обычно системных (таких, как щелчок мышью на кнопке или нажатие клавиши на клавиатуре). Компоненты содержат специальные свойства, называемые событиями, и пользователь компонента может задать код, выполняемый при наступлении того или иного события.

Назначение событию кода во время разработки программы

Если посмотреть на страницу событий компонента TEdit, то можно увидеть события OnChange, OnClick и OnDblClick. Для разработчика компонента событие — это просто указатель на соответствующие методы. Когда пользователи компонентов назначают событию некоторый код, тем самым они создают обработчик события. Например, если дважды щелкнуть на некотором событии компонента во вкладке событий окна инспектора объектов, Delphi сгенерирует заготовку метода, в которую можно будет добавить собственный программный код, как это сделано для события OnClick компонента TButton в следующем примере:

```
TForm1 = class(TForm)
  Button1: Tbutton;
  procedure Button1Click(Sender: TObject);
end;
...
procedure TForm1.Button1Click(Sender: TObject);
begin
  { Event code goes here }
end;
```

Этот фрагмент программы сгенерирован системными средствами Delphi.

Динамическое назначение кода событию

Тот факт, что события — это указатели на методы, наиболее наглядно проявляется при динамическом назначении обработчика событию. Например, для связи вашего собственного обработчика с событием OnClick компонента TButton следует вначале объявить и определить метод, который будет назначен этому событию. Этот метод должен принадлежать форме, владеющей компонентом TButton, как показано в следующем фрагменте программы:

```
TForm1 = class(TForm)
  Button1: TButton;
...
```

```
private
  MyOnClickEvent(Sender: TObject); // Объявление собственного метода
end;
...
{ Определение метода }
procedure TForm1.MyOnClickEvent(Sender: TObject);
begin
  { Ваш собственный код }
end;
```

Определенный пользователем метод `MyOnClickEvent()` должен быть назначен обработчиком события `Button1.OnClick`. Следующая строка показывает, как назначить этот метод событию `Button1.OnClick` динамически (данное назначение обычно выполняется в обработчике события `OnCreate` формы):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Button1.OnClick := MyOnClickEvent;
end;
```

Подобная технология позволяет использовать различные обработчики событий в зависимости от выполнения определенных условий в программе. Кроме того, можно вообще отключить обработчик события, присвоив событию значение `nil`:

```
Button1.OnClick := nil;
```

Динамическое назначение обработчика мало отличается от назначения обработчика события в окне инспектора объектов, за исключением того факта, что в последнем случае Delphi генерирует объявление метода. Нельзя назначить обработчику события абсолютно произвольный метод. Поскольку свойства событий — это указатели на определенные методы, они обладают специфическими признаками, зависящими от типа события. Например, типом метода `OnMouseDown` является `TMouseEvent`, определяемый как

```
TMouseEvent = procedure (Sender: TObject; Button: TMouseButton; Shift:
TShiftState; X, Y: Integer) of object;
```

Методы, назначаемые обработчику события, должны удовлетворять некоторым условиям, зависящим от типа события. Обработчики должны иметь определенное количество параметров заданных типов (с определенным порядком их следования).

Как уже отмечалось, события являются свойствами. Подобно свойствам данных, события ссылаются на закрытые поля данных компонента. Эти поля всегда имеют процедурный тип, подобный `TMouseEvent`. Взгляните на следующий фрагмент программы:

```
TControl = class(TComponent)
private
  FOnMouseDown: TMouseEvent;
protected
  property OnMouseDown: TMouseEvent read FOnMouseDown write FOnMouseDown;
public
end;
```

Вспомните, как свойства обеспечивают доступ к полям данных компонента. Как видите, событие, будучи свойством, так же предоставляет доступ к закрытому полю процедурного типа.

Более подробно вопрос создания событий и их обработчиков освещен в главе 21, "Создание пользовательских компонентов в Delphi".

Работа с потоками данных

Важной характеристикой компонентов является способность их работы с потоками, которая позволяет хранить компонент и значения относящихся к нему свойств в файле. Delphi при этом берет работу с потоками данных на себя, однако разработчикам компонентов иногда могут понадобиться возможности работы с потоками данных, выходящие за рамки автоматически предоставляемых Delphi. Фактически, создаваемый Delphi DFM-файл — не более чем файл ресурсов, содержащий информацию о потоках данных в форме и ее компонентах в виде ресурса RCDATA. Механизм работы с потоками данных в Delphi подробно изложен в главе 22, "Сложные методики работы с компонентами".

Отношения владения

Компоненты могут владеть другими компонентами. Владелец компонента определяется его свойством `Owner`. При закрытии компонента-владельца принадлежащие ему компоненты также закрываются и освобождаются. В качестве примера можно привести форму, которая обычно владеет всеми расположенными на ней компонентами. При помещении некоторого компонента в форму в окне конструктора форм она автоматически становится владельцем этого компонента. Если вы создаете компонент динамически, то должны передать его принадлежность в конструктор компонента `Create` — это значение присваивается свойству `Owner` вновь созданного компонента. В следующей строке кода показано, как передать неявную переменную `self` в конструктор `TButton.Create()`, в результате чего форма становится владельцем создаваемого компонента:

```
MyButton := TButton.Create(self);
```

Когда форма закрывается, экземпляр `TButton`, на который ссылается компонент `MyButton`, также освобождается. Это один из краеугольных камней фундамента VCL — форма определяет компоненты, которые подлежат уничтожению при закрытии, исходя из свойства `Components`, представляющего собой массив.

Можно также создать "ничейный" компонент, без владельца, передав в метод компонента `Create()` параметр `nil`. Но в этом случае вы сами должны проследить за удалением такого компонента. Этот подход можно проиллюстрировать следующим кодом:

```
MyTable := TTable.Create(nil)
try
  { Обрабатывайте MyTable по своему усмотрению }
finally
  MyTable.Free;
end;
```

Необходимо использовать конструкцию `try...finally`, обеспечивающую освобождение ресурсов при возникновении исключения. Однако не следует пользоваться описанной технологией "ничейных" компонентов, кроме тех редких специфических случаев, когда просто невозможно дать компоненту владельца.

Другим свойством, связанным с владением, является свойство Components. Это свойство представляет собой массив, содержащий список всех компонентов, принадлежащих данному компоненту-владельцу. Приведенный ниже код выводит информацию о классах всех компонентов, перечисленных в свойстве Components:

```
var
  i: integer;
begin
  for i := 0 to ComponentCount - 1 do
    ShowMessage(Components[i].ClassName);
end;
```

Приведенный фрагмент лишь иллюстрирует принципы работы с компонентами, которыми владеет некоторый компонент. На практике обычно используются более сложные операции с ними.

Отношения наследования

Не смешивайте понятия владельца и родителя компонента — это отношения совершенно разные. Некоторые компоненты могут быть родительскими для других компонентов. Такими родительскими компонентами могут быть оконные компоненты, например потомки класса TWinControl. Родительские компоненты отвечают за вызов методов отображения дочерних компонентов, а также за корректность этого отображения. Родительский компонент задается значением свойства Parent.

Родительский компонент не обязательно должен быть владельцем дочернего. Иметь разных родителя и владельца — совершенно нормальная ситуация для компонента.

Иерархия визуальных компонентов

Прочитав главу 2, "Язык программирования Object Pascal" (том I), вы узнали, что абстрактный класс TObject — это базовый класс, от которого произошли все остальные.

На рис. 20.1 изображено древо иерархии классов компонентов VCL, приведенное в справочной системе Delphi.

Разработчикам компонентов не следует делать свои компоненты непосредственными потомками класса TObject. Библиотека VCL предоставляет широкий выбор классов-потомков класса TObject, и ваши новые компоненты могут быть производными от них. Эти уже существующие классы обеспечивают большинство функциональных возможностей, которые могут понадобиться вашим компонентам. Лишь при создании классов, не являющихся компонентами, их имеет смысл делать потомками класса TObject.

Методы Create() и Destroy() класса TObject предназначены для выделения и освобождения памяти для экземпляра объекта. Конструктор TObject.Create() возвращает указатель на созданный объект. Класс TObject содержит несколько полезных функций, позволяющих получить информацию об объекте.

Библиотека VCL использует в основном внутренние вызовы методов класса TObject. Вы можете получить необходимую информацию о типе класса, имени класса, классах-предках для экземпляра класса TObject или любого его потомка.

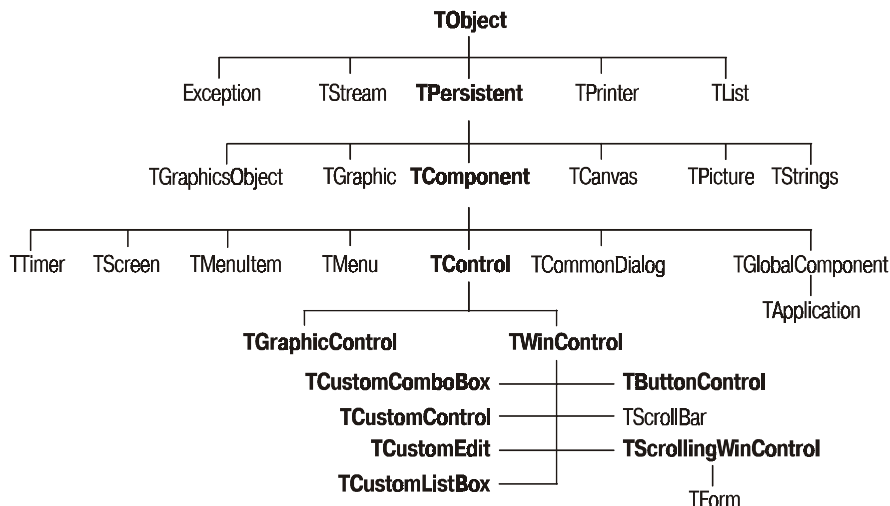


Рис. 20.1. Иерархия классов библиотеки VCL

Используйте метод `TObject.Free()` вместо метода `TObject.Destroy()`. Метод `Free` вызывает метод `Destroy()`, но перед этим проверяет, не имеет ли указатель объекта значение `nil`. Это позволяет избежать генерации исключения при попытке уничтожения несуществующего объекта.

Класс `TPersistent`

Класс `TPersistent` происходит непосредственно от класса `TObject`. Особенностью класса `TPersistent` является то, что экземпляры происходящих от него объектов могут читать и записывать свои свойства в поток. Так как все компоненты являются потомками класса `TPersistent`, то все они обладают этой способностью. Класс `TPersistent` определяет не специальные свойства или события, а только некоторые методы, полезные как пользователям, так и разработчикам компонентов.

Методы класса `TPersistent`

В табл. 20.2 приведен список некоторых интересных методов, определенных в классе `TPersistent`.

Таблица 20.2. Методы класса TPersistent

Метод	Цель
`Assign()`	Метод типа `public`, позволяет компоненту присваивать себе данные, связанные с другим компонентом
`AssignTo()`	Это защищенный метод, в который производные от `TPersistent` классы должны помещать реализацию собственного метода `AssignTo()`, объявленного в VCL. При вызове этого метода из класса `TPersistent` возникает исключительная ситуация. С помощью этого метода компонент может присвоить свои данные другому экземпляру класса — операция, обратная выполняемой методом `Assign()`

Окончание табл. 20.2

Метод	Цель
`DefineProperties()`	Этот защищенный метод позволяет разработчику компонентов определить, каким образом компонент хранит значения дополнительных или закрытых свойств. Обычно он используется для хранения в компонентах данных необычного формата, например бинарных

Работа компонентов с потоками более подробно описана в главе 12, “Работа с файлами” (том I). Благодаря возможности работы с потоками данных, компоненты могут храниться в файле на диске и считываться оттуда при необходимости.

Класс `TComponent`

Этот класс происходит непосредственно от класса `TPersistent`. Специальными характеристиками класса `TComponent` являются возможность редактирования его свойств в окне инспектора объектов и способность объектов этого класса владеть другими компонентами.

Невизуальные компоненты также происходят от класса `TComponent`, наследуя возможность их редактирования в процессе разработки. Компонент `TTimer` — прекрасный пример такого невизуального потомка класса `TComponent`. Не являясь визуальным элементом управления, он, тем не менее, доступен в палитре компонентов.

Класс `TComponent` определяет несколько интересных свойств и методов, описанных в следующих разделах.

Свойства класса `TComponent`

Свойства класса `TComponent` и описание их назначения приведены в табл. 20.3.

Таблица 20.3. Специальные свойства класса TComponent

Свойство	Назначение
`Owner`	Указывает владельца компонента
`ComponentCount`	Определяет количество компонентов, принадлежащих данному
`ComponentIndex`	Текущая позиция в списке компонентов, принадлежащих данному. Первый компонент списка имеет номер 0
`Components`	Массив, содержащий список компонентов, принадлежащих данному компоненту
`ComponentState`	Это свойство содержит информацию о текущем состоянии компонента типа `TComponentState`. Дополнительная информация о классе `TComponentState` может быть получена в интерактивной справочной системе Delphi или в главе 21, “Создание пользовательских компонентов в Delphi”
`ComponentStyle`	Управляет характеристиками поведения компонента. Этому свойству могут быть присвоены два значения — `csInheritable` и `csCheckPropAvail`. Их назначение объясняется в интерактивной справочной системе Delphi
`Name`	Содержит имя компонента

Окончание табл. 20.3

Свойство	Назначение
Tag	Целочисленное свойство без определенного значения. Предназначено для пользователей компонентов, а не их разработчиков. Благодаря целочисленному формату, свойство Tag может быть использовано для хранения указателя на структуру данных или экземпляр объекта
DesignInfo	Используется конструктором форм. Значение этого свойства изменять не следует

Методы класса TComponent

Класс TComponent определяет несколько методов, связанных с его способностью владеть другими компонентами и с его редактированием в конструкторе форм.

Класс TComponent определяет конструктор компонентов Create(). Этот конструктор создает экземпляр компонента и назначает ему владельца на основе передаваемого ему параметра. В отличие от конструктора TObject.Create(), конструктор TComponent.Create() виртуальный. Потомки класса TComponent, включающие реализацию этого конструктора, обязаны заявить о конструкторе Create(), используя директиву override. Хотя можно использовать и другие конструкторы для класса компонента, но только конструктор VCL TComponent.Create() позволяет создать экземпляр класса во время разработки или загрузить компонент из потока во время выполнения программы.

Деструктор TComponent.Destroy() уничтожает компонент вместе с любыми ресурсами, выделенными этим компонентом.

Метод TComponent.Destroying() приводит компонент и принадлежащие ему компоненты в состояние, означающее, что они разрушены. Метод TComponent.DestroyComponents() уничтожает все компоненты, принадлежащие данному. Однако вам вряд ли придется использовать эти методы самостоятельно.

Метод TComponent.FindComponent() удобно использовать для получения указателя на компонент с известным именем. Предположим, вы знаете, что главная форма имеет компонент TEdit с именем Edit1. Тогда для получения указателя на экземпляр компонента достаточно выполнить следующий код:

```
EditInstance := FindComponent.('Edit1');
```

В этом примере экземпляр компонента EditInstance должен иметь тип TEdit. Метод FindComponent возвратит значение nil, если объекта с указанным именем не существует.

Метод TComponent.GetParentComponent() позволяет найти экземпляр компонента-предка. Этот метод возвращает значение nil в случае отсутствия такого компонента.

Метод TComponent.HasParent() возвращает булево значение, определяющее, есть ли у компонента родитель. Запомните, что этот метод не проверяет наличия владельца у заданного компонента.

Метод TComponent.InsertComponent() позволяет добавить компонент, переходящий во владение вызывающего компонента. Метод TComponent.RemoveComponent() удаляет все компоненты, принадлежащие компоненту, вызывающему этот метод. В обычных случаях эти методы не используются, потому что они автоматически вызываются конструктором компонента Create() и его деструктором Destroy().

Класс TControl

Класс TControl определяет множество свойств, методов и событий, часто используемых визуальными компонентами. Так, класс TControl предоставляет визуальным компонентам возможность их отображения. Класс TControl содержит такие позиционные свойства, как

Top и Left, свойства размеров Width и Height, значения которых определяют размеры элемента по горизонтали и вертикали. Имеются и некоторые другие свойства: ClientRect, ClientWidth и ClientHeight.

Класс TControl вводит свойства, отвечающие за внешний вид и доступ к компоненту: Visible, Enabled и Color. В свойстве Font даже можно задать шрифт, используемый для текста, помещаемого в компонент TControl. Этот текст выводится с использованием свойств Text и Caption.

В классе TControl впервые появляются некоторые стандартные события: события мыши OnClick, OnDblClick, OnMouseDown, OnMouseMove и OnMouseUp, а также события перетаскивания с помощью мыши OnDragOver, OnDragDrop и OnEndDrag.

Сам по себе класс TControl не очень полезен на своем уровне иерархии. Прямые потомки этого класса никогда не создаются.

Компонент TControl может иметь родительский компонент. Он обязательно должен принадлежать классу TWinControl (родительские элементы управления должны быть оконными элементами управления). Для этого в классе TControl введено свойство Parent.

Большинство элементов управления Delphi являются производными от прямых потомков класса TControl: классов TWinControl и TGraphicControl.

Класс TWinControl

Стандартные элементы управления Windows происходят от класса TWinControl. Объекты пользовательского интерфейса программ Windows — это и есть стандартные элементы управления. Поля ввода, списки, комбинированные списки и кнопки — примеры подобных элементов. Так как Delphi инкапсулирует поведение стандартных элементов управления Windows, то вместо функций Windows API для манипулирования этими компонентами используются их свойства.

Три основные особенности класса TWinControl: объекты класса TWinControl имеют дескриптор, могут получить фокус ввода и являться родительским элементом для других компонентов. Свойства, методы и события класса TWinControl поддерживают изменение фокуса, события клавиатуры, отображение элементов и другие необходимые функции.

Разработчик приложений чаще всего работает с классами-потомками класса TWinControl. Разработчик компонентов, несомненно, заинтересуется классом TCustomControl — потомком класса TWinControl.

Свойства класса TWinControl

В классе TWinControl определено несколько свойств, предназначенных для изменения фокуса и внешнего вида элемента.

Свойство TWinControl.Brush используется для отображения элемента управления. О нем уже шла речь в главе 8, "GDI, шрифты и графика" (том I).

Свойство-массив TWinControl.Controls предоставляет доступ к списку всех элементов управления, для которых вызывающий компонент TWinControl является родительским.

Свойство TWinControl.Ctl3D определяет использование трехмерной стилизации при отображении элемента.

Свойство TWinControl.Handle содержит дескриптор объекта Windows, инкапсулированный классом TWinControl. Именно этот дескриптор следует использовать в качестве параметра функций Windows API, требующих указывать дескриптор окна.

Свойство TWinControl.HelpContext содержит контекстный справочный номер, соответствующий определенному справочному окну в файле справки. Это свойство применяется для поддержки функций контекстно-зависимой справки для отдельных элементов управления.

Свойство TWinControl.Showing содержит информацию о видимости элемента.

Свойство TWinControl.TabStop содержит булево значение, определяющее использование в данном элементе вкладок. Свойство TWinControl.TabOrder показывает, где в списке элементов-вкладок родительского компонента находится указанный элемент.

Методы класса TWinControl

Компонент TWinControl также предлагает несколько методов создания и позиционирования окна, управления фокусом ввода и диспетчеризации событий. Подробное обсуждение этих методов явно не укладывается в рамки одной главы, однако все они исчерпывающе описаны в интерактивной справочной системе Delphi. О самых интересных методах мы поговорим подробнее.

Методы создания окна, его повторного создания и уничтожения в основном применяются разработчиками компонентов и рассматриваются в главе 21, "Создание пользовательских компонентов в Delphi". Это методы CreateParams(), CreateWnd(), CreateWindowHandle(), DestroyWnd(), DestroyWindowHandle() и RecreateWnd().

К методам управления фокусом, позиционирования и выравнивания окна относятся: CanFocus(), Focused(), AlignControls(), EnableAlign(), DisableAlign() и ReAlign().

События класса TWinControl

Класс TWinControl предлагает новые события для работы с клавиатурой и управления фокусом. События клавиатуры — это OnKeyDown, OnKeyPress и OnKeyUp. К событиям управления фокусом относятся OnEnter и OnExit. Все они описаны в интерактивной справочной системе Delphi.

Класс TGraphicControl

В отличие от класса TWinControl, объекты класса TGraphicControl не имеют дескриптора окна и поэтому не могут получить фокус. Они также не могут быть родительскими для других элементов управления. Класс TGraphicControl используется, когда необходимо отобразить на форме какую-то информацию, не предоставляя пользователю доступ к этому элементу. Преимуществом элементов TGraphicControl является то, что они не нуждаются в дескрипторе, использующем системные ресурсы Windows. К тому же, отсутствие дескриптора говорит о том, что элемент TGraphicControl не участвует в непрерывном процессе перерисовки окон. Таким образом, отображение элемента типа TGraphicControl происходит значительно быстрее, чем его TWinControl-эквивалента.

Объект класса TGraphicControl может реагировать на события, связанные с мышью. В действительности сообщения мыши обрабатываются родительским компонентом класса TGraphicControl, а затем передаются его дочерним элементам.

Класс TGraphicControl позволяет окрашивать представляемые им элементы управления, и поэтому обладает свойством Canvas типа TCanvas. Он также содержит метод Paint(), который элементы-потомки обязаны переопределять.

Класс TCustomControl

Возможно, вы заметили, что имена некоторых потомков класса TWinControl начинаются с сочетания "TCustom", например: TCustomComboBox, TCustomControl, TCustomEdit и TCustomListBox.

Нестандартные (иначе — пользовательские), т.е. Custom-элементы управления выполняют почти те же функции, что и потомки класса TWinControl, за исключением того, что благодаря некоторым специальным визуальным особенностям и характеристикам поведения, разработчик получает заготовку, на основе которой можно создавать свои собственные пользовательские компоненты. Как разработчику компонентов, вам придется позаботиться о средствах отображении пользовательского элемента управления.

Другие классы

Несколько классов не принадлежат к разряду компонентов, однако служат для программной поддержки существующих компонентов. Как правило, эти классы являются свойствами других компонентов и происходят непосредственно от класса TPersistent. Некоторые из них имеют типы TString, TCanvas и TCollection.

Классы TStrings и TStringsLists

Абстрактный класс TStrings предоставляет возможность манипулировать списками строк, принадлежащих компонентам, подобно тому, как это делается с помощью элемента управления типа TListBox. Однако на самом деле класс TStrings не управляет памятью для хранения и обработки этих строк (этим занимается тот элемент управления, который является владельцем класса TStrings). Класс TStrings лишь определяет методы и свойства, необходимые для доступа к строкам и их обработки, не используя функции Win32 API и системные сообщения.

Заметьте, что TStrings — абстрактный класс, т.е. в нем нет реализации кода, необходимого для работы со строками, он лишь определяет требуемые методы. Реализация же методов обработки строк относится к компетенции потомков этого класса.

Чтобы внести ясность, приведем такой пример. Пусть есть несколько компонентов со свойствами типа TStrings — TListBox.Items, TMemo.Lines и TComboBox.Items. Каждое из этих свойств имеет тип TStrings. Но как вызвать методы этих свойств, если они еще не воплощены в коде? Хороший вопрос. Ответ таков: хотя все эти свойства определены как TStrings, переменная, к которой относится, например, свойство TListBox.FItems, создана как экземпляр класса-потомка, методы которого и будут вызываться. Чтобы прояснить этот момент, отметим, что переменная FItems является закрытым полем свойства Items класса TListBox:

```
TCustomListBox = class(TWinControl)
 private
   FItems: TStrings;
```

На заметку

Несмотря на то что в качестве "классового" типа в предыдущем программном фрагменте используется класс TCustomListBox, класс TListBox выводится непосредственно из класса TCustomListBox в том же самом модуле и, следовательно, имеет доступ к его закрытым полям.

Модуль STDCTRLS.PAS, являющийся частью библиотеки VCL, определяет класс TListBoxStrings — потомок класса TStrings. Его объявление приведено в листинге 20.1.

Листинг 20.1. Объявление класса TListBoxStrings

```
TListBoxStrings = class(TStrings)
  private
    ListBox: TCustomListBox;
  protected
```

```
    procedure Put(Index: Integer; const S: string); override;
    function Get(Index: Integer): string; override;
    function GetCount: Integer; override;
    function GetObject(Index: Integer): TObject; override;
    procedure PutObject(Index: Integer; AObject: TObject); override;
    procedure SetUpdateState(Updating: Boolean); override;
  public
    function Add(const S: string): Integer; override;
    procedure Clear; override;
    procedure Delete(Index: Integer); override;
    procedure Exchange(Index1, Index2: Integer); override;
    function IndexOf(const S: string): Integer; override;
    procedure Insert(Index: Integer; const S: string); override;
    procedure Move(CurIndex, NewIndex: Integer); override;
end;
```

Далее в этом же модуле следует реализация методов класса-потомка. Когда класс TListBox создает свои экземпляры для переменной FItems, на самом деле создается экземпляр класса-потомка, на который можно ссылаться с помощью свойства FItems:

```
constructor TCustomListBox.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ...
// Обратите внимание: создается экземпляр TListBoxStrings
  FItems := TListBoxStrings.Create;
  ...
end;
```

Следует четко уяснить, что класс TStrings лишь определяет свои методы, но не содержит их реализацию. Ее осуществляют классы-потомки. Всем разработчикам компонентов полезно изучить эту технологию. Если у вас возникнут вопросы, обратитесь к исходному коду библиотеки VCL.

Если вы не разрабатываете компоненты, но хотели бы использовать список строк, следует воспользоваться другим потомком класса TStrings — вполне самодостаточным классом TStringList. Класс TStringList обрабатывает список строк, размещаемых вне компонента. Его главным достоинством является полная совместимость с классом TStrings. Это означает, что вы можете напрямую присваивать экземпляр класса TStringList свойству типа TStrings соответствующего элемента управления. В следующем фрагменте кода показано, как можно создать экземпляр класса TStringList:

```
var
  MyStringList: TStringList;
begin
  MyStringList := TStringList.Create;
```

Для добавления строк к этому экземпляру класса TStringList поступите следующим образом:

```
MyStringList.Add('Red');
MyStringList.Add('White');
MyStringList.Add('Blue');
```

Если требуется добавить одинаковые строки одновременно в компоненты TMemo и TListBox, следует воспользоваться совместимостью свойств TStrings, принадлежащих различным компонентам, и выполнить присвоение одной строкой кода:

```
Memo1.Lines.Assign(MyStringList);
ListBox1.Items.Assign(MyStringList);
```

Вместо непосредственного присвоения, подобного Memo1.Lines := MyStringList, можно копировать экземпляры класса TStrings с помощью метода Assign(). В табл. 20.4 приведены некоторые часто используемые методы классов типа TStrings.

Таблица 20.4. Часто используемые методы классов типа TStrings

Метод класса TStrings	Описание
Add (const S: String): Integer	Добавляет строку S в список и возвращает ее позицию в нем
AddObject (const S: string; AObject: TObject): Integer	Добавляет строку и объект в строку или в объект списка строк
AddStrings (Strings: TStrings)	Копирует строки из указанного объекта TStrings в конец существующего списка строк
Assign (Source: TPersistent)	Замещает существующие строки строками, заданными параметром Source
Clear	Удаляет все строки из списка
Delete (Index: Integer)	Удаляет строку, находящуюся в позиции Index
Exchange (Index1, Index2: Integer)	Меняет местами строки с позициями Index1 и Index2
IndexOf (const S: String): Integer	Возвращает позицию заданной строки S в списке
Insert (Index: Integer; const S: String)	Вставляет строку S в позицию Index
Move (CurIndex, NewIndex: Integer)	Перемещает строку с позицией CurIndex в позицию NewIndex
LoadFromFile (const FileName: String)	Считывает текстовый файл с именем FileName и помещает его строки в список
SaveToFile (const FileName: string)	Сохраняет список в текстовом файле FileName

Класс TCanvas

Свойство Canvas типа TCanvas применяется в оконных элементах управления и представляет собой использующуюся ими поверхность рисования. Класс TCanvas инкапсулирует так называемый *контекст устройства* (device context) окна. В этом классе содержится множество функций и объектов, необходимых для прорисовки поверхности окна. Детальное описание этого класса дано в главе 8, “GDI, шрифты и графика” (том I).

Информация о типах времени выполнения (RTTI)

В главе 2, “Язык программирования Object Pascal” (том I) мы уже познакомились с понятием типа времени выполнения (RTTI). Теперь наступил момент более углубленно рассмотреть информацию RTTI, использование которой позволяет расширить обычные возможности языка Object Pascal. Мы обсудим, как получать информацию о текущих типах объектов и данных в своих программах, — подобно тому, как это делается в интегрированной среде Delphi.

Итак, каким образом проявляет себя RTTI? Судить об этом можно, по крайней мере, по двум областям применения. Первая — работа с интегрированной средой Delphi. С помощью RTTI интегрированная среда узнает все данные об объекте и компоненте, с которым вы работаете, помещая их в окно инспектора объектов. Конечно, не вся эта информация относится к RTTI, но для удобства будем рассматривать только ее. Вторая область — выполняемый код. В главе 2, “Язык программирования Object Pascal” (том I), вы уже познакомились с операторами is и as.

Для иллюстрации типичного использования информации RTTI рассмотрим оператор is.

Предположим, что требуется сделать все компоненты TEdit создаваемой формы доступными только для чтения. Это достаточно просто: пройдитесь в цикле по всем компонентам, используя оператор is для выявления компонентов класса TEdit, а затем присвоите их свойствам ReadOnly значения True.

```
for i := 0 to ComponentCount - 1 do
  if Components[i] is TEdit then
    TEdit(Components[i]).ReadOnly := True;
```

Оператор as обычно применяется для выполнения определенных действий над параметром Sender обработчика события, используемого в различных компонентах. Если все компоненты произошли от общего предка, к свойству которого вы хотите получить доступ, в обработчике события можно использовать оператор as для безопасного приведения типа параметра Sender к соответствующему потомку, что позволит получить доступ к нужному свойству.

```
procedure TForm1.ControlOnClickEvent(Sender: TObject);
var
  i: integer;
begin
 (Sender as TControl).Enabled := False;
end;
```

Эти примеры с *безопасным использованием типов* иллюстрируют применение расширенных возможностей языка Object Pascal, опосредованным образом использующих RTTI. Теперь рассмотрим задачу, решение которой требует непосредственного использования информации RTTI.

Допустим, у вас есть форма с различными компонентами: одни из них предназначены для работы с данными, а другие — нет. Однако вам нужно выполнить некоторые действия только над компонентами, работающими с данными. С одной стороны, конечно, вы могли бы в цикле перебрать все компоненты массива Components формы и проверить каждый на соответствие требуемому типу. А для этого пришлось бы перебрать все типы компонентов, используе-

мых для работы с данными. С другой стороны, у ваших компонентов, работающих с данными, может не быть общего предка (базового класса), соответствие которому было бы легко установить. Для такого случая, например, неплохо было бы иметь тип `TdataAwareControl`, но его, к сожалению, не существует.

Проще всего определить, интересует ли нас данный компонент, проверив существование свойства `DataSource` (известно, что это свойство присуще всем компонентам, работающим с данными). Для такой проверки потребуется непосредственное использование информации RTTI.

Далее в этой главе более подробно рассматриваются возможности использования информации RTTI для решения подобных проблем.

Модуль `TypInfo.pas` — определитель RTTI

Информация о типе существует для любого объекта (потомка класса `TObject`). Она содержится в памяти и при необходимости считывается интегрированной средой или библиотекой времени выполнения. В модуле `TypInfo.pas` определяются структуры, позволяющие запрашивать информацию о типах. Некоторые методы класса `Tobject`, описанные в главе 2, "Язык программирования Object Pascal" (том I), перечислены в табл. 20.5.

Таблица 20.5. Некоторые методы класса TObject

Функция	Возвращаемый тип	Что возвращается
`ClassName()`	Строка	Имя класса объекта
`ClassType()`	`TClass`	Тип объекта
`InheritsFrom()`	`Boolean`	Значение, определяющее, происходит ли класс от другого заданного класса
`ClassParent()`	`TClass`	Тип объекта-предка
`InstanceSize()`	Слово	Размер экземпляра объекта в байтах
`ClassInfo()`	Указатель	Указатель на находящуюся в памяти информацию (RTTI) об объекте

Теперь пришло время подробно остановиться на функции `ClassInfo()`, определяемой как

```
class function ClassInfo: Pointer;
```

Эта функция возвращает указатель на информацию RTTI для вызывающего класса, а именно — на структуру типа `PTypeInfo`. Этот тип определен в модуле `TypeInfo.pas` как указатель на структуру `TTypeInfo`. Оба определения приведены в следующем фрагменте `TypeInfo.pas`:

```
PPTypeInfo = ^PTypeInfo;
  PTypeInfo = ^TTypeInfo;
  TTypeInfo = record
    Kind: TTypeKind;
    Name: ShortString;
   {TypeData: TTypeData}
  end;
```

Превращенное в комментарий поле TypeData содержит информацию о типе данного класса. Тип этой информации зависит от значения поля Kind. Это поле может принимать одно из значений перечисления TTypeKind:

```
TTypeKind = (tkUnknown, tkInteger, tkChar, tkEnumeration, tkFloat,
    tkString, tkSet, tkClass, tkMethod, tkWChar, tkLString, tkWString,
    tkVariant, tkArray, tkRecord, tkInterface);
```

Вновь обратимся к тексту модуля TypeInfo.pas и ознакомимся с подтипами некоторых перечисленных выше значений. Например, tkFloat может содержать такие подтипы:

```
TFloatType = (ftSingle, ftDouble, ftExtended, ftComp, ftCurr);
```

Теперь вам известно, что по значению поля Kind можно определить, к какому типу относится переменная TypeData. Структура TTypeData определяется в модуле TypeInfo.pas, как показано в листинге 20.2.

Листинг 20.2. Определение структуры TTypeData

```
PTypeData = ^TTypeData;
TTypeData = packed record
  case TTypeKind of
    tkUnknown, tkLString, tkWString, tkVariant: ();
    tkInteger, tkChar, tkEnumeration, tkSet, tkWChar: (
        OrdType: TOrdType;
        case TTypeKind of
          tkInteger, tkChar, tkEnumeration, tkWChar: (
            MinValue: Longint;
            MaxValue: Longint;
            case TTypeKind of
              tkInteger, tkChar, tkWChar: ();
              tkEnumeration: (
                BaseType: PPTypeInfo;
                NameList: ShortStringBase));
          tkSet: (
            CompType: PPTypeInfo));
    tkFloat: (FloatType: TFloatType);
    tkString: (MaxLength: Byte);
    tkClass: (
        ClassType: TClass;
        ParentInfo: PPTypeInfo;
        PropCount: SmallInt;
        UnitName: ShortStringBase;
       {PropData: TPropData});
    tkMethod: (
      MethodKind: TMethodKind;
      ParamCount: Byte;
      ParamList: array[0..1023] of Char
      {ParamList: array[1..ParamCount] of
        record
```

```
        Flags: TParamFlags;
        ParamName: ShortString;
        TypeName: ShortString;
      end;
      ResultType: ShortString});
  tkInterface: (
      IntfParent : PPTypeInfo; { ancestor }
      IntfFlags : TIntfFlagsBase;
      Guid : TGUID;
      IntfUnit : ShortStringBase;
     {PropData: TPropData});
  tkInt64: (
      MinInt64Value, MaxInt64Value: Int64);
end;
```

Как видите, структура `TTypeData` представляет собой большую запись с вариантами. Если вы знакомы с подобными записями и указателями, то легко разберетесь и с принципами RTTI. Они кажутся сложными из-за того, что технология RTTI недокументированна.

На заметку

Часто Borland не документирует некоторые возможности, так как они подлежат изменению в следующей версии продукта. При использовании таких возможностей (например, недокументированной технологии RTTI) может оказаться, что ваш код будет не вполне совместим с будущими версиями Delphi.

Теперь продемонстрируем использование этих структур RTTI для получения информации о типах.

Получение информации о типах

Чтобы продемонстрировать технологию получения информации о типе времени выполнения объекта, был создан проект, текст определения главной формы которого представлен в листинге 20.3.

Листинг 20.3. Главная форма проекта `ClassInfo.dpr`

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls, DBClient, MidasCon, MConnect;

type

  TMainForm = class(TForm)
    pnlTop: TPanel;
    pnlLeft: TPanel;
    lbBaseClassInfo: TListBox;
    spSplit: TSplitter;
```

```
    lblBaseClassInfo: TLabel;
    pnlRight: TPanel;
    lblClassProperties: TLabel;
    lbPropList: TListBox;
    lbSampClasses: TListBox;
    procedure FormCreate(Sender: TObject);
    procedure lbSampClassesClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation
uses TypInfo;

{$R *.DFM}

function CreateAClass(const AClassName: string): TObject;
{ Этот метод показывает, как создать класс по его имени. Не забудьте
  зарегистрировать этот класс с помощью метода RegisterClasse(), как это сделано
  в инициализации данного модуля. }
var
  C : TFormClass;
  SomeObject: TObject;
begin
  C := TFormClass(FindClass(AClassName));
  SomeObject := C.Create(nil);
  Result := SomeObject;
end;

procedure GetBaseClassInfo(AClass: TObject; AStrings: TStrings);
{ Этот метод позволяет получить некоторые основные данные RTTI объекта
  и возвращает эту информацию параметру AStrings. }
var
  ClassTypeInfo: PTypeInfo;
  ClassTypeData: PTypeData;
  EnumName: String;
begin
  ClassTypeInfo := AClass.ClassInfo;
  ClassTypeData := GetTypeData(ClassTypeInfo);
  with AStrings do
  begin
    Add(Format('Class Name:     %s', [ClassTypeInfo.Name]));
    EnumName := GetEnumName(TypeInfo(TTypeKind), Integer(ClassTypeInfo.Kind));
    Add(Format('Kind:           %s', [EnumName]));
    Add(Format('Size:           %d', [AClass.InstanceSize]));
```

```
    Add(Format('Defined in:     %s.pas', [ClassTypeData.UnitName]));
    Add(Format('Num Properties: %d',[ClassTypeData.PropCount]));
  end;
end;
procedure GetClassAncestry(AClass: TObject; AStrings: TStrings);
{ Этот метод определяет предков данного объекта и возвращает имена классов в
параметр AStrings. }
var
  AncestorClass: TClass;
begin
  AncestorClass := AClass.ClassParent;
  { Проходит по всему генеалогическому древу класса - от родителя Sender до
родоначальника. }
  AStrings.Add('Class Ancestry');
  while AncestorClass <> nil do
  begin
    AStrings.Add(Format('    %s',[AncestorClass.ClassName]));
    AncestorClass := AncestorClass.ClassParent;
  end;
end;

procedure GetClassProperties(AClass: TObject; AStrings: TStrings);
{ Этот метод позволяет получить имена свойств и методов данного объекта и вернуть
эту информацию параметру AStrings. }
var
  PropList: PPropList;
  ClassTypeInfo: PTypeInfo;
ClassTypeData: PTypeData;
  i: integer;
  NumProps: Integer;
begin

  ClassTypeInfo := AClass.ClassInfo;
  ClassTypeData := GetTypeData(ClassTypeInfo);

  if ClassTypeData.PropCount <> 0 then
  begin
    { Размещает необходимую память для хранения указателей на структуры
      TPropInfo исходя из количества свойств.}
    GetMem(PropList, SizeOf(PPropInfo) * ClassTypeData.PropCount);
    try
      // Заполняет список PropList указателями
      // на структуры TPropInfo
      GetPropInfos(AClass.ClassInfo, PropList);
      for i := 0 to ClassTypeData.PropCount - 1 do
        // Отфильтровываются свойства, являющиеся событиями
        // (свойства, хранящие указатели на методы)
        if not (PropList[i]^.PropType^.Kind = tkMethod) then
          AStrings.Add(Format('%s: %s', [PropList[i]^.Name,
          PropList[i]^.PropType^.Name]));
```

```
      { Теперь ищутся свойства-события (свойства, хранящие указатели на методы)}
      NumProps := GetPropList(AClass.ClassInfo, [tkMethod], PropList);
      if NumProps <> 0 then begin
        AStrings.Add('');
        AStrings.Add('   СОБЫТИЯ   ================ ');
        AStrings.Add('');
      end;
      // Параметр AStrings заполняется именами событий
      for i := 0 to NumProps - 1 do
          AStrings.Add(Format('%s: %s', [PropList[i]^.Name,
          PropList[i]^.PropType^.Name]));

    finally
      FreeMem(PropList, SizeOf(PPropInfo) * ClassTypeData.PropCount);
    end;
  end;

end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  // В список добавляются некоторые дополнительные классы
  lbSampClasses.Items.Add('TApplication');
  lbSampClasses.Items.Add('TButton');
  lbSampClasses.Items.Add('TForm');
  lbSampClasses.Items.Add('TListBox');
  lbSampClasses.Items.Add('TPaintBox');
  lbSampClasses.Items.Add('TMidasConnection');
  lbSampClasses.Items.Add('TFindDialog');
  lbSampClasses.Items.Add('TOpenDialog');
  lbSampClasses.Items.Add('TTimer');
  lbSampClasses.Items.Add('TComponent');
  lbSampClasses.Items.Add('TGraphicControl');
end;

procedure TMainForm.lbSampClassesClick(Sender: TObject);
var
  SomeComp: TObject;
begin
  lbBaseClassInfo.Items.Clear;
  lbPropList.Items.Clear;

  // Создание экземпляра выбранного класса
  SomeComp := CreateAClass(lbSampClasses.Items[lbSampClasses.ItemIndex]);
  try
    GetBaseClassInfo(SomeComp, lbBaseClassInfo.Items);
    GetClassAncestry(SomeComp, lbBaseClassInfo.Items);
    GetClassProperties(SomeComp, lbPropList.Items);
  finally
    SomeComp.Free;
```

```
  end;
end;

initialization
begin
  RegisterClasses([TApplication, TButton, TForm, TListBox, TPaintBox,
    TMidasConnection, TFindDialog, TOpenDialog, TTimer, TComponent,
    TGraphicControl]);
end;
end.
```

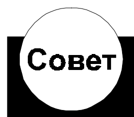
В этой главной форме содержится три списка. Список lbSampClasses включает имена классов нескольких объектов, информацию о типах которых необходимо получить. При выборе объекта из списка lbSampClasses в список lbBaseClassInfo заносится основная информация о размере и происхождении этого объекта, а в список lbPropList — сведения о свойствах выбранного объекта.

Для получения информации о классе используются три вспомогательные процедуры.

- GetBaseClassInfo() — заполняет список основной информацией об объекте — типе, размере, модуле, в котором тот определяется, и количестве свойств.
- GetClassAncestry() — заполняет список именами объектов-предков.
- GetClassProperties() — заполняет список свойствами данного класса и их типами.

Каждая процедура в качестве параметров использует экземпляр объекта и список строк.

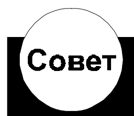
При выборе пользователем одного из классов в списке lbSampClasses, его процедура обработки события OnClick (lbSampClassesClick()) вызывает вспомогательную функцию CreateAClass(), которая создает экземпляр класса, заданного именем типа класса. Затем этот экземпляр объекта передается по назначению, чтобы в нужный список было занесено значение соответствующего свойства TListBox.Items.



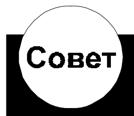
Функция CreateAClass() может быть использована для создания класса, заданного его именем. Но, как уже было сказано, вы должны зарегистрировать класс, передаваемый этой функции в качестве параметра, с помощью вызова процедуры RegisterClasses().

Получение информации о типах объектов

Процедура GetBaseClassInfo() передает значение, возвращаемое функцией TObject.ClassInfo(), функции GetTypeData(), определенной в модуле TypeInfo.pas. Целью этой процедуры является возвращение указателя на структуру TTypeData, построенную для класса, структура PTypeInfo которого была передана этой процедуре (см. листинг 20.2). Процедура GetBaseClassInfo() просто указывает на различные поля структур TTypeInfo и TTypeData для расширения списка AStrings. Обратите внимание на использование функции GetEnumName() для получения строки с именем перечислимого типа. Это также функция RTTI, определенная в файле TypInfo.pas. О RTTI перечислимых типов рассказывается в следующем разделе.



Используйте функцию GetTypeData(), определенную в файле TypInfo.pas, чтобы получить указатель на структуру TTypeInfo данного класса. Результат вызова функции TObject.ClassInfo() следует передать функции GetTypeData().



Вы можете воспользоваться функцией GetEnumName() для получения имени перечислимого типа, представленного в виде строки. Функция GetEnumValue() возвращает значение перечислимого типа по заданному имени.

Получение информации о происхождении объекта

Процедура GetClassAncestry() заполняет список строк именами классов-предков данного объекта. Это довольно простая операция, использующая процедуру ClassParent() данного объекта, возвращающая указатель на тип TClass класса-предка или nil, если достигнут класс-прародитель. Процедура GetClassAncestry() "проходит" по всему генеалогическому древу и добавляет имена классов-предков в соответствующий список строк.

Получение информации о типах свойств объекта

Значение TTypeData.PropCount соответствует общему количеству свойств (если таковые имеются) данного объекта. Существует несколько подходов для получения информации о свойствах данного класса, но мы приведем лишь два из них.

Процедура GetClassProperties() начинается так же, как и предыдущих два метода, — с передачи результата функции ClassInfo() в функцию GetTypeData() для получения указателя на структуру TTypeData класса. Затем, исходя из значения ClassTypeData.PropCount, выделяется память для переменной PropList, определенной как указатель на массив PPropList, который, в свою очередь, определен в модуле TypeInfo.pas:

```
type
  PPropList = ^TPropList;
  TPropList = array[0..16379] of PPropInfo;
```

Массив TPropList хранит указатели на данные TPropInfo каждого свойства. Тип TPropInfo определен в модуле TypeInfo.pas следующим образом:

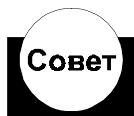
```
PropInfo = ^TPropInfo;
  TPropInfo = packed record
    PropType: PPTypeInfo;
    GetProc: Pointer;
    SetProc: Pointer;
    StoredProc: Pointer;
    Index: Integer;
    Default: Longint;
    NameIndex: SmallInt;
    Name: ShortString;
  end;
```

Процедура GetClassProperties() использует функцию GetPropInfos() для заполнения массива указателями на RTTI-информацию обо всех свойствах данного объекта. Затем, перебирая в цикле все элементы массива, считывает имена и типы свойств из соответствующих RTTI-данных. Обратите внимание на следующую строку:

```
if not (PropList[i]^.PropType^.Kind = tkMethod) then
```

Таким образом отфильтровываются свойства, являющиеся событиями (указатели на методы). Эти свойства мы добавим в соответствующий список позднее, попутно продемонстрировав альтернативный метод получения RTTI-информации о свойстве. В заключительной части

метода `GetClassProperties()` используется функция `GetPropList()` для получения массива `TPropList` для свойств конкретного типа. В нашем случае представляют интерес лишь свойства типа `tkMethod`. Функция `GetPropList()` также определена в модуле `TypeInfo.pas` (обратите внимание на комментарии в исходном коде).



Используйте функцию `GetPropInfos()`, если хотите получить указатель на RTTI-информацию обо всех свойствах данного объекта. Для получения информации о свойствах определенного типа используйте функцию `GetPropList()`.

На рис. 20.2 изображена главная форма с информацией о типах времени выполнения.

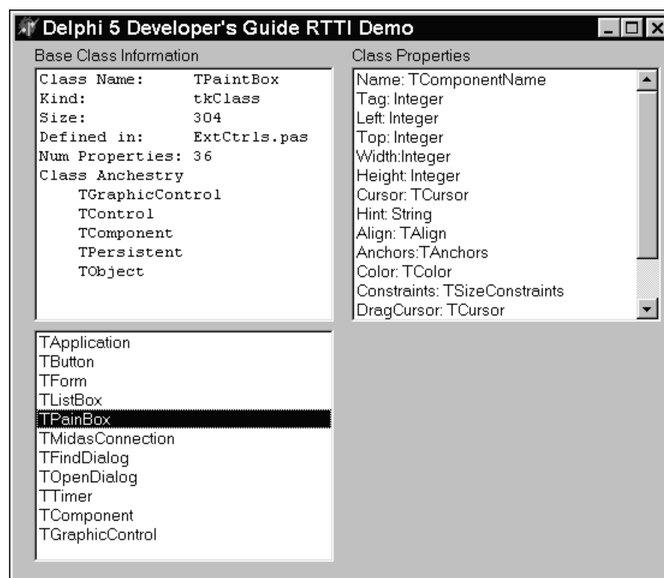


Рис. 20.2. Главная форма с информацией о типах

Проверка наличия некоторого свойства у объекта

Ранее была поставлена задача проверки существования свойства данного объекта. Тогда речь шла о свойстве `DataSource`. Используя функции, определенные в модуле `TypInfo.pas`, можно написать функцию проверки того, предназначен ли элемент управления для работы с данными:

```
function IsDataAware(AComponent: TComponent): Boolean;
var
  PropInfo: PPropInfo;
begin
  // Отыскивается свойство с именем DataSource
  PropInfo := GetPropInfo(AComponent.ClassInfo, 'DataSource');
  Result := PropInfo <> nil;

  // Двойная проверка, является ли свойство потомком TDataSource
```

```
  if Result then
    if not ((PropInfo^.Proptype^.Kind = tkClass) and
       (GetTypeData(PropInfo^.PropType^).ClassType.InheritsFrom(TDataSource)))
then
    Result := False;
end;
```

Здесь используется функция `GetPropInfo()`, возвращающая указатель `TPropInfo` на данное свойство. Эта функция возвращает значение `nil`, если свойства не существует. Для дополнительной проверки необходимо удостовериться, что свойство `DataSource` действительно является потомком класса `TDataSource`.

Эту функцию можно усовершенствовать так, чтобы она проверяла наличие свойства с любым заданным именем:

```
function HasProperty(AComponent: TComponent; APropertyName: String): Boolean;
var
  PropInfo: PPropInfo;
begin
  PropInfo := GetPropInfo(AComponent.ClassInfo, APropertyName);
  Result := PropInfo <> nil;
end;
```

Обратите внимание, что такой подход применим только к публикуемым свойствам. RTTI-данных нет для непубликуемых свойств.

Получение информации о типах указателей на методы

С помощью RTTI можно также получить информацию о типах указателей на методы. Например, вы можете получить тип метода (процедуры, функции и т.п.) и его параметры. В листинге 20.4 показано, как получить RTTI-информацию для выбранной группы методов.

Листинг 20.4. Получение RTTI-информации для методов

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls, ExtCtrls, DBClient, MidasCon, MConnect;

type

  TMainForm = class(TForm)
    lbSampMethods: TListBox;
    lbMethodInfo: TMemo;
    lblBasicMethodInfo: TLabel;
    procedure FormCreate(Sender: TObject);
```

```
    procedure lbSampMethodsClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation
uses TypInfo, DBTables, Provider;

{$R *.DFM}

type
  //Необходимо переопределить эту запись,
  // так как она закоментирована в typinfo.pas.

  PParamRecord = ^TParamRecord;
  TParamRecord = record
    Flags:     TParamFlags;
    ParamName: ShortString;
    TypeName:  ShortString;
  end;

procedure GetBaseMethodInfo(ATypeInfo: PTypeInfo; AStrings: TStrings);
{ Этот метод получает некоторые основные RTTI-данные из структуры
  TTypeInfo и помещает их в параметр AStrings. }
var
  MethodTypeData: PTypeData;
  EnumName: String;
begin
  MethodTypeData := GetTypeData(ATypeInfo);
  with AStrings do
  begin
    Add(Format('Class Name:     %s', [ATypeInfo^.Name]));
    EnumName := GetEnumName(TypeInfo(TTypeKind), Integer(ATypeInfo^.Kind));
    Add(Format('Kind:           %s', [EnumName]));
    Add(Format('Num Parameters: %d',[MethodTypeData.ParamCount]));
  end;
end;

procedure GetMethodDefinition(ATypeInfo: PTypeInfo; AStrings: TStrings);
{ Этот метод позволяет получить информацию об указателе метода.
  Мы используем эту информацию для реконструкции определения метода. }
var
  MethodTypeData: PTypeData;
  MethodDefine:   String;
  ParamRecord:    PParamRecord;
```

```
  TypeStr:        ^ShortString;
  ReturnStr:      ^ShortString;
  i: integer;
begin
  MethodTypeData := GetTypeData(ATypeInfo);

  // Определение типа метода
  case MethodTypeData.MethodKind of
    mkProcedure:      MethodDefine := 'procedure ';
    mkFunction:       MethodDefine := 'function ';
    mkConstructor:    MethodDefine := 'constructor ';
    mkDestructor:     MethodDefine := 'destructor ';
    mkClassProcedure: MethodDefine := 'class procedure ';
    mkClassFunction:  MethodDefine := 'class function ';
  end;

  // Указание на первый параметр
  ParamRecord    := @MethodTypeData.ParamList;
  i := 1; // первый параметр

  // Перебор параметров метода и, если они нормально определены,
  // добавление их в список строк.
  while i <= MethodTypeData.ParamCount do
  begin
    if i = 1 then
      MethodDefine := MethodDefine+'(';

    if pfVar in ParamRecord.Flags then
      MethodDefine := MethodDefine+('var ');
    if pfconst in ParamRecord.Flags then
      MethodDefine := MethodDefine+('const ');
    if pfArray in ParamRecord.Flags then
      MethodDefine := MethodDefine+('array of ');
      { Мы не будем изменять pfAddress, а лишь убедимся в том, что
        параметр Self передается с установленным флагом. }
{
    if pfAddress in ParamRecord.Flags then
      MethodDefine := MethodDefine+('*address* ');
}
    if pfout in ParamRecord.Flags then
      MethodDefine := MethodDefine+('out ');

    // Использование арифметики указателей для определения строки типа параметра
    TypeStr := Pointer(Integer(@ParamRecord^.ParamName) +
      Length(ParamRecord^.ParamName)+1);

    MethodDefine := Format('%s%s: %s', [MethodDefine, ParamRecord^.ParamName,
TypeStr^]);

    inc(i); // Приращение счетчика
```

```
     // Переход к следующему параметру
     // Расчет положения следующего параметра
    ParamRecord := PParamRecord(Integer(ParamRecord) + SizeOf(TParamFlags) +
      (Length(ParamRecord^.ParamName) + 1) + (Length(TypeStr^)+1));

    // Если еще есть параметры, продолжаем
    if i <= MethodTypeData.ParamCount then
    begin
      MethodDefine := MethodDefine + '; ';
    end
    else
      MethodDefine := MethodDefine + ')';
  end;

  // Если метод - функция, то он должен возвращать значение
  // Оно также помещается в строку определения метода
  // Возвращаемое значение следует за последним параметром
  if MethodTypeData.MethodKind = mkFunction then
  begin
    ReturnStr := Pointer(ParamRecord);
    MethodDefine := Format('%s: %s;', [MethodDefine, ReturnStr^])
  end
  else
    MethodDefine := MethodDefine+';';

  // Наконец, добавляем строку в список
  with AStrings do
  begin
    Add(MethodDefine)
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  { Добавление некоторых типов методов в список, а также
   сохранение указателя на данные RTTI в массиве списка Objects }
  with lbSampMethods.Items do
  begin
    AddObject('TNotifyEvent', TypeInfo(TNotifyEvent));
    AddObject('TMouseEvent', TypeInfo(TMouseEvent));
    AddObject('TBDECallBackEvent', TypeInfo(TBDECallBackEvent));
    AddObject('TDataRequestEvent', TypeInfo(TDataRequestEvent));
    AddObject('TGetModuleProc', TypeInfo(TGetModuleProc));
    AddObject('TReaderError', TypeInfo(TReaderError));
  end;
end;

procedure TMainForm.lbSampMethodsClick(Sender: TObject);
begin
  lbMethodInfo.Lines.Clear;
```

```
  with lbSampMethods do
  begin
    GetBaseMethodInfo(PTypeInfo(Items.Objects[ItemIndex]), lbMethodInfo.Lines);
    GetMethodDefinition(PTypeInfo(Items.Objects[ItemIndex]), lbMethodInfo.Lines);
  end;
end;

end.
```

Как видно из листинга 20.4, в список lbSampMethods помещены имена методов использованного объекта. Указатели на RTTI этих методов находятся в массиве списка Objects. Они получены с помощью функции TypeInfo(), предназначенной для отыскания указателя на информацию о типах времени выполнения для заданного идентификатора типа. Если пользователь выбирает один из этих методов, данные RTTI из массива Objects используются для реконструкции определения метода. На рис. 20.3 изображена форма с выбранным методом.

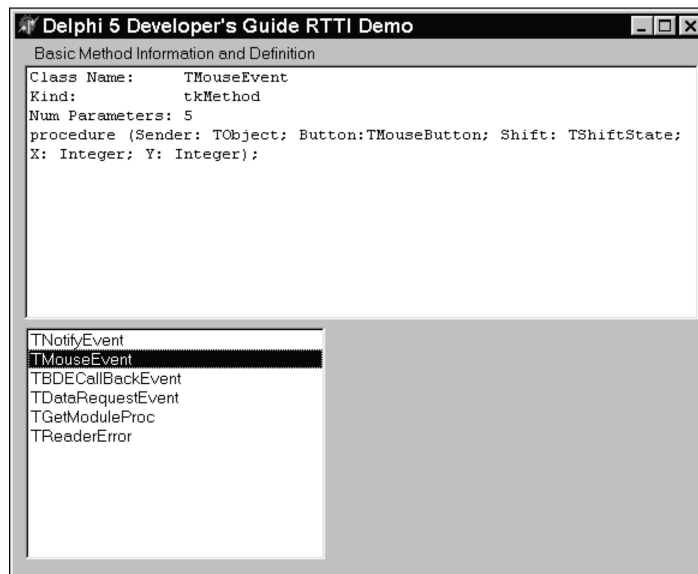
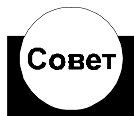


Рис. 20.3. Вывод информации о методе



Применяйте функцию TypeInfo() для получения указателя на генерируемую компилятором информацию RTTI для определенного идентификатора типа. Например, следующая строка позволяет получить указатель на RTTI-информацию для типа TButton:

```
TypeInfoPointer := TypeInfo(TButton);
```

Получение RTTI-информации для упорядоченных типов

Мы уже рассмотрели наиболее сложную часть RTTI. Для более простых типов также можно получить информацию о типах времени выполнения. В следующих разделах показано, как получить RTTI-данные для целого и перечислимого типов, а также для множества.

RTTI-информация для целочисленных типов

Получить RTTI-информацию для целочисленных типов очень просто. Листинг 20.5 иллюстрирует этот процесс.

Листинг 20.5. Получение RTTI-информации для целочисленных типов

```
procedure TMainForm.lbSampsClick(Sender: TObject);
var
  OrdTypeInfo: PTypeInfo;
  OrdTypeData: PTypeData;

  TypeNameStr: String;
  TypeKindStr: String;
  MinVal, MaxVal: Integer;
begin
  memInfo.Lines.Clear;
  with lbSamps do
  begin

    // Получаем указатель на структуру TTypeInfo
    OrdTypeInfo := PTypeInfo(Items.Objects[ItemIndex]);
    // Получаем указатель на структуру TTypeData
    OrdTypeData := GetTypeData(OrdTypeInfo);

    // Получаем строку с именем типа
    TypeNameStr := OrdTypeInfo.Name;
    // Получаем строку с разновидностью типа
    TypeKindStr := GetEnumName(TypeInfo(TTypeKind), BInteger(OrdTypeInfo^.Kind));

    // Получаем минимальные и максимальные значения для типа
    MinVal := OrdTypeData^.MinValue;
    MaxVal := OrdTypeData^.MaxValue;

    // Добавляем информацию в memo-поле
    with memInfo.Lines do
    begin
      Add('Type Name: '+TypeNameStr);
      Add('Type Kind: '+TypeKindStr);

      Add('Min Val: '+IntToStr(MinVal));
      Add('Max Val: '+IntToStr(MaxVal));
    end;
  end;
end;
```

Здесь функция `TypeInfo()`используется для получения указателя на структуру `TTypeInfo` для типа данных `Integer`. Затем этот указатель передается функции `GetTypeData()` для получения указателя на структуру `TTypeData`. Обе эти структуры используются для помещения в список RTTI-информации для целого типа. Более подробный демонстрационный вариант `IntegerRTTI.dpr` можно найти на прилагаемом компакт-диске, в каталоге, относящемся к данной главе.

RTTI-информация для перечислимых типов

Получить RTTI-информацию для перечислимых типов также несложно. Как видите, листинг 20.6 практически идентичен листингу 20.5, за исключением дополнительного цикла for для отображения значений перечислимого типа.

Листинг 20.6. Получение RTTI-информации для перечислимого типа

```
procedure TMainForm.lbSampsClick(Sender: TObject);
var
  OrdTypeInfo: PTypeInfo;
  OrdTypeData: PTypeData;

  TypeNameStr: String;
  TypeKindStr: String;
  MinVal, MaxVal: Integer;
  i: integer;
begin
  memInfo.Lines.Clear;
  with lbSamps do
  begin

    // Получаем указатель на структуру TTypeInfo
    OrdTypeInfo := PTypeInfo(Items.Objects[ItemIndex]);
    // Получаем указатель на структуру TTypeData
    OrdTypeData := GetTypeData(OrdTypeInfo);

    // Получаем строку с именем типа
    TypeNameStr := OrdTypeInfo.Name;
    // Получаем строку с разновидностью типа
    TypeKindStr := GetEnumName(TypeInfo(TTypeKind),
                   ↳Integer(OrdTypeInfo^.Kind));

    // Получаем минимальное и максимальное значения типа
    MinVal := OrdTypeData^.MinValue;
    MaxVal := OrdTypeData^.MaxValue;


    // Добавляем информацию в memo-поле
    with memInfo.Lines do
    begin
      Add('Type Name: '+TypeNameStr);
      Add('Type Kind: '+TypeKindStr);

      Add('Min Val: '+IntToStr(MinVal));
      Add('Max Val: '+IntToStr(MaxVal));

      // Отображаем значения и имена перечислимых типов
      if OrdTypeInfo^.Kind = tkEnumeration then
        for i := MinVal to MaxVal do
```

```
        Add(Format('  Value: %d   Name: %s', [i,
          GetEnumName(OrdTypeInfo, i)]));

    end;
  end;
end;
```

Более подробный демонстрационный вариант EnumRTTI.dpr можно найти на прилагаемом компакт-диске, в каталоге, относящемся к данной главе.

RTTI-информация для множеств

Получение RTTI-информации для множества ненамного сложнее предыдущих технологий. В листинге 20.7 представлен программный код, обеспечивающий функционирование главной формы проекта SetRTTI.dpr, который можно найти на прилагаемом компакт-диске, в каталоге, относящемся к данной главе.

Листинг 20.7. Получение RTTI-информации для множеств

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Grids;

type
  TMainForm = class(TForm)
    lbSamps: TListBox;
    memInfo: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure lbSampsClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation
uses TypInfo, Buttons;

{$R *.DFM}


procedure TMainForm.FormCreate(Sender: TObject);
begin
```

```
  // Добавление некоторых примеров перечислимых типов
  with lbSamps.Items do
 begin
    AddObject('TBorderIcons', TypeInfo(TBorderIcons));
    AddObject('TGridOptions', TypeInfo(TGridOptions));
  end;
end;

procedure GetTypeInfoForOrdinal(AOrdTypeInfo: PTypeInfo; AStrings: TStrings);
var
//  OrdTypeInfo: PTypeInfo;
  OrdTypeData: PTypeData;

  TypeNameStr: String;
  TypeKindStr: String;
  MinVal, MaxVal: Integer;
  i: integer;
begin

  // Получение указателя на структуру TTypeData
  OrdTypeData := GetTypeData(AOrdTypeInfo);

  // Получение строки с именем типа
  TypeNameStr := AOrdTypeInfo.Name;
  // Получение строки с разновидностью типа
  TypeKindStr := GetEnumName(TypeInfo(TTypeKind), Integer(AOrdTypeInfo^.Kind));

  // Получение минимального и максимального значений типа
  MinVal := OrdTypeData^.MinValue;
  MaxVal := OrdTypeData^.MaxValue;


  // Добавление информации в список
  with AStrings do
  begin
    Add('Type Name: '+TypeNameStr);
    Add('Type Kind: '+TypeKindStr);

    // Рекурсивный вызов функции для отображения
    // перечислимых значений множества
    if AOrdTypeInfo^.Kind = tkSet then
    begin
      Add('==========');
      Add('');
      GetTypeInfoForOrdinal(OrdTypeData^.CompType^, AStrings);
    end;

    // Отображение значения и имени перечислимых типов,
    // принадлежащих множеству
    if AOrdTypeInfo^.Kind = tkEnumeration then
```

```
    begin
      Add('Min Val: '+IntToStr(MinVal));
      Add('Max Val: '+IntToStr(MaxVal));

      for i := MinVal to MaxVal do
        Add(Format('  Value: %d   Name: %s', [i, GetEnumName(AOrdTypeInfo, i)]));
    end;
  end;

end;

procedure TMainForm.lbSampsClick(Sender: TObject);
begin
  memInfo.Lines.Clear;
  with lbSamps do
    GetTypeInfoForOrdinal(PTypeInfo(Items.Objects[ItemIndex]), memInfo.Lines);
end;
end.
```

В список этой демонстрационной программы мы поместили в два типа множества. Указатель на структуры `TTypeInfo` для этих двух типов был помещен в массив `Objects` с помощью функции `TypeInfo()`. Когда пользователь выбирает из списка один из этих элементов, вызывается процедура `GetTypeInfoForOrdinal()`, которой передаются в качестве параметров как указатель типа `PTypeInfo`, так и свойство `memInfo.Lines` для заполнения RTTI-данными.

Процедура `GetTypeInfoForOrdinal()` выполняет стандартные действия для получения указателя на структуру `TTypeData`, принадлежащую заданному типу. Начальная информация о типе хранится в параметре типа `TStrings`, а затем происходит рекурсивный вызов той же процедуры `GetTypeInfoForOrdinal()` с передачей в качестве первого параметра значения `OrdTypeData^.CompType^` — указателя на перечислимый тип множества. Получаемые при этом RTTI-данные также добавляются в то же самое свойство типа `TStrings`.

Присваивание значений свойствам с помощью RTTI

После того как мы показали, как найти и определить, какие свойства компонентов существуют, вам будет весьма полезно узнать, как с помощью RTTI-информации присвоить значения этим свойствам. Эта задача довольно проста. В модуле `TypInfo.pas` содержится много вспомогательных процедур, позволяющих опрашивать и устанавливать публикуемые свойства компонентов. Речь идет о тех же самых вспомогательных процедурах, которые используются интегрированной средой Delphi (IDE) (в окне Object Inspector). Вам совсем не помешало бы открыть модуль `TypInfo.pas` и ознакомиться с этими процедурами. Некоторые из них мы покажем ниже.

Предположим, что требуется присвоить целочисленное значение некоторому свойству данного компонента. При этом допустим, что неизвестно, существует ли это свойство в рассматриваемом компоненте. Вот как выглядит процедура присваивания целочисленного значения свойству данного компонента, при условии, что это свойство существует:

```
procedure SetIntegerPropertyIfExists(AComp: TComponent; APropName: String;
  AValue: Integer);
var
  PropInfo: PPropInfo;
begin
```

```
  PropInfo := GetPropInfo(AComp.ClassInfo, APropName);
  if PropInfo <> nil then
  begin
    if PropInfo^.PropType^.Kind = tkInteger then
      SetOrdProp(AComp, PropInfo, Integer(AValue));
  end;
end;
```

Эта процедура принимает три параметра. Первый параметр, AComp, представляет собой компонент, свойство которого мы хотим модифицировать. Второй параметр, APropName, — это имя свойства, которому мы хотим присвоить значение третьего параметра, AValue. В этой процедуре используется функция GetPropInfo(), предназначенная для считывания указателя TPropInfo на заданное свойство. Если этого свойства не существует, функция GetPropInfo() вернет значение nil. Если же свойство существует, то с помощью второго оператора if определяется корректность типа заданного свойства. Тип свойства tkInteger определяется в модуле TypInfo.pas вместе с другими возможными типами свойств, как показано ниже:

```
TTypeKind = (tkUnknown, tkInteger, tkChar, tkEnumeration, tkFloat,
    tkString, tkSet, tkClass, tkMethod, tkWChar, tkLString, tkWString,
    tkVariant, tkArray, tkRecord, tkInterface, tkInt64, tkDynArray);
```

Наконец, присваивание значения свойству выполняется с использованием еще одной процедуры SetOrdProp() из модуля TypInfo.pas, которая как раз и предназначена для установки значений свойств порядкового типа. Обращение к этой процедуре может выглядеть примерно следующим образом:

```
SetIntegerPropertyIfExists(Button2, 'Width', 50);
```

Метод SetOrdProp() называют методом установки, поскольку он устанавливает заданное свойство равным заданному значению. Существует также и метод чтения значения свойства. В модуле TypInfo.pas есть несколько таких вспомогательных процедур с соответствующими именами (SetXXXProp()), предусмотренных для всех возможных типов свойств (табл. 20.6).

Таблица 20.6. Методы чтения и установки свойств

Тип свойства	Метод установки	Метод чтения
Порядковый	SetOrdProp()	GetOrdProp()
Перечислимый	SetEnumProp()	GetEnumProp()
Объект	SetObjectProp()	GetObjectProp()
Строка	SetStrProp()	GetStrProp()
С плавающей запятой	SetFloatProp()	GetFloatProp()
Вариантный	SetVariantProp()	GetVariantProp()
Методы (События)	SetMethodProp()	GetMethodProp()
Int64	SetInt64Prop()	GetInt64Prop()

И опять обращаю ваше внимание на то, что в модуле TypInfo.pas содержится много других вспомогательных методов, которые могут вам очень пригодиться.

В следующем фрагменте показано, как присваивается значение свойству объекта:

```
procedure SetObjectPropertyIfExists(AComponent: TComponent; APropName: String;
  AValue: TObject);
var
  PropInfo: PPropInfo;
begin
  PropInfo := GetPropInfo(AComponent.ClassInfo, APropName);
  if PropInfo <> nil then
  begin
    if PropInfo^.PropType^.Kind = tkClass then
      SetObjectProp(AComponent, PropInfo, AValue);
  end;
end;
```

Для вызова этого метода можно использовать следующий вариант кода:

```
var
  F: TFont;
begin
  F := TFont.Create;
  F.Name   := 'Arial';
  F.Size   := 24;
  F.Color  := clRed;
  SetObjectPropertyIfExists(Panel1, 'Font', F);
end;
```

А вот как можно присвоить значение свойству метода:

```
procedure SetMethodPropertyIfExists(AComp: TComponent; APropName: String;
  AMethod: TMethod);
var
  PropInfo: PPropInfo;
begin
  PropInfo := GetPropInfo(AComp.ClassInfo, APropName);
  if PropInfo <> nil then
  begin
    if PropInfo^.PropType^.Kind = tkMethod then
      SetMethodProp(AComp, PropInfo, AMethod);
  end;
end;
```

Этот метод требует использования типа TMethod, который определен в модуле SysUtils.pas. Чтобы вызвать метод, назначающий обработчик события одного компонента другому, можно использовать метод GetMethodProp, считывающий значение TMethod из исходного компонента:

```
SetMethodPropertyIfExists(Button5, 'OnClick',
    GetMethodProp(Panel1, 'OnClick'));
```

На прилагаемом компакт-диске есть проект SetProperties.dpr, демонстрирующий использование этих методов.

Резюме

В этой главе вы познакомились с библиотекой VCL. Были рассмотрены иерархия классов VCL и характеристики компонентов различных уровней иерархии. Кроме того, подробно обсуждалась информация о типах времени выполнения (RTTI).

Глава

21 Создание пользовательских компонентов в Delphi

Основные концепции разработки компонентов 63

Примеры разработки компонентов 87

Компонент-контейнер `TddgButtonEdit` 103

Пакеты компонентов 113

Пакеты надстроек 122

Резюме 129

Возможность разрабатывать нужные компоненты в Delphi 5 дает вам неоценимое преимущество перед другими программистами. Пользователи большинства других сред программирования вынуждены довольствоваться стандартными элементами управления Windows или же приобретать совершенно иной набор сложных элементов управления, разработанных сторонними производителями. Создавая пользовательские компоненты и встраивая их в приложение Delphi, вы получаете полный контроль над интерфейсом пользователя в любых разрабатываемых приложениях. Применение таких элементов управления оставляет за вами последнее слово в вопросах внешнего вида программы и производимого ею впечатления.

Если разработку компонентов вы решили сделать своим коньком, то данная глава, несомненно, заинтересует вас. Мы проанализируем все этапы разработки компонента — от создания концепции до интеграции готового компонента в среду Delphi, — отметим все подводные камни на этом пути, а также снабдим вас ценными советами и поделимся разнообразными приемами, необходимыми для разработки высокоорганизованных и расширяемых компонентов.

Даже если основная сфера ваших интересов — разработка не компонентов, а приложений, вы все равно сможете многое почерпнуть, читая эту главу. Используя в программе один-два собственных компонента, можно придать приложению некую изюминку и повысить его продуктивность. Рано или поздно вы обязательно столкнетесь с ситуацией, когда ни один из компонентов, имеющихся в вашем распоряжении, не будет удовлетворять поставленным задачам. Тогда-то вам и пригодится умение создавать собственные компоненты. Вы сможете построить компонент, идеально соответствующий вашим требованиям, а затем многократно использовать его во всех своих программах.

Основные концепции разработки компонентов

Прочитав этот раздел, вы приобретете основные навыки, необходимые для создания собственного компонента. На примерах создания некоторых полезных компонентов мы покажем, как применить эти навыки на практике.

Решение о необходимости создания компонента

Зачем мучиться, создавая собственный новый компонент, когда можно воспользоваться существующими или слепить на скорую руку что-нибудь попроще из подсобных средств? Имеется несколько важных причин для разработки нового компонента.

- Требуется разработать новый элемент пользовательского интерфейса и в дальнейшем использовать его в разных приложениях.
- Необходимо сделать свое приложение устойчивым к ошибкам, разделив его на логические элементы в виде объектно-ориентированных классов.
- Среди существующих компонентов Delphi и элементов ActiveX нет такого, который полностью удовлетворял бы всем вашим требованиям.
- Имеются потенциальные пользователи создаваемого компонента и желательно распространить его среди других программистов либо за деньги, либо ради собственного удовольствия.
- Вы хотите глубже разобраться в Delphi, библиотеке VCL и функциях Win32 API.

Лучше всего учиться создавать пользовательские компоненты у тех, кто их изобрел. Исходный код библиотеки VCL — настоящая сокровищница знаний, и мы настоятельно рекомендуем изучить его, особенно если вы серьезно решили заняться созданием компонентов. Исходный текст компонентов библиотеки VCL включен в обе версии Delphi — и в Professional, и в Enterprise.

Многих пугает сама мысль о создании своих компонентов, и, поверьте, совершенно напрасно. Легко это или тяжело — зависит только от вас. Конечно, существуют очень сложные компоненты, но, в принципе, создать полезный компонент довольно легко.

Этапы разработки компонента

Если вы уже определили проблему и наметили способ ее решения, основанный на использовании некоторого компонента, создание такого компонента — от концепции до ее воплощения — будет происходить в несколько этапов.

- Прежде всего нужна идея полезного компонента, отличного от уже существующих.
- Затем необходимо построить алгоритм, по которому компонент будет работать.
- Начните с подготовительных мероприятий, не бросайтесь сразу с головой в проектирование компонента. Спросите себя: "Что нужно сделать, чтобы заставить этот компонент функционировать?"
- Мысленно разложите конструкцию компонента на независимые логические части. Это не только упростит и упорядочит создание компонента, но и позволит написать ясный программный код со стройной организацией. При разработке компонента имейте в виду, что, возможно, кто-то захочет создать на его базе компонент-потомок.
- Сначала протестируйте компонент в специально написанном для этого тестовом проекте. Если вы сразу добавите его в палитру компонентов, то, вероятнее всего, будете разочарованы.
- Наконец, добавьте компонент и его пиктограмму в палитру компонентов. После небольшой настройки он будет готов для переноса в приложения Delphi.

Вот шесть основных этапов создания компонента Delphi.

1. Выбор класса-предка.
2. Создание модуля компонента.
3. Добавление в новый компонент свойств, методов и событий.
4. Тестирование.
5. Регистрация компонента в среде Delphi.
6. Создание файла справки для компонента.

В этой главе рассматриваются первых пять этапов — создание файлов справки выходит за ее рамки. Однако это не значит, что шестой этап менее важный, чем предыдущие. Рекомендуем изучить распространенные программные средства создания файлов справки. Фирма Borland также поместила необходимую информацию по данному вопросу в собственную интерактивную справочную систему. Обратитесь к разделу "Providing Help for Your Component" ("Создание справочной информации для пользовательского компонента") этой системы.

Выбор класса-предка

В главе 20, "Ключевые элементы VCL и информация о типах времени выполнения", обсуждалась иерархия компонентов библиотеки VCL и назначение различных классов, расположенных на разных уровнях этой иерархии. Были описаны четыре базовых компонента, от

которых может происходить ваш компонент: стандартные элементы управления, пользовательские элементы управления, графические элементы управления и невизуальные компоненты. Например, если требуется просто расширить поведение существующего элемента Win32, скажем, TMemo, следует создать компонент путем расширения возможностей стандартного. Если необходимо создать абсолютно новый компонентный класс, вам придется иметь дело с пользовательским элементом управления. Компоненты на основе графических элементов применяются для создания визуальных эффектов, реализуемых без использования ресурсов Win32. И, наконец, если требуется создать не имеющий визуальных характеристик компонент, который, тем не менее, можно будет редактировать в окне инспектора объектов, то вам предстоит разработать невизуальный компонент. Различные классы библиотеки VCL представляют разные типы компонентов. Возможно, вам придется возвращаться к главе 20, "Ключевые элементы VCL и информация о типах времени выполнения" до тех пор, пока вы не освоитесь с этими концепциями. В табл. 21.1 содержится лишь краткая справка.

Таблица 21.1. Базовые классы библиотеки VCL

Класс VCL	Тип пользовательских элементов управления
TObject	Хотя классы, непосредственно происходящие от класса TObject, не являются компонентами, тем не менее они заслуживают внимания. Класс TObject используется как базовый для создания многих объектов, с которыми вам едва ли придется сталкиваться в процессе разработки. В качестве примера можно привести класс TIniFile
TComponent	Это исходная точка для многих невизуальных компонентов. Основным его достоинством является возможность самостоятельно загружать и сохранять потоки данных в интегрированной среде Delphi во время разработки
TGraphicControl	Используйте этот класс для создания пользовательского компонента, не имеющего дескриптора окна. Потомки класса TGraphicControl отображаются в клиентской области своих родительских элементов и поэтому не требуют использования системных ресурсов
TWinControl	Это базовый класс для создания всех компонентов, обладающих дескриптором окна. Этот класс предоставляет общие свойства и события, характерные для оконных элементов управления
TCustomControl	Этот класс происходит от класса TWinControl. Он вводит концепцию канвы и содержит метод Paint(), предоставляющий расширенный контроль над внешним видом вашего компонента. Этот класс используется в основном для создания пользовательских оконных элементов
TCustom*ClassName*	В библиотеке VCL содержится несколько классов, не все свойства которых являются публикуемыми. То же самое они разрешают делать и своим классам-потомкам. Это позволяет разработчикам создать несколько "пользовательских" компонентов на основе одного и того же класса и в каждом пользовательском классе публиковать только заранее определенные свойства, необходимые для данного конкретного класса
TComponent*Name*	Некоторый существующий класс, подобный TEdit, TPanel или TScrollBox. Если требуется расширить поведение какого-нибудь элемента управления, то в качестве базового для своего класса используйте существующий компонент, например TEdit или иной подходящий пользовательский компонент, а не создавайте новый элемент с нуля. Большинство создаваемых вами компонентов попадет именно в эту категорию

Очень важно понимать особенности различных классов и возможности имеющихся компонентов. В большинстве случаев вы обнаружите, что имеющиеся классы смогут предоставить бо́льшую часть необходимой новому компоненту функциональности. Только досконально изучив возможности существующих компонентов, вы сможете выбрать предка для нового компонента. Эти сведения относятся к разряду знаний, приобретаемых лишь опытным путем, поэтому будет недостаточно просто прочесть эту книгу. Придется приложить значительные усилия для исследования каждого компонента и класса библиотеки VCL Delphi. Единственный способ для этого — использовать компоненты библиотеки VCL в своих приложениях, пусть даже в чисто экспериментальных целях.

Создание модуля компонента

Определившись с выбором компонента-предка для своего нового компонента, можно приступать к созданию модуля этого компонента. В следующих нескольких разделах мы последовательно пройдем все этапы разработки нового компонента. При этом основное внимание будет уделено сущности этих этапов, а не конкретной функциональности, поэтому функциональность создаваемого нами в качестве примера компонента будет минимальной, но вполне достаточной для иллюстрации этих этапов разработки.

Этот компонент будет называться `TddgWorthless`. Он является потомком класса `TCustomControl`, а значит, имеет дескриптор окна и способен отображать себя. Этот компонент унаследовал несколько свойств, методов и событий от компонента `TCustomControl`.

Создание модуля компонента проще всего начать с вызова окна New Component (Новый компонент), показанного на рис. 21.1.

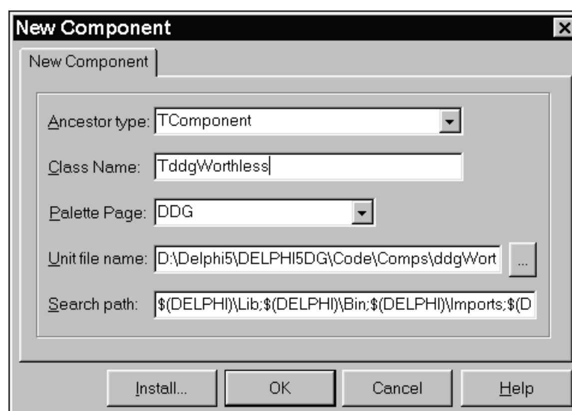


Рис. 21.1. Окно New Component

Окно New Component можно вывести на экран с помощью команды Component⇨New Component. Введите в нем имя класса-предка, имя класса компонента, вкладку палитры, на которой должен отображаться ваш компонент, и имя модуля компонента. Затем щелкните на кнопке OK — и Delphi автоматически создаст модуль компонента с готовым объявлением типа компонента и процедурой регистрации. В листинге 21.1 приведен модуль, созданный системными средствами Delphi.

Листинг 21.1. Модуль `Worthless.pas` — пример компонента Delphi

```
unit Worthless;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
type
  TddgWorthless = class(TCustomControl)
  private
    { Закрытые объявления }
  protected
    { Защищенные объявления }
  public
    { Открытые объявления }
  published
    { Публикуемые объявления }
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('DDG', [TddgWorthless]);
end;
end.
```

Как видите, пока класс `TddgWorthless` — просто заготовка компонента. В последующих разделах в класс `TddgWorthless` будут добавлены свойства, методы и события.

Создание свойств

В главе 20, "Ключевые элементы VCL и информация о типах времени выполнения", показаны преимущества использования свойств компонентов. В этом разделе мы обсудим, как добавлять различные типы свойств к создаваемым компонентам.

Типы свойств

В табл. 20.1, приведенной в главе 20, "Ключевые элементы VCL и информация о типах времени выполнения", описаны различные типы свойств. Добавим в компонент `TddgWorthless` свойства каждого типа, чтобы проиллюстрировать различия между ними. Каждый тип по-разному редактируется в окне инспектора объектов. Ниже мы подробно рассмотрим эти типы и способы их редактирования.

Добавление в компонент простых свойств

Под простыми свойствами следует понимать числа, строки и символы. Они могут непосредственно редактироваться пользователем в окне инспектора объектов и не требуют специальных методов доступа. В листинге 21.2 представлен вариант компонента `TddgWorthless` с тремя простыми свойствами.

Листинг 21.2. Простые свойства

```
TddgWorthless = class(TCustomControl)
  private
    // Внутренние данные
    FIntegerProp: Integer;
    FStringProp: String;
    FCharProp: Char;
  published
    // Простые типы свойств
    property IntegerProp: Integer read FIntegerProp write FIntegerProp;
    property StringProp: String read FStringProp write FStringProp;
    property CharProp: Char read FCharProp write FCharProp;
  end;
```

Используемый здесь синтаксис вам должен быть уже знаком, поскольку он подробно описан в главе 20, "Ключевые элементы VCL и информация о типах времени выполнения". Здесь представлены внутренние данные компонента, объявленные в разделе private. Свойства, относящиеся к этим полям, объявлены в разделе published; это означает, что при установке компонента в Delphi вы сможете редактировать свойства в окне инспектора объектов.

На заметку

При создании компонентов используется соглашение о начале имен полей с буквы F. Имена компонентов и вообще типов начинайте с буквы T. В вашем коде будет значительно проще разобраться, если вы будете соблюдать эти несложные соглашения.

Добавление в компонент перечислимых свойств

Определенные пользователем перечислимые и булевы свойства также можно редактировать в окне инспектора объектов. Для этого нужно дважды щелкнуть на поле Value этого окна и выбрать подходящее значение свойства в раскрывающемся списке. В качестве примера можно привести свойство Align, присущее большинству визуальных компонентов. Для создания свойства перечислимого типа сначала нужно определить сам перечислимый тип, например, таким образом:

```
TEnumProp = (epZero, epOne, epTwo, epThree);
```

Затем следует определить внутреннее поле для хранения значения, задаваемого пользователем. В листинге 21.3 демонстрируются два перечислимых свойства компонента TddgWorthless.

Листинг 21.3. Свойства перечислимого типа

```
TddgWorthless = class(TCustomControl)
  private
    // Перечислимые типы данных
    FEnumProp: TEnumProp;
    FBooleanProp: Boolean;
  published
    property EnumProp: TEnumProp read FEnumProp write FEnumProp;
    property BooleanProp: Boolean read FBooleanProp write FBooleanProp;
  end;
```

Для простоты остальные свойства исключены из компонента. Если бы вы инсталлировали этот компонент, его перечислимые свойства отобразились бы в окне инспектора объектов, как показано на рис. 21.2.

Добавление в компонент свойства типа множества

Свойство типа множества при редактировании в окне инспектора объектов выглядит как множество, определенное в синтаксисе языка Pascal. Простейший способ его редактирования — развернуть свойство в окне инспектора объектов, в результате чего каждый его элемент станет отдельным булевым свойством. Для создания свойства типа множества сначала нужно этот тип определить:

```
TSetPropOption = (poOne, poTwo, poThree, poFour, poFive);
TSetPropOptions = set of TSetPropOption;
```

Здесь вначале указывается диапазон множества путем определения соответствующего перечислимого типа, `TSetPropOption`, а затем уже определяется само множество `TSetPropOptions`.

Теперь можно добавить свойство `TSetPropOptions` в компонент `TddgWorthless`:

```
TddgWorthless = class(TCustomControl)
  private
    FOptions: TSetPropOptions;
  published
    property Options: TSetPropOptions read FOptions write FOptions;
end;
```

На рис. 21.3 показано, как выглядит это свойство в развернутом виде в окне инспектора объектов.

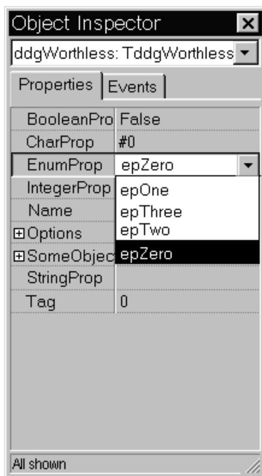


Рис. 21.2. Отображение перечислимых свойств компонента `TddgWorthless` в окне инспектора объектов

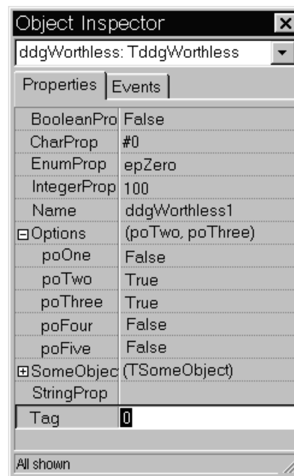


Рис. 21.3. Представление свойства типа множества в окне инспектора объектов

Добавление в компонент свойства-объекта

Свойства могут являться объектами или другими компонентами. Например, у компонента TShape есть свойства-объекты TBrush и TPen. Когда свойство является объектом, оно может быть развернуто в окне инспектора объектов таким образом, чтобы его собственные свойства также могли быть модифицированы. Свойства-объекты должны быть потомками класса TPersistent, для того чтобы их публикуемые свойства (т.е. свойства, объявленные в разделе published) могли быть записаны в поток данных и отображены в окне инспектора объектов.

Для определения объектного свойства компонента TddgWorthless необходимо сначала определить объект, который будет использован в качестве типа свойства. Объявление этого объекта приведено в листинге 21.4.

Листинг 21.4. Определение объекта TSomeObject

```
TSomeObject = class(TPersistent)
  private
    FProp1: Integer;
    FProp2: String;
  public
    procedure Assign(Source: TPersistent)
  published
    property Prop1: Integer read FProp1 write FProp1;
    property Prop2: String read FProp2 write FProp2;
  end;
```

Класс TSomeObject является прямым потомком класса TPersistent, но в общем случае это необязательно. Если объект, от которого происходит новый класс, является потомком класса TPersistent, то этот объект также может быть использован в качестве свойства другого объекта.

В нашем примере класс TSomeObject, используемый для создания объектного свойства, имеет два собственных простых свойства: Prop1 и Prop2. В него также включена процедура Assign(), назначение которой поясняется ниже.

Теперь можно добавить в компонент TddgWorthless внутреннее поле типа TSomeObject. Так как свойство представляет объект, его нужно создать. В противном случае, когда пользователь поместит в форму компонент TddgWorthless, того экземпляра класса TSomeObject, который он мог бы редактировать, просто не будет существовать. В листинге 21.5 показано объявление компонента TddgWorthless с его новым объектным свойством.

Листинг 21.5. Добавление свойств-объектов

```
TddgWorthless = class(TCustomControl)
private
  FSomeObject: TSomeObject;
  procedure SetSomeObject(Value: TSomeObject);
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
published
  property SomeObject: TSomeObject read FSomeObject write SetSomeObject;
end;
```

Обратите внимание на то, что в код включены переопределенные конструктор Create() и деструктор Destroy(). Кроме того, объявлен метод доступа SetSomeObject(), предназначенный для записи свойства SomeObject. Метод доступа "для записи" часто называют просто *методом записи* (или *методом установки*), а метод доступа "для чтения" — *методом чтения* (или *методом выборки*). Как указывалось в главе 20, "Ключевые элементы VCL и информация о типах времени выполнения", методы записи должны иметь один параметр такого же типа, что и свойство, которому они принадлежат. Существует соглашение начинать имена методов записи со слова Set.

Конструктор TddgWorthless.Create() определяем следующим образом:

```
constructor TddgWorthless.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FSomeObject := TSomeObject.Create;
end;
```

Здесь вначале был вызван унаследованный конструктор Create(), а затем создан экземпляр класса TSomeObject. Поскольку конструктор Create() вызывается как во время разработки, когда пользователь помещает компонент в форму, так и при выполнении приложения, вы можете быть уверены, что объект FSomeObject всегда действителен.

Кроме того, необходимо переопределить деструктор Destroy(), который перед освобождением компонента TddgWorthless должен предварительно освободить объект TSomeObject. Вот код переопределенного деструктора:

```
destructor TddgWorthless.Destroy;
begin
  FSomeObject.Free;
  inherited Destroy;
end;
```

Теперь, когда вы научились создавать экземпляр объекта TSomeObject, подумайте, что произойдет при выполнении следующего программного кода:

```
var
  MySomeObject: TSomeObject;
begin
  MySomeObject := TSomeObject.Create;
  ddgWorthless.SomeObject := MySomeObject;
end;
```

Если свойство TddgWorthless.SomeObject было бы определено без метода записи, подобного приведенному ниже, то при попытке пользователя присвоить полю SomeObject собственный объект, предыдущий экземпляр, на который ссылался FSomeObject, был бы потерян:

```
property SomeObject: TSomeObject read FSomeObject write FSomeObject;
```

Напомним, что экземпляры объектов — это на самом деле указатели, ссылающиеся на реальный объект (см. главу 2, "Язык программирования Object Pascal", том I). Выполняя присвоение, как в предыдущем примере, вы получаете указатель на другой экземпляр объекта, в то время как предыдущий экземпляр никуда из памяти не исчезает. При разработке компонентов обычно пытаются исключить необходимость строгого соблюдения пользователями формальных

правил при доступе к свойствам. Чтобы избежать этого подводного камня и защитить новый компонент от неумелого использования, для объектных свойств создают методы доступа. Это гарантирует, что при присвоении этим свойствам новых значений системные ресурсы не будут потеряны. Метод доступа к объекту SomeObject предназначен как раз для этого:

```
procedure TddgWorthLess.SetSomeObject(Value: TSomeObject);
begin
  if Assigned(Value) then
    FSomeObject.Assign(Value);
end;
```

Метод SetSomeObject() вызывает метод FSomeObject.Assign(), передавая ему указатель на новый объект TSomeObject. Вот как выглядит реализация метода TSomeObject.Assign():

```
procedure TSomeObject.Assign(Source: TPersistent);
begin
  if Source is TSomeObject then
  begin
    FProp1 := TSomeObject(Source).Prop1;
    FProp2 := TSomeObject(Source).Prop2;
    inherited Assign(Source);
  end;
end;
```

В методе TSomeObject.Assign() вначале выполняется проверка того, действительно ли пользователь в качестве параметра передал экземпляр объекта TSomeObject. Если это так, то из параметра Source копируются соответствующие значения свойств. Это еще один способ присвоения значений объектов другим объектам, используемым в библиотеке VCL. Если у вас есть исходный код компонентов VCL, вам стоит посмотреть на реализацию методов Assign() в различных объектах, например, таких как TBrush и TShape. Это может подсказать идею реализации метода Assign() для вашего компонента.

Никогда не присваивайте свойству значение внутри его собственного метода записи. Рассмотрим, например, следующее объявление свойства:

```
property SomeProp: integer read FSomeProp write SetSomeProp;
  ....
  procedure SetSomeProp(Value:integer);
  begin
    SomeProp := Value; {Это присваивание приведет к бесконечной
рекурсии }
  end;
```

Поскольку вы получаете доступ к самому свойству, а не ко внутреннему полю, происходит повторный вызов метода SetSomeProp(), приводящий к бесконечному циклу рекурсивных вызовов. В конце концов программа аварийно завершается из-за переполнения стека. Поэтому в методах записи свойств всегда работайте только с внутренними полями!

Добавление в компонент свойства-массива

Некоторые свойства позволяют выполнять к ним доступ таким образом, как если бы они представляли собой массивы. То есть, они содержат список элементов, к каждому из которых можно обращаться по его значению индекса. Эти элементы могут быть произвольного объектного типа.

Примерами могут служить такие свойства, как `TScreen.Fonts`, `TMemo.Lines` и `TDBGrid.Columns`. Эти свойства требуют наличия собственных редакторов свойств. На создании редакторов свойств мы остановимся в следующей главе, а сейчас покажем, как определить свойство, которое может быть проиндексировано как массив, но, тем не менее, совсем не содержит списков.

Оставим пока в стороне компонент `TddgWorthless` и обратимся к новому компоненту `TddgPlanets`. Он содержит два свойства: `PlanetName` и `PlanetPosition`. Свойство `PlanetName` — массив, возвращающий название планеты по заданному значению целого индекса. Массив `PlanetPosition` использует не целый, а строковый индекс. Если строка — название существующей планеты, то массив `PlanetPosition` возвратит положение планеты в Солнечной системе.

Например, выполнение следующего оператора с использованием свойства `TddgPlanets.PlanetName` приведет к отображению строки “Нептун”:

```
ShowMessage(ddgPlanets.PlanetName[8]);
```

Сравните это выражение с обычным оператором, генерирующим сообщение “Нептун — 8-я планета солнечной системы”:

```
ShowMessage('Нептун - '+ IntToStr(ddgPlanets.PlanetPosition['Нептун'])
  +'планета солнечной системы ');
```

Прежде чем представить исходный код этого компонента, приведем основные характеристики свойств-массивов, отличающие их от других, уже рассмотренных свойств.

- Свойства-массивы объявляются с использованием одного или нескольких индексных параметров. Тип индексов должен быть простым (это может быть целое число или строка, но не запись и не класс).
- Директивы доступа свойства `read` и `write` должны быть методами. Они не могут быть внутренними полями компонента.
- Если в определении свойства-массива используется несколько индексов (т.е. свойство представлено многомерным массивом), методы доступа должны содержать параметры для каждого индекса в том же порядке, что и в свойстве.

Программный код класса `TddgPlanets` приведен в листинге 21.6.

Листинг 21.6. Класс `TddgPlanets`, иллюстрирующий создание свойств-массивов

```
unit planets;

interface

uses
  Classes, SysUtils;

type

  TddgPlanets = class(TComponent)
  private
    // Методы доступа к свойству-массиву
    function GetPlanetName(const AIndex: Integer): String;
    function GetPlanetPosition(const APlanetName: String): Integer;
  public
```

```
    { Массив индексирован целым значением. Этот вариант используется
      для индексации массива по умолчанию. }
    property PlanetName[const AIndex: Integer]: String
        read GetPlanetName; default;
    { Массив индексируется строковым значением }
    property PlanetPosition[const APlantetName: String]: Integer
        read GetPlanetPosition;
  end;

implementation

const
  // Объявление массива-константы с названиями планет
  PlanetNames: array[1..9] of String[8] =
   ('Меркурий', 'Венера', 'Земля', 'Марс', 'Юпитер', 'Сатурн',
    'Уран', 'Нептун', 'Плутон');

function TddgPlanets.GetPlanetName(const AIndex: Integer): String;
begin
  { Возвращает название планеты, указанной индексом. Если индекс выходит
    за пределы допустимого диапазона, генерируется исключение. }
  if (AIndex < 0) or (AIndex > 9) then
    raise Exception.Create('Неверный номер планеты, введите число от 1 до 9')
  else
    Result := PlanetNames[AIndex];
end;

function TddgPlanets.GetPlanetPosition(const APlanetName: String): Integer;
var
  i: integer;
begin
  Result := 0;
  i := 0;
  { Сравнивает PName с названием каждой планеты и возвращает индекс подходящего
    значения массива-константы, при несовпадении возвращает нуль. }
  repeat
    inc(i);
  until (i = 10) or (CompareStr(UpperCase(APlanetName),
        UpperCase(PlanetNames[i])) = 0);

  if i <> 10 then // Имя планеты найдено
    Result := i;
end;

end.
```

Этот компонент может подсказать идею создания свойств-массивов с целой или строковой переменной, используемой в качестве индекса. Обратите внимание на то, что при чтении свойства возвращается значение, возвращаемое функцией, а не значение внутреннего поля, как в случае с другими свойствами. Лучше понять схему построения компонента вам помогут комментарии в коде.

Значения по умолчанию

Свойству можно назначить значение по умолчанию (стандартное значение), выполнив соответствующее присваивание в конструкторе компонента. Таким образом, если вы добавите следующие операторы в конструктор компонента TddgWorthless, его свойство FIntegerProp при помещении компонента в форму всегда по умолчанию будет равным 100:

```
FIntegerProp := 100;
```

Теперь самое время рассмотреть директивы Default и NoDefault, присутствующие в объявлениях свойств. Если вы обратитесь к исходным текстам библиотеки VCL Delphi, то заметите, что некоторые объявления свойств содержат директиву Default, как и в случае свойства TComponent.Ftag:

```
property Tag: Longint read FTag write FTag default 0;
```

Не путайте действие этого оператора с присвоением значения по умолчанию в конструкторе. Например, измените следующим образом объявление свойства IntegerProp компонента TddgWorthless:

```
property IntegerProp: Integer read FIntegerProp write FIntegerProp default 100;
```

Этот оператор не присваивает свойству значение 100. Здесь просто определяется, будет или нет сохранено значения свойства при сохранении формы, содержащей компонент TddgWorthless. Если значение свойства IntegerProp не равно 100, оно будет сохранено в .DFM-файле. В противном случае оно не будет сохранено, так как 100 — значение, которое присваивается данному свойству заново создаваемого объекта перед считыванием его свойств из потока данных. Для ускорения загрузки форм рекомендуется везде, где только это возможно, использовать директиву Default. Важно понимать, что директива Default не присваивает значение свойству, — это необходимо делать в конструкторе компонента, как было показано выше.

Директива NoDefault используется для переобъявления свойства (со значением по умолчанию) таким образом, что оно независимо от исходного значения всегда записывается в поток данных. Например, можно переобъявить компонент так, чтобы для свойства Tag вообще не задавалось значение по умолчанию:

```
TSample = class(TComponent)
published
  property Tag NoDefault;
```

Вам не следует без особой необходимости применять директиву NoDefault. Например, свойство TForm.PixelsPerInch всегда должно сохраняться для правильного отображения формы при выполнении программы. Необходимо также отметить, что для свойств строкового, вещественного (с плавающей точкой) и типа int64 значения по умолчанию не могут быть объявлены.

Для изменения значения свойства по умолчанию достаточно переопределить свойство новым значением (но без методов чтения и записи).

Свойство по умолчанию типа массив

Вы можете объявить свойство-массив таким образом, что оно будет стандартным свойством того компонента, которому оно принадлежит. Это разрешает при использовании компонента так обращаться с экземпляром объекта, как если бы тот был переменной типа массив.

Например, в компоненте TddgPlanets объявлено свойство TddgPlanets.PlanetName с ключевым словом Default. Благодаря этому пользователь компонента может не применять имя свойства PlanetName для получения соответствующего значения. Он может просто поместить индекс возле идентификатора объекта. Две следующие строки кода приводят к одному и тому же результату:

```
ShowMessage(ddgPlanets.PlanetName[8]);
ShowMessage(ddgPlanets[8]);
```

У объекта может быть только одно стандартное свойство-массив, и потомки этого объекта не могут его переопределять.

Создание событий

В главе 20, "Ключевые элементы VCL и информация о типах времени выполнения", события были описаны как специальные свойства, связанные с кодом, выполняющимся при совершении определенных действий. В данном разделе события рассматриваются более детально. Вы узнаете, каким образом генерируются события и как можно определить собственные свойства-события в вашем компоненте.

Происхождение событий

В самом общем виде *событие* можно определить как любое происшествие, вызванное вмешательством пользователя, операционной системы или логикой программы. Событие связано с некоторым программным кодом, отвечающим на это происшествие. Совокупность события и кода, выполняющегося в ответ на это событие, называется *свойством-событием* и реализуется в виде указателя на некоторый метод. Метод, на который указывает это свойство-событие, называется *обработчиком события.*

Например, когда пользователь щелкает кнопкой мыши, в систему Win32 посылается сообщение WM_MOUSEDOWN. Система Win32 передает это сообщение элементу управления, для которого оно предназначено, и на которое он должен тем или иным образом ответить. Элемент управления может ответить на это событие, сначала проверив наличие кода, предусмотренного для выполнения. Для этого он проверяет, ссылается ли свойство-событие на какой-либо код. Если да, то элемент выполняет этот код, называемый обработчиком события.

Событие OnClick — лишь одно из стандартных свойств-событий, определенных в Delphi. Оно, как и другие свойства-события, имеет соответствующий *метод диспетчеризации событий* (event-dispatching method). Эти методы обычно объявляются как защищенные методы того компонента, которому они принадлежат (в разделе protected). Они выполняют операции по определению того, ссылается ли данное свойство-событие на какой-либо код, предоставленный пользователем компонента. Для свойства OnClick таким методом является метод Click(). Свойство OnClick и метод Click() определены в классе TControl следующим образом:

```
TControl = class(TComponent)
private
  FOnClick: TNotifyEvent;
protected
  procedure Click; dynamic;
  property OnClick: TNotifyEvent read FOnClick write FOnClick;
end;
```

А вот как выглядит реализация метода TControl.Click():

```
procedure TControl.Click;
begin
  if Assigned(FOnClick) then FOnClick(Self);
end;
```

Основное, что от вас требуется — это четко понимать, что свойства-события являются не более чем указателями на методы. Заметьте, что свойство FOnClick имеет тип TNotifyEvent, который определяется следующим образом:

```
TNotifyEvent = procedure(Sender: TObject) of object;
```

Как видите, тип TNotifyEvent представляет собой процедуру с одним параметром Sender типа Tobject, а директива of object делает эту процедуру методом. Это означает, что в процедуру передается дополнительный *неявный* параметр, который не указывается в списке параметров, передаваемых данной процедуре. Это параметр Self, ссылающийся на объект, которому принадлежит данный метод. Когда вызывается метод Click() компонента, он проверяет, действительно ли переменная FOnClick ссылается на метод, и, если да, вызывает этот метод.

Разработчик компонента пишет весь программный текст, определяющий событие, свойство-событие и метод диспетчеризации. Пользователь компонента создает лишь обработчик события. В функции метода диспетчеризации события входит проверка того, назначил ли пользователь событию какой-либо код, и выполнение этого кода, если таковой существует.

В главе 20, “Ключевые элементы VCL и информация о типах времени выполнения”, было описано, как обработчики событий присваиваются свойствам-событиям во время разработки и выполнения программы. В следующем разделе мы покажем, как определять собственные события, свойства-события и методы их диспетчеризации.

Определение свойств-событий

Прежде чем определять свойство-событие, нужно решить, необходим ли новый тип события. Чтобы принять такое решение, полезно познакомиться с типичными свойствами-событиями компонентов библиотеки VCL Delphi. В большинстве случаев можно определить компонент как производный от уже существующего компонента и просто использовать его свойства-события или сделать доступным одно из его защищенных свойств-событий. Но если обнаружится, что ни одно из существующих событий вам не подходит, то тогда определяйте ваше собственное.

В качестве примера рассмотрим следующий сценарий. Предположим, требуется, чтобы в новом компоненте имелось событие, происходящее каждые полминуты по системным часам. Конечно, можно воспользоваться компонентом TTimer для проверки системного времени и выполнения требуемых действий в установленные моменты времени. Однако можно встроить этот код непосредственно в ваш собственный компонент, а затем сделать его доступным пользователям, которым осталось бы лишь подготовить собственный текст обработки события OnHalfMinute.

Код компонента TddgHalfMinute приведен в листинге 21.7 и призван проиллюстрировать способы определения собственных событий.

Листинг 21.7. Создание события TddgHalfMiute

```
unit halfmin;

interface

uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls;

type
  { Определяем процедуру для обработчика события. Свойство-событие будет
    иметь этот процедурный тип, который предусматривает передачу двух
    параметров - генерирующего событие объекта и значения TDataTime,
    фиксирующего время генерации события. В случае нашего компонента это
    событие будет происходить каждые полминуты. }
  TTimeEvent = procedure(Sender: TObject; TheTime: TDateTime) of object;

  TddgHalfMinute = class(TComponent)
  private
    FTimer: TTimer;
    { Определяем поле, которое будет хранить ссылку на обработчик события
      пользователя. Предоставленный пользователем обработчик события
      должен иметь процедурный тип TTimeEvent. }
    FOnHalfMinute: TTimeEvent;
    FOldSecond, FSecond: Word; // Переменные, используемые в программе
    { Определяем процедуру FTimerTimer, которая будет связана со
      свойством-событием FTimer.OnClick. Эта процедура должна иметь
      тип TNotifyEvent, как и само свойство-событие TTimer.OnClick. }
    procedure FTimerTimer(Sender: TObject);
  protected
    { Определение метода диспетчеризации события OnHalfMinute. }
    procedure DoHalfMinute(TheTime: TDateTime); dynamic;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    { Определяем реальное свойство, отображаемое в окне инспектора объектов. }
    property OnHalfMinute: TTimeEvent read FOnHalfMinute write FOnHalfMinute;
  end;

implementation

constructor TddgHalfMinute.Create(AOwner: TComponent);
{ Конструктор Create создает экземпляр компонента TTimer с именем Ftimer,
  а затем задает различные свойства экземпляра FTimer, включая его
  обработчик события OnTimer, являющийся методом FTimerTimer() компонента
  TddgHalfMinute. Заметьте, что свойство FTimer.Enabled устанавливается
  равным значению true при выполнении программы и значению false,
  если компонент находится в режиме разработки. }
begin
  inherited Create(AOwner);
  { Если компонент находится в режиме разработки, FTimer не активизируется.}
  if not (csDesigning in ComponentState) then
  begin
    FTimer := TTimer.Create(self);
    FTimer.Enabled := True;
    { Установка остальных свойств, включая обработчик события FTimer.OnTimer.}
```

```
    FTimer.Interval := 500;
    FTimer.OnTimer := FTimerTimer;
   end;
end;

destructor TddgHalfMinute.Destroy;
begin
  FTimer.Free;
  inherited Destroy;
end;

procedure TddgHalfMinute.FTimerTimer(Sender: TObject);
{ Этот метод служит обработчиком события FTimer.OnTimer. Он назначается
  свойству FTimer.OnTimer динамически в конструкторе TddgHalfMinute.
  Этот метод считывает системное время и проверяет, не кратно ли оно
  30 секундам. Если это условие выполняется, он вызывает метод
  диспетчеризации DoHalfMinute события OnHalfMinute. }
var
  DT: TDateTime;
  Temp: Word;
begin
  DT := Now; // Получаем системное время
  FOldSecond := FSecond; // Сохраняем старое значение секунд
  { Получаем значение секундной составляющей системного времени.}
  DecodeTime(DT, Temp, Temp, FSecond, Temp);

  { Если сейчас не то же самое время, когда метод вызывался в последний
    раз, и оно кратно 30 секундам, вызывается метод DoOnHalfMinute. }
  if FSecond <> FOldSecond then
    if ((FSecond = 30) or (FSecond = 0)) then
      DoHalfMinute(DT)
end;

procedure TddgHalfMinute.DoHalfMinute(TheTime: TDateTime);
{ Это метод диспетчеризации события OnHalfMinute. Он проверяет,
  есть ли у события обработчик, и если да, вызывает его код. }
begin
  if Assigned(FOnHalfMinute) then
    FOnHalfMinute(Self, TheTime);
end;

end.
```

При создании собственного события вы должны решить, какую информацию следует передавать пользователям данного компонента в качестве параметра обработчика события. Например, при создании обработчика события `TEdit.OnKeyPress` его текст может выглядеть следующим образом:

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
end;
```

В этом случае передается не только ссылка на объект, ответственный за возникновение события, но и параметр Char, определяющий нажатую клавишу. В недрах подпрограмм библиотеки VCL это событие происходит в результате поступления сообщения Win32 WM_CHAR, содержащего дополнительную информацию о нажатой клавише. Delphi извлекает из этого сообщения необходимые данные и делает их доступными пользователям компонента в качестве параметров обработчика события. Эта схема взаимодействия позволяет разработчику компонента преобразовывать слишком сложную информацию и делать ее доступной пользователям компонента в более простом и понятном представлении.

Обратите внимание на передаваемый по ссылке параметр в методе Edit1KeyPress(). Может вызвать интерес тот факт, что этот метод был объявлен процедурой, а не функцией, возвращающей значение Char. Хотя методы могут быть функциями, не следует объявлять события как функции, так как это внесет дополнительную путаницу: когда вы обращаетесь к указателю метода-функции, вы не знаете, ссылается ли он на результат функции или является указателем функции. Между прочим, одна функция-событие все же уцелела со времени разработки первых версий Delphi — это событие TApplication.OnHelp.

Просмотрев листинг 21.7, можно заметить, что процедурный тип TOnHalfMinute определен следующим образом:

```
TTimeEvent = procedure(Sender: TObject; TheTime: TDateTime) of object;
```

Этот процедурный тип определяет тип обработчика события OnHalfMinute. В нем мы решили передать пользователю ссылку на объект, вызывающий событие, и значение времени TDataTime, в которое это событие произошло.

Поле хранения FOnHalfMinute ссылается на пользовательский обработчик события и отображается в окне инспектора объектов на этапе проектирования как свойство OnHalfMinute.

Базовая функциональность компонента использует объект TTimer для определения секундной составляющей времени каждые полсекунды. Если это значение равно 0 или 30, вызывается метод DoHalfMinute(), отвечающий за проверку существования обработчика события и последующий его вызов. Прочтите комментарии, приведенные в тексте программы, — они содержат много полезной информации.

После установки компонента в палитре компонентов Delphi вы можете поместить его в форму и добавить следующий обработчик для события OnHalfMinute:

```
procedure TForm1.ddgHalfMinuteHalfMinute(Sender: TObject; TheTime: TDateTime);
begin
  ShowMessage('Время '+TimeToStr(TheTime));
end;
```

Этот код демонстрирует, как новый тип события получает свой обработчик.

Создание методов

Добавление в компонент методов не отличается от их добавления в любой другой объект. Тем не менее существует несколько моментов, которые следует учитывать при разработке компонентов.

Никакой взаимозависимости!

Одна из главных целей, преследуемых при создании компонентов, — упростить его использование конечным пользователем. Поэтому нужно исключить в максимально возможной степени взаимозависимость методов. Например, не нужно заставлять пользователя вызывать

определенный метод при использовании компонента или вызывать методы в определенном порядке. Кроме того, методы, вызываемые пользователем, не должны переводить компонент в такое состояние, при котором другие методы и события не действуют. Наконец, методам следует давать осмысленные имена, чтобы пользователю не приходилось гадать, что делает тот или иной метод.

Открытость метода

При разработке компонента необходимо решить, какие методы следует объявить закрытыми (`private`), открытыми (`public`) или защищенными (`protected`). Следует принять во внимание нужды не только пользователей компонента, но и тех, кто будет использовать ваш компонент в качестве предка для другого пользовательского компонента. Информация в табл. 21.2 поможет вам принять верное решение.

Таблица 21.2. Выбор: Private, protected, public или published?

Директива	Для чего предназначается
`Private`	Переменные и методы экземпляра, которые нежелательно предоставлять типу потомка для доступа или модификации. Для того чтобы помочь пользователю избежать неприятных ситуаций, доступ к таким переменным обычно предоставляется посредством свойств с директивами `read` и `write`, определяющими методы доступа. Естественно, не следует предоставлять пользователям доступ к любым методам реализации свойств
`Protected`	Переменные, методы и свойства экземпляров, к которым классы-потомки (но не пользователи вашего класса) смогут иметь доступ и изменять их. Часто в базовом классе свойства помещаются в раздел `protected`, для того чтобы при необходимости в классе-потомке они могли быть публикуемыми
`Public`	Методы и свойства, доступные любому пользователю вашего класса. Если к некоторым свойствам необходимо иметь доступ во время выполнения программы, но не во время разработки, поместите их в раздел `Public`
`Published`	Свойства, к которым требуется иметь доступ в окне инспектора объектов во время разработки. Для всех свойств, объявленных в этом разделе, генерируется информация о типах времени выполнения (RTTI)

Конструкторы и деструкторы

Создавая новый компонент, вы имеете возможность переопределить конструктор компонента-предка и определить собственный. При этом необходимо соблюдать некоторые меры предосторожности.

Переопределение конструкторов

Объявляя конструктор при построении потомка класса `TComponent`, всегда включайте директиву `override`, как показано ниже.

```
TSomeComopnent = class(TComponent)
private
  { Закрытые объявления }
protected
```

```
  { Защищенные объявления }
public
  constructor Create(AOwner: TComponent); override;
published
  { Публикуемые объявления }
end;
```

На заметку

Конструктор `Create()` на уровне класса `TComponent` является виртуальным. Некомпонентные классы имеют статические конструкторы, которые вызываются конструкторами классов `TComponent`. Следовательно, если вы создаете некомпонентный класс-потомок, конструктор не может быть переопределен, так как он не является виртуальным:

```
TMyObject = class(TPersistent)
```

В этом случае достаточно просто переобъявить конструктор.

Несмотря на то что с точки зрения синтаксиса можно не добавлять директиву `override`, ее отсутствие может вызвать проблемы при использовании вашего компонента. Дело в том, что в процессе использования компонента (как во время разработки, так и во время выполнения) невиртуальный конструктор не будет вызван кодом, создающим этот компонент посредством ссылки на класс (например, в потокообразующей системе).

Не забудьте также убедиться, что унаследованный конструктор вызывается в коде вашего конструктора:

```
constructor TSomeComponent.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  // Поместите здесь ваш код
end;
```

Поведение компонента во время разработки

Напомню, что при создании компонента всегда вызывается его конструктор. Это относится и к созданию компонента во время разработки (при помещении его в форму). Кстати, можно потребовать отмены выполнения некоторых действий, когда компонент находится в состоянии разработки. Например, в конструкторе компонента `TddgHalfMinute` создается компонент `TTimer`. В общем-то ничего страшного в этом нет, но вы можете добиться, чтобы `TTimer` вызывался лишь во время выполнения.

Для определения текущего состояния компонента следует проверить его свойство `ComponentState`. В табл. 21.3 приведены данные о возможных состояниях компонента, взятые из интерактивной справочной системы Delphi 5.

Таблица 21.3. Возможные значения состояния компонента

Флаг	Состояние компонента
`csAncestor`	Устанавливается, если компонент используется в качестве предка. Это значение устанавливается только при установленном флаге `csDesigning`
`csDesigning`	Режим разработки, т.е. компонент находится в форме в окне конструктора форм
`csDestroying`	Объект уничтожается

Флаг	Состояние компонента
csFixups	Установлен, если данный компонент связан с компонентом другой формы, которая еще не загружена. Флаг сбрасывается, когда все подобные связи разрешены
csLoading	Загрузка из объекта файловой системы
csReading	Считывание значений свойств из потока
csUpdating	Компонент обновляется с целью отображения изменений, выполненных в форме-предке. Устанавливается только при установленном флаге csAncestor
csWriting	Запись свойств компонента в поток

Состояние csDesigning чаще всего используется с целью проверки того, находится ли компонент в режиме разработки. Это можно сделать, с помощью следующих операторов:

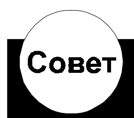
```
inherited Create(AOwner);
if csDesigning in ComponentState then
  { Выполнение необходимых действий }
```

Следует заметить, что состояние csDesigning не определяется до тех пор, пока не будет вызван унаследованный конструктор и данный компонент не будет создан его владельцем. Эти условия почти всегда выполняются при работе с компонентом в конструкторе форм.

Переопределение деструкторов

Переопределяя деструктор, очень важно освободить все распределенные данным компонентом ресурсы *до того*, как будет вызван унаследованный деструктор:

```
destructor TMyComponent.Destroy;
begin
  FTimer.Free;
  MyStrings.Free;
  inherited Destroy;
end;
```



Вот простое, но удобное правило: в переопределенном конструкторе первым вызывается унаследованный конструктор, а в переопределенном деструкторе, наоборот, унаследованный деструктор вызывается последним. Соблюдение этого правила гарантирует, что класс будет корректно настроен перед его модификацией, и все связанные с ним ресурсы будут освобождены перед завершением работы с этим классом.
Из этого правила есть исключения, однако вряд ли вам придется столкнуться с ними на практике.

Регистрация компонента

В процессе регистрации компонент помещается в палитру компонентов Delphi. Если для разработки компонента использовалось окно New Component, для регистрации вам ничего предпринимать не потребуется — Delphi сама сгенерирует необходимый код. Если же вы создаете компонент вручную, в модуль созданного компонента следует включить процедуру Register().

В этом случае от вас требуется лишь добавить процедуру Register() в раздел interface модуля вашего компонента.

Процедура Register просто вызывает процедуру RegisterComponents() для каждого регистрируемого компонента. Процедуре RegisterComponents() передается два параметра: имя вкладки, в которой будет размещаться компонент, и массив типов компонентов. В листинге 21.8 показано, как использовать эту процедуру.

Листинг 21.8. Регистрация компонентов

```
Unit MyComp;
interface
type
  TMyComp = class(TComponent)
  ...
  end;
  TOtherComp = class(TComponent)
  ...
  end;
procedure Register;
implementation
{ методы класса TMyComp }
{ методы класса TOtherComp }
procedure Register;
begin
  RegisterComponents('DDG', [TMyComp, TOtherComp]);
end;
end.
```

При выполнении этого кода компоненты TMyComp и TOtherComp регистрируются и помещаются во вкладку DDG палитры компонентов Delphi.

Палитра компонентов

В Delphi версий 1 и 2 вся библиотека с компонентами, пиктограммами и редакторами, использовавшимися во время разработки, находилась в одном файле. Хотя иметь дело лишь с одним файлом было очень удобно, тем не менее по мере того, как в библиотеку добавлялись все новые и новые компоненты, управлять ею становилось все тяжелее и все больше времени уходило на ее перекомпоновку при каждом добавлении.

С появлением в Delphi 3 пакетов стало возможным разбить библиотеку компонентов на несколько отдельных частей. Хотя иметь дело со множеством файлов сложнее, чем с одним, это решение существенно расширяет возможности управления конфигурацией и значительно сокращает время, необходимое для перекомпоновки библиотеки при добавлении в нее нового компонента.

По умолчанию новые компоненты добавляются в пакет DCLUSR50, но можно создавать и устанавливать новые пакеты времени разработки с помощью команды File⇨New⇨Package. На прилагаемом компакт-диске находится пакет разработки DdgDsgn50.dpk, включающий компоненты, описываемые в этой книге. Пакет времени выполнения называется DdgStd50.dpk.

Если ваша поддержка режима разработки включает нечто большее, чем просто вызов RegisterComponents() (например, редакторы свойств или редакторы компонентов, или регистрацию экспертов), следует перенести процедуру Register() и регистрируемые ею элементы в модуль, отдельный от самих компонентов. Если этого не сделать, то такой “всеобщий” модуль, скомпилированный в пакет времени выполнения и содержащий процедуру Register, ссылающуюся на классы или процедуры, существующие лишь во время разработки, окажется непригодным к использованию. Именно поэтому программная поддержка времени разработки должна быть отделена от соответствующей поддержки времени выполнения.

Тестирование компонента

Несмотря на весь оптимизм и радость, вызванную завершением создания компонента, не спешите добавлять “новоиспеченный” компонент в палитру компонентов, пока он не будет тщательно отлажен. Следует обязательно провести предварительное тестирование, построив проект, в котором создается и используется динамический экземпляр нового компонента. Дело в том, что во время разработки компонент находится в среде IDE. Если в нем содержится ошибка, приводящая к порче памяти, это может, ко всему прочему, привести к зависанию самой среды разработки Delphi. В листинге 21.9 содержится модуль, предназначенный для тестирования компонента TddgExtendedMemo, создание которого описано ниже в этой главе. Этот проект находится на прилагаемом компакт-диске под именем TestEMem.dpr.

Листинг 21.9. Тестирование компонента TddgExtendedMemo

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, exmemo, ExtCtrls;

type

  TMainForm = class(TForm)
    btnCreateMemo: TButton;
    btnGetRowCol: TButton;
    btnSetRowCol: TButton;
    edtColumn: TEdit;
    edtRow: TEdit;
    Panel1: TPanel;
    procedure btnCreateMemoClick(Sender: TObject);
    procedure btnGetRowColClick(Sender: TObject);
    procedure btnSetRowColClick(Sender: TObject);
  public
    EMemo: TddgExtendedMemo;  // Объявление компонента
    procedure OnScroll(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnCreateMemoClick(Sender: TObject);
begin
  { Динамическое создание компонента. Удостоверьтесь, что выполнены
    все надлежащие присвоения свойств, необходимые для правильной работы
```

```
    компонента. Эти присвоения зависят от тестируемого компонента. }
  if not Assigned(EMemo) then
  begin
    EMemo := TddgExtendedMemo.Create(self);
    EMemo.Parent := Panel1;
    EMemo.ScrollBars := ssBoth;
    EMemo.WordWrap := True;
    EMemo.Align := alClient;
    { Назначение обработчиков неоттестированным событиям. }
    EMemo.OnVScroll := OnScroll;
    EMemo.OnHScroll := OnScroll;
  end;
end;

{ Напишите все методы, необходимые для тестирования поведения компонента
  во время выполнения, включая методы доступа ко всем новым свойствам
  и методам компонента.

  Также создайте обработчики событий, определенных пользователем, с тем,
  чтобы их можно было протестировать. Поскольку компонент  создается
  во время выполнения, следует вручную присвоить обработчики событиям,
  как это сделано в конструкторе Create().}
procedure TMainForm.btnGetRowColClick(Sender: TObject);
begin
  if Assigned(EMemo) then
    ShowMessage(Format('Row: %d  Column: %d', [EMemo.Row, EMemo.Column]));
  EMemo.SetFocus;
end;

procedure TMainForm.btnSetRowColClick(Sender: TObject);
begin
  if Assigned(EMemo) then
  begin
    EMemo.Row := StrToInt(edtRow.Text);
    EMemo.Column := StrToInt(edtColumn.Text);
    EMemo.SetFocus;
  end;
end;

procedure TMainForm.OnScroll(Sender: TObject);
begin
  MessageBeep(0);
end;

end.
```

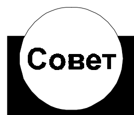
Имейте в виду, что даже тестирование компонента в режиме разработки не позволяет избежать неприятных инцидентов. Некоторые действия программы могут привести к зависанию среды Delphi — например, если вы забудете вызвать унаследованный конструктор `Create`.

На заметку

Не думайте, что во время разработки ваш компонент создается и устанавливается самой средой Delphi — он становится "пригодным к употреблению" только после выполнения конструктора `Create()`. Следовательно, не следует рассматривать метод `Loaded()` как часть процесса создания компонента. Метод `Loaded()` вызывается только при загрузке компонента из потока (например, когда вы помещаете его в форму во время разработки). Метод `Loaded()` отмечает конец процесса обработки потока. Если ваш компонент просто создан, а не загружен из потока, метод `Loaded()` не вызывается.

Создание пиктограммы компонента

Ни один пользовательский компонент не обходится без собственной пиктограммы в палитре компонентов. Для создания такой пиктограммы используйте включенный в Delphi редактор Image Editor или любой другой редактор растровых изображений. Создайте изображение размером 24×24 пикселя и нарисуйте в нем требуемую пиктограмму. Это изображение нужно сохранить в формате файла DCR. Файл с расширением `.dcr` — не что иное, как переименованный `.RES`-файл. Следовательно, если вы сохранили пиктограмму в `.RES`-файле, то вам достаточно просто изменить его расширение на `.dcr`.



Даже если драйвер используемой видеоплаты поддерживает режим 256 цветов и более, сохраните пиктограмму как растровое изображение с 16 цветами, если компонент предполагается передавать другим пользователям. На мониторах с 16 цветами 256-цветные изображения выглядят, как правило, просто ужасно.

Теперь, после помещения пиктограммы в файл DCR, присвойте ей имя класса вашего компонента, но используйте при этом прописные буквы. Сохраните файл ресурсов под тем же именем, что и у модуля компонента, но с расширением `.dcr`. Таким образом, если новый компонент называется `TXYZComponent`, то имя пиктограммы — `TXYZCOMPONENT`. Если имя модуля компонента `XYZCOMP.PAS`, то имя файла ресурсов — `XYZCOMP.DCR`. Поместите этот файл в папку, в которой находится модуль компонента. При перекомпиляции модуля пиктограмма будет автоматически связана с библиотекой компонентов.

Примеры разработки компонентов

В оставшихся разделах этой главы речь пойдет о создании нескольких реальных компонентов. Созданные компоненты будут использованы в двух основных целях. Во-первых, они явятся иллюстрацией использования всех технологий, описанных в предыдущей части главы. А во-вторых, приведенные здесь компоненты вы сможете смело использовать в своих приложениях. При этом ничто не мешает вам расширить их функциональные возможности, для того чтобы они полностью удовлетворяли возникающим потребностям.

Расширение возможностей компонентов-оболочек для классов Win32

В некоторых случаях может потребоваться расширение функциональных возможностей существующих компонентов, в особенности компонентов, инкапсулирующих классы элементов управления Win32. Как этого можно добиться, мы рассмотрим на примере расширения функций элементов управления `TMemo` и `TListBox`.

Компонент TddgExtendedMemo — расширение компонента TMemo

Несмотря на известную гибкость, компонент TMemo не позволяет реализовать некоторые полезные возможности. В частности, он не позволяет управлять положением курсора вставки с точки зрения строк и столбцов. Поэтому расширим компонент TMemo дополнительными открытыми свойствами.

Кроме того, иногда может оказаться удобным выполнять определенные действия всякий раз, когда пользователь проводит некоторую операцию с полосами прокрутки компонента. Для этого будут созданы события, которым можно назначить код, выполняющийся при возникновении событий прокрутки.

Исходный код компонента TddgExtendedMemo приведен в листинге 21.10.

Листинг 21.10. Исходный код компонента TddgExtendedMemo

```
unit ExMemo;

interface

uses
  Windows, Messages, Classes, StdCtrls;

type

  TddgExtendedMemo = class(TMemo)
  private
    FRow: Longint;
    FColumn: Longint;
    FOnHScroll: TNotifyEvent;
    FOnVScroll: TNotifyEvent;
    procedure WMHScroll(var Msg: TWMHScroll); message WM_HSCROLL;
    procedure WMVScroll(var Msg: TWMVScroll); message WM_VSCROLL;
    procedure SetRow(Value: Longint);
    procedure SetColumn(Value: Longint);
    function GetRow: Longint;
    function GetColumn: Longint;
  protected
    // Методы диспетчеризации событий
    procedure HScroll; dynamic;
    procedure VScroll; dynamic;
  public
    property Row: Longint read GetRow write SetRow;
    property Column: Longint read GetColumn write SetColumn;
  published
    property OnHScroll: TNotifyEvent read FOnHScroll write FOnHScroll;
    property OnVScroll: TNotifyEvent read FOnVScroll write FOnVScroll;
  end;

implementation
```

```
procedure TddgExtendedMemo.WMHScroll(var Msg: TWMHScroll);
begin
  inherited;
  HScroll;
end;
procedure TddgExtendedMemo.WMVScroll(var Msg: TWMVScroll);
begin
  inherited;
  VScroll;
end;

procedure TddgExtendedMemo.HScroll;
{ Это метод диспетчеризации события OnHScroll. Он проверяет, связан ли
  обработчик с событием OnHScroll, и если да, вызывает его. }
begin
  if Assigned(FOnHScroll) then
    FOnHScroll(self);
end;

procedure TddgExtendedMemo.VScroll;
{ Это метод диспетчеризации события OnVScroll. Он проверяет, связан ли
  обработчик с событием OnVScroll, и если да, вызывает его. }
begin
  if Assigned(FOnVScroll) then
    FOnVScroll(self);
end;

procedure TddgExtendedMemo.SetRow(Value: Longint);
{ Сообщение EM_LINEINDEX возвращает позицию первого символа строки, заданной
  параметром wParam. В этом экземпляре в качестве wParam используется
  параметр Value. Установка возвращаемого значения SelStart устанавливает
  позицию курсора вставки в строке, заданной параметром Value. }
begin
  SelStart := Perform(EM_LINEINDEX, Value, 0);
  FRow := SelStart;
end;

function TddgExtendedMemo.GetRow: Longint;
{ Сообщение EM_LINEFROMCHAR возвращает строку, в которой находится
  символ, определяемый значением wParam. Если в качестве wParam передается -1,
  то возвращается номер строки, в которой находится курсор вставки. }
begin
  Result := Perform(EM_LINEFROMCHAR, -1, 0);
end;

procedure TddgExtendedMemo.SetColumn(Value: Longint);
begin
  { Получает длину текущей строки с помощью сообщения EM_LINELENGTH.
    Это сообщение принимает позицию символа как параметр WParam.
    Возвращается длина строки, в которой находится этот символ. }
```

```
  FColumn := Perform(EM_LINELENGTH, Perform(EM_LINEINDEX, GetRow, 0), 0);
  { Если FColumn больше значения, переданного в процедуру, это значение
    присваивается переменной FColumn. }
  if FColumn > Value then
    FColumn := Value;
  // SelStart получает значение найденной позиции
  SelStart := Perform(EM_LINEINDEX, GetRow, 0) + FColumn;
end;

function TddgExtendedMemo.GetColumn: Longint;
begin
  { Сообщение EM_LINEINDEX возвращает индекс строки символа, переданного
    в качестве параметра wParam. Если wParam равен -1, то возвращается
    индекс текущей строки. Отнимая это значение от SelStart, получаем
    положение столбца. }
  Result := SelStart - Perform(EM_LINEINDEX, -1, 0);
end;

end.
```

Прежде всего рассмотрим добавленную в компонент TddgExtendedMemo информацию о строках и столбцах. Обратите внимание на то, что в этот компонент было добавлено два закрытых поля: FRow и FColumn. Эти поля предназначены для хранения значений строки и столбца курсора вставки. Заметьте также, что определены открытые свойства Row и Column. Эти свойства объявлены в разделе public, так как во время разработки они не нужны. Они оба обладают методами доступа — как для записи, так и для чтения. В качестве методов доступа к свойствам Row используются методы GetRow() и SetRow(), а к свойствам Column — методы GetColumn() и SetColumn(). На практике можно было бы обойтись без полей FRow и FColumn, так как доступ к свойствам Row и Column осуществляется посредством соответствующих методов доступа; однако мы оставили их, чтобы иметь возможность дальнейшего расширения компонента.

Четыре упомянутых выше метода доступа используют различные сообщения EM_XXXX. Комментарии в коде поясняют, что происходит в каждом методе и как эти сообщения используются для обеспечения компонента информацией о значениях Row и Column.

Компонент TddgExtendedMemo вводит также два события: OnHScroll и OnVScroll. Событие OnHScroll происходит, когда пользователь щелкает на горизонтальной полосе прокрутки, а событие OnVScroll — когда пользователь щелкает на вертикальной полосе прокрутки. Для обнаружения этих событий необходимо перехватить сообщения Win32 WM_HSCROLL и WM_VSCROLL, передаваемые в элемент управления, когда пользователь щелкает на одной из его полос прокрутки. Для этого созданы два обработчика сообщений — WMHScroll() и WMVScroll(), — которые вызывают методы диспетчеризации событий HScroll() и VScroll(), а те, в свою очередь, проверяют, назначил ли пользователь компонента обработчики событиям OnHScroll и OnVScroll, и затем вызывают их. Если вас удивляет, почему эта проверка не выполняется в обработчиках сообщений, то объяснение лежит в обеспечении возможности вызывать обработчик событий в результате выполнения других действий, например изменения пользователем позиции курсора вставки.

Вы можете установить компонент TddgExtendedMemo и использовать его в вашем приложении. Можете также обдумать, как расширить этот компонент (например, когда пользователь изменяет позицию курсора вставки, владельцу элемента посылается сообщение

WM_COMMAND). Функция HiWord(wParam) содержит уведомляющий код, подтверждающий выполнение определенного действия. Этот код мог бы иметь значение EN_CHANGE, наличие которого говорило бы об изменении сообщения, связанного с редактируемым элементом. Затем можно сделать подкласс вашего компонента родительским и перехватывать это сообщение в родительской процедуре окна, которая автоматически обновит поля FRow и FColumn. Подклассы — еще одна сложная тема, обсуждение которой нам еще предстоит.

TddgTabbedListBox — расширение компонента TListBox

Компонент TListBox библиотеки VCL в Object Pascal является оболочкой стандартного элемента управления Win32 API LISTBOX. Совершенству нет предела, поэтому, хотя компонент TListBox инкапсулирует бóльшую часть функциональности элемента Win32, все же попытаемся его расширить. В этом разделе мы проанализируем шаг за шагом процесс создания пользовательского компонента на базе компонента TListBox.

Идея

Идея создания нового компонента, как часто бывает, подсказана практикой. Однажды потребовался список с использованием позиций табуляции и горизонтальной полосы прокрутки для чтения строк, оказавшихся длиннее ширины списка. Обе эти возможности поддерживаются интерфейсом Win32 API, но не реализованы в компоненте TListBox. Назовем новый компонент TddgTabListbox.

План создания этого компонента прост: создается потомок TListBox с корректными свойствами полей, переопределенными методами и новыми методами, предназначенными для получения желаемого поведения компонента.

Код

Первым шагом в создании прокручиваемого списка с позициями табуляции является включение соответствующих оконных стилей в стиль компонента TddgTabListbox при создании окна списка. К необходимым оконным стилям относятся стиль lbs_UseTabStops для табуляции и стиль ws_HScroll для горизонтальной полосы прокрутки. Для добавления оконных стилей к потомку класса TWinControl необходимо переопределить метод CreateParams(), как показано в следующем коде:

```
procedure TddgTabListbox.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params);
  Params.Style := Params.Style or lbs_UseTabStops or ws_HScroll;
end;
```

Метод CreateParams()

Если необходимо изменить один из параметров (таких как стиль или класс окна), передаваемых функции Win32 API CreateWindowEx(), следует воспользоваться методом CreateParams(). Функция CreateWindowEx() используется для создания дескриптора окна, связанного с потомком класса TWinControl. Переопределение метода CreateParams() позволит управлять созданием окна на уровне интерфейса API.

Методу CreateParams передается один параметр типа TCreateParams:

```
TCreateParams = record
    Caption: PChar;
    Style: Longint;
    ExStyle: Longint;
    X, Y: Integer;
    Width, Height: Integer;
    WndParent: HWnd;
    Param: Pointer;
    WindowClass: TWndClass;
    WinClassName: array[0..63] of Char;
  end;
```

Разработчики компонентов осуществляют переопределение метода `CreateParams()` всякий раз, когда возникает необходимость управлять созданием компонента на уровне API. При этом нельзя забывать, что вначале нужно обязательно вызвать унаследованный метод `CreateParams()` для заполнения записи `Params`.

Для установки позиций табуляции компонент `TddgTabListbox` создает сообщение `lb_SetTabStops`, передавая ему количество позиций табуляции и указатель на их массив в параметрах `wParam` и `lParam` соответственно. Эти две переменные будут храниться в классе — в полях `FNumTabStops` и `FTabStops`. Единственная сложность состоит в том, что позиции табуляции в окне списка обрабатываются с использованием единиц измерения диалоговых окон. Delphi не поддерживает никаких других единиц измерения, кроме пикселей, поэтому мы приводим позиции табуляции к пикселям. С помощью модуля `PixDlg.pas`, текст которого приведен в листинге 21.11, можно преобразовывать единицы измерения диалоговых окон для осей X и Y в экранные пиксели и обратно.

Листинг 21.11. Исходный текст модуля `PixDlg.pas`

```
unit Pixdlg;

interface

function DialogUnitsToPixelsX(DlgUnits: word): word;
function DialogUnitsToPixelsY(DlgUnits: word): word;
function PixelsToDialogUnitsX(PixUnits: word): word;
function PixelsToDialogUnitsY(PixUnits: word): word;

implementation
uses WinProcs;

function DialogUnitsToPixelsX(DlgUnits: word): word;
begin
  Result := (DlgUnits * LoWord(GetDialogBaseUnits)) div 4;
end;

function DialogUnitsToPixelsY(DlgUnits: word): word;
begin
  Result := (DlgUnits * HiWord(GetDialogBaseUnits)) div 8;
end;
```

```
function PixelsToDialogUnitsX(PixUnits: word): word;
begin
  Result := PixUnits * 4 div LoWord(GetDialogBaseUnits);
end;

function PixelsToDialogUnitsY(PixUnits: word): word;
begin
  Result := PixUnits * 8 div HiWord(GetDialogBaseUnits);
end;

end.
```

Зная позиции табуляции, можно вычислить протяженность горизонтальной полосы прокрутки. Она должна быть больше самой длинной строки списка. К счастью, в интерфейсе Win32 API есть функция `GetTabbedTextExtent()`, возвращающая как раз эту информацию. Если известна длина самой длинной строки, то можно установить диапазон прокрутки, создав сообщение `lb_SetHorizontalExtent`, передающее требуемый размер в качестве параметра `wParam`.

Кроме того, требуется написать обработчики для некоторых специальных сообщений Win32. В частности, необходимо обеспечить обработку сообщений, управляющих вставкой и удалением, для того чтобы иметь возможность измерить длину любой новой строки или определить, не удалена ли самая длинная строка. В этом смысле для вас будут представлять интерес такие сообщения, как `lb_AddString`, `lb_InsertString` и `lb_DeleteString`. В листинге 21.12 содержится исходный текст модуля `LbTab.pas` компонента `TddgTabListbox`.

Листинг 21.12. Исходный текст модуля `LbTab.pas` компонента `TddgTabListbox`

```
unit Lbtab;

interface

uses
  SysUtils, Windows, Messages, Classes, Controls, StdCtrls;

type

  EddgTabListboxError = class(Exception);

  TddgTabListBox = class(TListBox)
  private
    FLongestString: Word;
    FNumTabStops: Word;
    FTabStops: PWord;
    FSizeAfterDel: Boolean;
    function GetLBStringLength(S: String): word;
    procedure FindLongestString;
    procedure SetScrollLength(S: String);
    procedure LBAddString(var Msg: TMessage); message lb_AddString;
    procedure LBInsertString(var Msg: TMessage); message lb_InsertString;
    procedure LBDeleteString(var Msg: TMessage); message lb_DeleteString;
  protected
```

```
    procedure CreateParams(var Params: TCreateParams); override;
  public
    constructor Create(AOwner: TComponent); override;
    procedure SetTabStops(A: array of word);
  published
    property SizeAfterDel: Boolean read FSizeAfterDel write
        FSizeAfterDel default True;
  end;

implementation

uses PixDlg;

constructor TddgTabListBox.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FSizeAfterDel := True;
  { Установка позиций табуляции по умолчанию. }
  FNumTabStops := 1;
  GetMem(FTabStops, SizeOf(Word) * FNumTabStops);
  FTabStops^ := DialogUnitsToPixelsX(32);
end;

procedure TddgTabListBox.SetTabStops(A: array of word);
{ Эта процедура устанавливает позиции табуляции списка равными значениям,
  заданным в открытом массиве слов A. Новые позиции табуляции указаны
  в пикселях и рассортированы в порядке возрастания. При невозможности
  установить новую позицию табуляции генерируется исключение. }
var
  i: word;
  TempTab: word;
  TempBuf: PWord;
begin
  { Сохраняет новые значения во временных переменных на случай
    возникновения исключения при установке позиций табуляции. }
  TempTab := High(A) + 1;       // Количество позиций табуляции
  GetMem(TempBuf, SizeOf(A));   // Выделение памяти
  Move(A, TempBuf^, SizeOf(A)); // Копирование новых позиций табуляции
  { Перевод пикселей в единицы диалогового окна и ... }
  for i := 0 to TempTab - 1 do
    A[i] := PixelsToDialogUnitsX(A[i]);
  { Передача новых позиций табуляции в список. Обратите внимание:
 следует использовать только единицы диалогового окна. }
  if Perform(lb_SetTabStops, TempTab, Longint(@A)) = 0 then
  begin
  { Если значение выражения равно нулю, новые позиции табуляции невозможно
    установить, временный буфер табуляции очищается и генерируется исключение. }
    FreeMem(TempBuf, SizeOf(Word) * TempTab);
    raise EddgTabListboxError.Create('Невозможно установить новую
        позицию табуляции.')
```

```
  end
  else begin
    { Если выражение не равно нулю, это значит, что установлены новые
      позиции табуляции и можно освободить предыдущие.}
    FreeMem(FTabStops, SizeOf(Word) * FNumTabStops);
    { Копирование значениий из временных переменных... }
    FNumTabStops := TempTab;  // Установка количества позиций табуляции
    FTabStops := TempBuf;     // Установка табуляции из буфера
    FindLongestString;        // Переустановка полосы прокрутки
    Invalidate;               // Перерисовка
  end;
end;

procedure TddgTabListBox.CreateParams(var Params: TCreateParams);
{ Мы должны добавить стили для табуляции и горизонтальной прокрутки.
  Эти стили будут использованы функцией API CreateWindowEx(). }
begin
  inherited CreateParams(Params);
  { Стиль lbs_UseTabStops позволяет использовать в списке табуляцию;
    стиль ws_HScroll позволяет выполнять горизонтальную прокрутку списка. }
  Params.Style := Params.Style or lbs_UseTabStops or ws_HScroll;
end;

function TddgTabListBox.GetLBStringLength(S: String): word;
{ Эта функция возвращает длину строки S в пикселях. }
var
  Size: Integer;
begin
  // Получает длину строки текста
   Canvas.Font := Font;
  Result := LoWord(GetTabbedTextExtent(Canvas.Handle, PChar(S),
         StrLen(PChar(S)), FNumTabStops, FTabStops^));
  { Добавляет немного пространства в конец полосы прокрутки
    с целью улучшения ее внешнего вида.}
  Size := Canvas.TextWidth('X');
  Inc(Result, Size);
end;

procedure TddgTabListBox.SetScrollLength(S: String);
{Эта процедура изменяет длину полосы прокрутки, если строка S
 длиннее предыдущей самой длинной строки. }
var
  Extent: Word;
begin
  Extent := GetLBStringLength(S);
  // Если эта строка оказалась самой длинной...
  if Extent > FLongestString then
  begin
    // Установка самой длинной строки
    FLongestString := Extent;
```

```
    // Установка размера полосы прокрутки
    Perform(lb_SetHorizontalExtent, Extent, 0);
  end;
end;

procedure TddgTabListBox.LBInsertString(var Msg: TMessage);
{ Эта процедура вызывается в ответ на сообщение lb_InsertString.
  Это сообщение посылается списку каждый раз, когда вставляется строка.
  Поле Msg.lParam содержит указатель на вставляемую строку. Если новая
  строка длиннее любой имеющейся, настраивается размер полосы прокрутки. }
begin
  inherited;
  SetScrollLength(PChar(Msg.lParam));
end;

procedure TddgTabListBox.LBAddString(var Msg: TMessage);
{ Эта процедура вызывается в ответ на сообщение lb_AddString.
  Это сообщение передается списку каждый раз, когда добавляется новая строка.
  Поле Msg.lParam содержит указатель на добавляемую строку с завершающим
  нуль-символом. Если новая строка длиннее любой строки списка,
  то настраивается размер полосы прокрутки.}
begin
  inherited;
  SetScrollLength(PChar(Msg.lParam));
end;

procedure TddgTabListBox.FindLongestString;
var
  i: word;
  Strg: String;
begin
  FLongestString := 0;
  { Поиск в цикле самой длинной строки }
  for i := 0 to Items.Count - 1 do
  begin
    Strg := Items[i];
    SetScrollLength(Strg);
  end;
end;

procedure TddgTabListBox.LBDeleteString(var Msg: TMessage);
{ Эта процедура вызывается в ответ на сообщение lb_DeleteString.
  Это сообщение передается в список при каждом удалении строки.
  Msg.wParam содержит индекс удаляемой строки. Присвоением свойству
  SizeAfterDel значения False вы запрещаете обновление полосы прокрутки.
  Это увеличивает производительность при частом удалении строк. }
var
  Str: String;
begin
  if FSizeAfterDel then
```

```
  begin
    Str := Items[Msg.wParam]; // Строка, подлежащая удалению
    inherited;                // Удаление строки
    { Является ли удаленная строка самой длинной? }
    if GetLBStringLength(Str) = FLongestString then
      FindLongestString;
  end
  else
    inherited;
end;

end.
```

В этом компоненте особый интерес представляет метод `SetTabStops()`, использующий в качестве параметра открытый массив типа `word`. Это разрешает пользователям устанавливать такое количество позиций табуляции, какое им требуется, например:

```
ddgTabListboxInstance.SetTabStops([50, 75, 150, 300]);
```

Если текст в списке выходит за видимую часть окна, автоматически появляется горизонтальная полоса прокрутки.

Компонент `TddgRunButton` — создание свойств

Если в среде 16-разрядной Windows, помимо текущей, требовалось запустить еще одну программу, для этого можно было использовать функцию API `WinExec()`. Хотя эта функция в среде Win32 все еще поддерживается, ее не рекомендуется использовать. Для запуска очередного приложения теперь предлагается использовать функции `CreateProcess()` и `ShellExecute()`. Каждый раз применять для этого функцию Win32 API `CreateProcess()` довольно обременительно, поэтому мы создадим для этой цели собственный метод `ProcessExecute()`, о чем и пойдет речь ниже.

Для иллюстрации использования этого метода мы создадим компонент `TddgRunButton`. Его назначение заключается в запуске приложения по щелчку пользователя на предоставленной ему кнопке.

Компонент `TddgRunButton` — идеальный пример создания свойств, проверки их значений и инкапсуляции сложных операций. Вдобавок, мы покажем, как извлечь пиктограмму приложения из его выполняемого файла и как отобразить ее в компоненте `TddgRunButton` во время разработки. Компонент `TddgRunButton` является производным от компонента `TSpeedButton`. Поскольку компонент `TSpeedButton` содержит некоторые свойства, которые нежелательно делать доступными в окне инспектора объектов во время разработки, мы покажем, как можно скрыть уже существующие свойства от пользователя компонента. Конечно, эту методику нельзя считать идеальным подходом. Безусловно, в теоретическом плане правильнее было бы создать свой собственный новый компонент, и авторы вполне с этим согласны. Однако это один из тех случаев, когда компания Borland, при всей своей безграничной мудрости, все же допустила некоторое упущение, не предоставив промежуточного класса между компонентами `TSpeedButton` и `TCustomControl` (последний является предком компонента `TSpeedButton`) компонента, — хотя у всех остальных подобных компонентов они есть. Таким образом, пришлось выбирать: или возиться с собственным компонентом, который в значительной мере дублирует функциональность `TSpeedButton`, или одолжить функциональность `TSpeedButton`,

но скрыть при этом несколько ненужных свойств. Было выбрано последнее, но только по необходимости. Для вас же это — пример того, насколько тщательно следует продумывать возможности и методы расширения создаваемого вами компонента другими разработчиками.

Текст компонента `TddgButton` приведен в листинге 21.13.

Листинг 21.13. Модуль `RunBtn.pas`, содержащий исходный текст компонента `TddgRunButton`

```
{
 Copyright © 1999 by Delphi 5 Developer's Guide - Xavier Pacheco
 and Steve Teixceira
}

unit RunBtn;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons;

type

  TCommandLine = type String;

  TddgRunButton = class(TSpeedButton)
  private
    FCommandLine: TCommandLine;
    // Сокрытие свойств от отображения в инспекторе объектов
    FCaption: TCaption;
    FAllowAllUp: Boolean;
    FFont: TFont;
    FGroupIndex: Integer;
    FLayOut: TButtonLayout;
    procedure SetCommandLine(Value: TCommandLine);
  public
    constructor Create(AOwner: TComponent); override;
    procedure Click; override;
  published
    property CommandLine: TCommandLine read FCommandLine write SetCommandLine;
    // Свойства "только для чтения" скрыты
    property Caption: TCaption read FCaption;
    property AllowAllUp: Boolean read FAllowAllUp;
    property Font: TFont read FFont;
    property GroupIndex: Integer read FGroupIndex;
    property LayOut: TButtonLayOut read FLayOut;
  end;

implementation

uses ShellAPI;
```

```
const
  EXEExtension = '.EXE';

function ProcessExecute(CommandLine: TCommandLine; cShow: Word): Integer;
{ Этот метод инкапсулирует вызов функции CreateProcess(), создающей
  новый процесс и его первичный поток. Этот метод используется в Win32 для
  запуска других приложений и требует применения структур TStartInfo и
  TProcessInformation. Эти структуры не документированы в интерактивной
  справочной системе Delphi 5, но описаны в файле справки
  Win32 как STARTUPINFO и PROCESS_INFORMATION.

  Параметр CommandLine указывает путь к файлу, предназначенному для выполнения.

  Параметр cShow задает константу SW_XXXX, определяющую способ отображения окна.
  Это значение присваивается полю sShowWindow структуры TStartupInfo. }
var
  Rslt: LongBool;
  StartUpInfo: TStartUpInfo; // Информация в разделе STARTUPINFO
  ProcessInfo: TProcessInformation; // Информация в разделе PROCESS_INFORMATION
begin
  { Очищение содержимого структуры StartupInfo }
  FillChar(StartupInfo, SizeOf(TStartupInfo), 0);
  { Инициализация структуры StartupInfo требуемыми данными. Здесь мы
    присваиваем значение константы SW_XXXX полю wShowWindow структуры
    StartupInfo. При задании значения этого поля в поле dwFlags должен
    быть установлен флаг STARTF_USESSHOWWINDOW. Дополнительная информация
    о TStartupInfo находится в интерактивной справочной системе Win32,
    в разделе STARTUPINFO. }
  with StartupInfo do
  begin
    cb := SizeOf(TStartupInfo); // Определяем размер структуры
    dwFlags := STARTF_USESHOWWINDOW or STARTF_FORCEONFEEDBACK;
    wShowWindow := cShow
  end;

  { Создаем процесс путем вызова функции CreateProcess(). Эта функция
    заполняет структуру ProcessInfo информацией о новом процессе и его
    первичном потоке. Подробная информация о структуре TProcessInfo
    содержится в интерактивной справке Win32 в разделе PROCESS_INFORMATION. }
  Rslt := CreateProcess(PChar(CommandLine), nil, nil, nil, False,
    NORMAL_PRIORITY_CLASS, nil, nil, StartupInfo, ProcessInfo);
  { Если Rslt равно true, значит, вызов функции CreateProcess прошел
    успешно. В противном случае GetLastError вернет код ошибки. }
  if Rslt then
    with ProcessInfo do
    begin
      { Ожидание, пока процесс не запустился. }
      WaitForInputIdle(hProcess, INFINITE);
      CloseHandle(hThread); // Освобождение дескриптора hThread
      CloseHandle(hProcess);// Освобождение дескриптора hProcess
      Result := 0;          // Result равен 0, запуск успешный
```

```
    end
  else Result := GetLastError; // присваивание переменной Result кода ошибки
end;

function IsExecutableFile(Value: TCommandLine): Boolean;
{ Этот метод позволяет проверить, представляет ли Value выполняемый
  файл, путем сравнения расширения имени этого файла со строкой 'EXE'}
var
  Ext: String[4];
begin
  Ext := ExtractFileExt(Value);
  Result := (UpperCase(Ext) = EXEExtension);
end;

constructor TddgRunButton.Create(AOwner: TComponent);
{ Конструктор устанавливает стандартные значения свойств размеров - 45x45 }
begin
  inherited Create(AOwner);
  Height := 45;
  Width  := 45;
end;

procedure TddgRunButton.SetCommandLine(Value: TCommandLine);
{ Этот метод записи присваивает полю FCommandLine значение Value, но только
  если это значение Value является корректным именем выполняемого файла.
  Он также устанавливает для компонента TddgRunButton пиктограмму файла,
  заданного параметром Value. }
var
  Icon: TIcon;
begin
  { Вначале проверяется, находится ли Value в заданном месте, и
    действительно ли он представляет собой выполняемый файл. }
  if not IsExecutableFile(Value) then
    Raise Exception.Create(Value+' невыполняемый файл.');
  if not FileExists(Value) then
    Raise Exception.Create('Файл: '+Value+' не найден.');

  FCommandLine := Value;  // Сохранение Value в FCommandLine

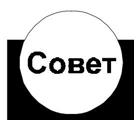
  { Отображение пиктограммы файла, заданного параметром Value, в пиктограмме
    компонента TddgRunButton. Это требует создания экземпляра класса Ticon,
    для помещения пиктограммы. Затем она копируется оттуда в свойство
    Canvas компонента TddgRunButton. Для получения пиктограммы приложения
    следует вызвать функцию Win32 API ExtractIcon(). }
  Icon := TIcon.Create; // Создание экземпляра класса TIcon
  try
    { Выделение пиктограммы из файла приложения. }
    Icon.Handle := ExtractIcon(hInstance, PChar(FCommandLine), 0);
    with Glyph do
    begin
      { Установка свойств TddgRunButton таким образом, чтобы пиктограмму
```

```
      из Icon можно было скопировать в этот компонент. }
    { Сначала очищаем канву на случай, если прежде в ней
      находилась другая пиктограмма. }
    Canvas.Brush.Style := bsSolid;
    Canvas.FillRect(Canvas.ClipRect);
    { Установка ширины и высоты, как в Icon. }
    Width := Icon.Width;
    Height := Icon.Height;
    { Отображение пиктограммы в канве TddgRunButton. }
    Canvas.Draw(0, 0, Icon);
  end;
finally
  Icon.Free; // Освобождение экземпляра TIcon
end;
end;

procedure TddgRunButton.Click;
var
  WERetVal: Word;
begin
  inherited Click; // Вызов унаследованного метода Click
  { Выполнение метода ProcessExecute и проверка возвращаемого им значения.
    Если возвращаемое значение не равно 0, вызывается исключение из-за
    произошедшей ошибки. Исключение показывает код ошибки. }
  WERetVal := ProcessExecute(FCommandLine, sw_ShowNormal);
  if WERetVal <> 0 then begin
    raise Exception.Create('Ошибка при выполнении программы. Код ошибки:; '+
          IntToStr(WERetVal));
  end;
end;

end.
```

У компонента TddgRunButton есть одно свойство CommandLine типа String. Значение этого свойства хранится в закрытом поле FCommandLine .



Определение типа TCommandLine заслуживает отдельного обсуждения. В данном случае используется следующий синтаксис:

```
TCommandLine = type string;
```

Такое определение предлагает компилятору рассматривать TCommandLine как уникальный тип и вместе с тем совместимый с остальными строковыми типами. Новый тип может получить собственную информацию о типах времени выполнения и собственный редактор свойства. Эта же технология может быть использована и для других типов, например:

```
TMySpecialInt = type Integer;
```

Создание редактора для свойства CommandLine описано в следующей главе. Пока не будем отвлекаться на это, так как данная тема сложная и ее придется обсудить особо.

Методом записи для свойства CommandLine является метод SetCommandLine(). Нами написаны две вспомогательные функции: IsExecutableFile() и ProcessExecute().

Функция IsExecutableFile() по расширению переданного ей файла проверяет, является ли он выполняемым.

Создание и выполнение процесса

Функция ProcessExecute() инкапсулирует функцию Win32 API CreateProcess(), запускающую другое приложение. Запускаемое приложение определяется параметром CommandLine, содержащим путь к файлу. Второй параметр содержит одну из констант SW_XXXX, которая задает способ отображения окна. В табл. 21.4 приведены различные константы SW_XXXX с объяснением их значений, взятым из интерактивной справочной системы.

Таблица 21.4. Константы SW_XXXX

Константа SW_XXXX	Значение
SW_HIDE	Скрывает окно. Активным становится другое окно
SW_MAXIMIZE	Отображает окно развернутым до максимума
SW_MINIMIZE	Сворачивает окно
SW_RESTORE	Отображает окно с тем размером, который оно имело перед последним разворачиванием или сворачиванием
SW_SHOW	Отображает окно в его текущем размере и позиции
SW_SHOWDEFAULT	Отображает окно в виде, определенном структурой TStartupInfo, передаваемой функции CreateProcess()
SW_SHOWMAXIMIZED	Активизирует или отображает развернутое окно
SW_SHOWMINIMIZED	Активизирует или отображает свернутое окно
SW_SHOWMINNOACTIVE	Отображает окно свернутым, но активное в данный момент окно остается таковым
SW_SHOWNA	Показывает окно в его текущем состоянии. Активное окно остается таковым
SW_SHOWNOACTIVATE	Отображает окно в таком виде, в каком оно отображалось последний раз. Текущее активное окно остается активным
SW_SHOWNORMAL	Активизирует или отображает окно в таком виде, в каком оно отображалось в последний раз. Позиция окна сохраняется, если оно было предварительно развернуто или свернуто

Функция ProcessExecute() — удобная функция-утилита, которую стоит выделить в отдельный модуль для использования другими приложениями.

Методы класса TddgRunButton

Конструктор TddgRunButton.Create() после вызова унаследованного конструктора просто устанавливает размеры объекта, принимаемые по умолчанию.

Метод SetCommandLine(), представляющий собой метод записи параметра CommandLine, выполняет несколько задач. Во-первых, он проверяет, является ли значение, присваиваемое параметру CommandLine, именем выполняемого файла и, если нет, генерирует исключение.

Проверенное значение присваивается полю FCommandLine. Затем функция SetCommandLine() извлекает пиктограмму из файла приложения и рисует ее на канве объекта TddgRunButton. Для этого используется функция Win32 API ExtractIcon(). Эта технология объяснена в комментариях к коду.

Метод TddgRunButton.Click() — это метод диспетчеризации события TSpeedButton.OnClick. Здесь сначала необходимо вызвать унаследованный метод Click(), вызывающий обработчик события OnClick (если он назначен). После вызова метода Click() вызывается функция ProcessExecute(), после чего проверяется возвращаемое ею значение, чтобы выяснить, был ли этот вызов успешным. Если нет — генерируется исключение.

Компонент-контейнер TddgButtonEdit

Иногда требуется создать компонент, состоящий из одного или нескольких других компонентов. Компонент Delphi TDBNavigator — отличный пример таких компонентов, поскольку он состоит из компонента TPanel и нескольких компонентов TSpeedButton. Данный раздел посвящен созданию компонента, являющегося комбинацией компонентов TEdit и TSpeedButton. Назовем этот компонент TddgButtonEdit.

Проектные решения

Ввиду того, что язык Object Pascal базируется на объектной модели с наследованием от единственного предка, компонент TddgButtonEdit должен быть самостоятельным компонентом, включающим компоненты TEdit и TSpeedButton. Более того, поскольку этот компонент должен включать оконные элементы управления, он и сам должен быть оконным элементом управления. Поэтому TddgButtonEdit необходимо сделать потомком компонента TWinControl. В этом случае экземпляры компонентов TEdit и TSpeedButton следует создать в конструкторе компонента TddgButtonEdit, используя следующий код:

```
constructor TddgButtonEdit.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FEdit        := TEdit.Create(Self);
  FEdit.Parent := self;
  FEdit.Height := 21;

  FSpeedButton := TSpeedButton.Create(Self);
  FSpeedButton.Left := FEdit.Width;
  FSpeedButton.Height := 19;
    // На два пикселя меньше вертикального размера TEdit
  FSpeedButton.Width  := 19;
  FSpeedButton.Caption := '...';
  FSpeedButton.Parent := Self;

  Width  := FEdit.Width+FSpeedButton.Width;
  Height := FEdit.Height;
end;
```

При создании компонента, содержащего другие компоненты, определенную трудность представляет выделение свойств внутренних компонентов из свойств компонента-контейнера. Например, компоненту TddgButtonEdit потребуется свойство Text. Для изменения шрифта этого текста понадобится свойство Font. Наконец, кнопке элемента необходимо событие OnClick. Естественно, не стоит пытаться самостоятельно реализовать эти свойства в компоненте-контейнере, если они уже имеются во внутренних компонентах. Наша задача — вывести необходимые свойства внутренних элементов управления на поверхность контейнера, не переписывая их интерфейса.

Выведение свойств вложенных объектов на поверхность

Эта задача сводится к простой, но трудоемкой процедуре создания методов записи и чтения для каждого свойства внутренних компонентов, выводимого на поверхность компонента-контейнера. Например, компонент TddgButtonEdit можно снабдить свойством Text с соответствующими методами доступа следующим образом:

```
TddgButtonEdit = class(TWinControl)
private
  FEdit: TEdit;
  protected
  procedure SetText(Value: String);
    function  GetText: String;
published
    property Text: String read GetText write SetText;
end;
```

Методы SetText() и GetText() получают прямой доступ к свойству Text вложенного элемента TEdit:

```
function TddgButtonEdit.GetText: String;
begin
  Result := FEdit.Text;
end;

procedure TddgButtonEdit.SetText(Value: String);
begin
  FEdit.Text := Value;
end;
```

Вывод на поверхность событий вложенных объектов

Помимо свойств, на поверхность контейнера требуется вывести и события внутренних компонентов. Например, если пользователь щелкает мышью на элементе TSpeedButton, необходимо обработать его событие OnClick. Вывод на поверхность событий ничем не отличается от вывода свойств; ведь, в конце концов, события — это тоже свойства.

Для начала необходимо снабдить компонент TddgButtonEdit собственным событием OnClick. Для удобства назовем его OnButtonClick. И тогда методы записи и чтения этого события просто перенаправят назначенный обработчик событию OnClick внутреннего компонента TSpeedButton.

В листинге 21.14 приведен исходный текст компонента-контейнера TddgButtonEdit.

Листинг 21.14. Компонент-контейнер TddgButtonEdit

```
unit ButtonEdit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,


 StdCtrls, Buttons;

type
  TddgButtonEdit = class(TWinControl)
  private
    FSpeedButton: TSpeedButton;
    FEdit: TEdit;
  protected
    procedure WMSize(var Message: TWMSize); message WM_SIZE;
    procedure SetText(Value: String);
    function  GetText: String;
    function GetFont: TFont;
    procedure SetFont(Value: TFont);
    function GetOnButtonClick: TNotifyEvent;
    procedure SetOnButtonClick(Value: TNotifyEvent);
  public
    constructor Create(AOwner: TComponent); override;
    destructor  Destroy; override;
  published
    property Text: String read GetText write SetText;
    property Font: TFont read GetFont write SetFont;
    property OnButtonClick: TNotifyEvent read GetOnButtonClick
        write SetOnButtonClick;
  end;

implementation

procedure TddgButtonEdit.WMSize(var Message: TWMSize);
begin
  inherited;
  FEdit.Width := Message.Width-FSpeedButton.Width;
  FSpeedButton.Left := FEdit.Width;
end;

constructor TddgButtonEdit.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FEdit       := TEdit.Create(Self);
  FEdit.Parent := self;
  FEdit.Height := 21;
```

```
  FSpeedButton := TSpeedButton.Create(Self);
  FSpeedButton.Left := FEdit.Width;
  FSpeedButton.Height := 19; // На два пикселя меньше высоты TEdit
  FSpeedButton.Width  := 19;
  FSpeedButton.Caption := '...';
  FSpeedButton.Parent := Self;

  Width  := FEdit.Width+FSpeedButton.Width;
  Height := FEdit.Height;
end;

destructor  TddgButtonEdit.Destroy;
begin
  FSpeedButton.Free;
  FEdit.Free;
  inherited Destroy;
end;

function TddgButtonEdit.GetText: String;
begin
  Result := FEdit.Text;
end;

procedure TddgButtonEdit.SetText(Value: String);
begin
  FEdit.Text := Value;
end;

function TddgButtonEdit.GetFont: TFont;
begin
  Result := FEdit.Font;
end;

procedure TddgButtonEdit.SetFont(Value: TFont);
begin
  if Assigned(FEdit.Font) then
    FEdit.Font.Assign(Value);
end;

function TddgButtonEdit.GetOnButtonClick: TNotifyEvent;
begin
  Result := FSpeedButton.OnClick;
end;

procedure TddgButtonEdit.SetOnButtonClick(Value: TNotifyEvent);
begin
  FSpeedButton.OnClick := Value;
end;

end.
```

Компонент TddgDigitalClock — создание событий компонента

Компонент TddgDigitalClock иллюстрирует процесс создания определенных пользователем событий и предоставления к ним доступа. Здесь используется та же технология, что и при создании событий описанного выше компонента TddgHalfMinute.

Компонент TddgDigitalClock является производным от компонента TPanel. Мы решили, что компонент TPanel идеально подходит для предка создаваемого компонента, так как содержит свойства BevelXXXX, благодаря которым компоненту TddgDigitalClock можно придать приятный внешний вид. К тому же, для отображения системного времени можно использовать свойство TPanel.Caption.

Компонент TddgDigitalClock содержит следующие события, которым пользователь может назначить требуемый код:

OnHour — происходит каждый час;

OnHalfPast — происходит каждые полчаса;

OnMinute — происходит каждую минуту;

OnHalfMinute — происходит каждые полминуты;

OnSecond — происходит каждую секунду.

Компонент TddgDigitalClock использует встроенный в него компонент TTimer. Его обработчик события OnTimer реализует логику отображения информации о времени и вызывает методы диспетчеризации перечисленных выше событий. В листинге 21.15 приведен исходный текст модуля DdgClock.pas.

Листинг 21.15. Модуль DdgClock.pas — исходный текст компонента TddgDigitalClock

```
{
 Copyright © 1999 by Delphi 5 Developer's Guide - Xavier Pacheco
 and Steve Teixceira
}

{$IFDEF VER110}
{$OBJEXPORTALL ON}
{$ENDIF}

unit DDGClock;
interface
uses
  Windows, Messages, Controls, Forms, SysUtils, Classes, ExtCtrls;

type
  { Объявление типа события, которому в качестве параметров будет
    передаваться источник события и переменная TDateTime. }
  TTimeEvent = procedure(Sender: TObject; DDGTime: TDateTime) of object;

  TddgDigitalClock = class(TPanel)
  private
    { Поля данных }
    FHour,
```

```
    FMinute,
    FSecond: Word;
    FDateTime: TDateTime;
    FOldMinute,
    FOldSecond: Word;
    FTimer: TTimer;
    { Обработчики событий }
    FOnHour: TTimeEvent;       // Происходит каждый час
    FOnHalfPast: TTimeEvent;   // Происходит каждые полчаса
    FOnMinute: TTimeEvent;     // Происходит каждую минуту
    FOnSecond: TTimeEvent;     // Происходит каждую секунду
    FOnHalfMinute: TTimeEvent; // Происходит каждые 30 секунд

    { Определение обработчика события OnTimer для внутренней
      переменной FTimer типа Ttimer. }
    procedure TimerProc(Sender: TObject);
  protected
    { Переопределение метода Paint }
    procedure Paint; override;

    { Определение методов диспетчеризации различных событий. }
    procedure DoHour(Tm: TDateTime); dynamic;
    procedure DoHalfPast(Tm: TDateTime); dynamic;
    procedure DoMinute(Tm: TDateTime); dynamic;
    procedure DoHalfMinute(Tm: TDateTime); dynamic;
    procedure DoSecond(Tm: TDateTime); dynamic;
  public
    { Переопределение конструктора Create и деструктора Destroy. }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    { Определение свойств-событий. }
    property OnHour: TTimeEvent read FOnHour write FOnHour;
    property OnHalfPast: TTimeEvent read FOnHalfPast write FOnHalfPast;
    property OnMinute: TTimeEvent read FOnMinute write FOnMinute;
    property OnHalfMinute: TTimeEvent read FOnHalfMinute
             write FOnHalfMinute;
    property OnSecond: TTimeEvent read FOnSecond write FOnSecond;
  end;

implementation

constructor TddgDigitalClock.Create(AOwner: TComponent);
begin
  inherited Create(AOwner); // Вызываем унаследованный конструктор
  Height := 25; // Устанавливаем размеры по умолчанию
  Width := 120;
  BevelInner := bvLowered; // Устанавливаем стандартные свойства
  BevelOuter := bvLowered;
  { Назначаем унаследованному свойству Caption пустую строку. }
```

```
  inherited Caption := '';
  { Создаем экземпляр компонента TTimer и устанавливаем его
    свойство Interval и обработчик события OnTime. }
  FTimer:= TTimer.Create(self);
  FTimer.interval:= 200;
  FTimer.OnTimer:= TimerProc;
end;

destructor TddgDigitalClock.Destroy;
begin
  FTimer.Free;       // Освобождаем экземпляр компонента TTimer
  inherited Destroy; // Вызываем унаследованный метод Destroy
end;

procedure TddgDigitalClock.Paint;
begin
  inherited Paint; // Вызываем унаследованный метод Paint
  { Теперь унаследованному свойству Caption назначается текущее время. }
  inherited Caption := TimeToStr(FDateTime);
end;

procedure TddgDigitalClock.TimerProc(Sender: TObject);
var
  HSec: Word;
begin
  { Сохраняем старые значения минут и секунд для дальнейшего использования. }
  FOldMinute := FMinute;
  FOldSecond := FSecond;
  FDateTime := Now; // Получаем текущее время
  { Извлечение отдельных составляющих времени }
  DecodeTime(FDateTime, FHour, FMinute, FSecond, Hsec);

  refresh; // Перерисовываем компонент для отображения нового времени

  { Вызов обработчиков событий времени }
  if FMinute = 0 then
    DoHour(FDateTime);
  if FMinute = 30 then
    DoHalfPast(FDateTime);
  if (FMinute <> FOldMinute) then
    DoMinute(FDateTime);
  if FSecond <> FOldSecond then
    if ((FSecond = 30) or (FSecond = 0)) then
      DoHalfMinute(FDateTime)
    else
       DoSecond(FDateTime);
end;

{ Приведенные ниже методы диспетчеризации событий проверяют наличие
  соответствующих обработчиков событий отсчета времени и вызывают их. }
```

```
 procedure TddgDigitalClock.DoHour(Tm: TDateTime);
begin
  if Assigned(FOnHour) then
   TTimeEvent(FOnHour)(Self, Tm);
end;

procedure TddgDigitalClock.DoHalfPast(Tm: TDateTime);
begin
  if Assigned(FOnHalfPast) then
   TTimeEvent(FOnHalfPast)(Self, Tm);
end;

procedure TddgDigitalClock.DoMinute(Tm: TDateTime);
begin
  if Assigned(FOnMinute) then
   TTimeEvent(FOnMinute)(Self, Tm);
end;

procedure TddgDigitalClock.DoHalfMinute(Tm: TDateTime);
begin
  if Assigned(FOnHalfMinute) then
   TTimeEvent(FOnHalfMinute)(Self, Tm);
end;

procedure TddgDigitalClock.DoSecond(Tm: TDateTime);
begin
  if Assigned(FOnSecond) then
   TTimeEvent(FOnSecond)(Self, Tm);
end;

end.
```

Логическая структура этого компонента разъясняется в комментариях к исходному коду. Применяемые здесь методы не отличаются от рассмотренных выше при создании событий. В компонент TddgDigitalClock лишь добавлено большее количество событий и включена логика для определения того, какое из событий произошло.

Добавление форм в палитру компонентов

Добавление форм в хранилище объектов (Object Repository) — удобный способ последующей работы с ними. Однако случается, что вы разрабатываете форму, которую собираетесь часто использовать и которая не предполагает наследования и внесения дополнительной функциональности. В этом случае Delphi 5 позволяет использовать такие формы, как компоненты палитры компонентов. В частности, компоненты TFontDialog и TOpenDialog — примеры форм, доступных в палитре компонентов. В действительности эти диалоговые окна не являются формами Delphi — они содержатся в библиотеке CommDlg.dll, но, тем не менее, идея остается той же.

Для добавления формы в палитру компонентов следует создать для нее компонент-оболочку, превращающий ее в независимый устанавливаемый компонент. В качестве примера рассмотрим диалоговое окно, автоматически проверяющее пароль пользователя. Это

очень простой пример, но нашей целью является демонстрация не установки в качестве компонентов сложных диалоговых окон, а общей идеи добавления диалоговых окон в палитру компонентов. Предлагаемую методику можно применять для добавления в палитру компонентов диалоговых окон любой сложности.

Сначала необходимо создать форму, которая впоследствии будет помещена в компонент-оболочку. Используемая здесь форма определена в файле `PwDlg.pas`. В нем же содержится и текст компонента-оболочки.

В листинге 21.16 приведен текст модуля `PwDlg.pas`, содержащего определение формы `TPasswordDlg` и ее компонента-оболочки `TddgPasswordDialog`.

Листинг 21.16. Модуль `PwDlg.pas` — форма `TPasswordDlg` и ее компонент-оболочка `TddgPasswordDialog`

```
unit PwDlg;

interface

uses Windows, SysUtils, Classes, Graphics, Forms, Controls, StdCtrls,
     Buttons;

type

TPasswordDlg = class(TForm)
    Label1: TLabel;
    Password: TEdit;
    OKBtn: TButton;
    CancelBtn: TButton;
  end;

  { Объявление компонента-оболочки. }
  TddgPasswordDialog = class(TComponent)
  private
    PassWordDlg: TPasswordDlg; // Экземпляр компонента TPassWordDlg
    FPassWord: String;         // Поле для хранения пароля
  public
    function Execute: Boolean; //Функция запуска диалогового окна
  published
    property PassWord: String read FPassword write FPassword;
  end;

implementation
{$R *.DFM}

function TddgPasswordDialog.Execute: Boolean;
begin
  { Создаем экземпляр компонента TPasswordDlg }
  PasswordDlg := TPasswordDlg.Create(Application);
  try
    Result := False;  // Инициализация Result значением false
    { Отображение диалогового окна. Если пароль правильный,
```

```
    возвращается значение true,. }
   if PasswordDlg.ShowModal = mrOk then
     Result := PasswordDlg.Password.Text = FPassword;
  finally
    PasswordDlg.Free;  // Освобождение экземпляра компонента PasswordDlg
  end;
end;

end.
```

Компонент TddgPasswordDialog называется *компонентом-оболочкой*, потому что он превращает форму в компонент, который можно установить в палитру компонентов Delphi 5.

Компонент TddgPasswordDialog является производным непосредственно от компонента TComponent. Как упоминалось в предыдущей главе, компонент TComponent — это класс самого низкого уровня, с которым можно работать в конструкторе форм интегрированной среды Delphi. В классе TddgPasswordDialog есть две закрытые переменные: PasswordDlg типа TPasswordDlg и FPassWord типа String. Переменная PasswordDlg — это экземпляр компонента TPasswordDlg, отображаемый оболочечным компонентом. Переменная FPassWord — внутреннее поле, хранящее строку пароля.

Переменная FPassWord получает данные через свойство PassWord. Таким образом, в действительности свойство PassWord не хранит данных, а лишь служит интерфейсом с переменной FPassWord.

Метод TddgPasswordDialog.Execute() создает экземпляр объекта TPasswordDlg и отображает его в виде модального диалогового окна. Когда диалоговое окно закрывается, строка, введенная в элемент TEdit, сравнивается со строкой, хранящейся в переменной FPassword.

Приведенный здесь код помещается в конструкцию try..finally. Часть кода, относящаяся к ключевому слову finally, гарантирует, что компонент TPasswordDlg будет освобожден в любом случае, невзирая на возможные ошибки.

Поместив компонент TddgPasswordDialog в палитру компонентов, можно создать проект, использующий этот компонент. Компонент TddgPasswordDialog можно будет выбирать и помещать в форму так же, как и любой другой компонент. Проект, создание которого описано в предыдущем разделе, содержит компонент TddgPasswordDialog и кнопку, обработчик события OnClick которой приведен ниже:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if ddgPasswordDialog.Execute then  // Запуск PasswordDialog
    ShowMessage('Правильно!')        // Правильный пароль
  else
    ShowMessage('Вы ошиблись!');     // Неправильный пароль
end;
```

В окне инспектора объектов отображаются три свойства компонента TddgPasswordDialog: Name, Password и Tag. Для использования этого компонента нужно присвоить свойству Password определенное значение. Когда вы запускаете проект на выполнение, TddgPasswordDialog получает от пользователя пароль и сравнивает его со значением свойства Password.

Пакеты компонентов

В Delphi 3 появилась новая возможность — *пакеты*, позволяющие располагать части приложения в разных модулях, которые могут совместно использоваться многими приложениями. Пакеты подобны библиотекам динамической компоновки (DLL), но отличаются от них способом использования. Пакеты в основном предназначены для хранения коллекций компонентов в отдельном, разделяемом модуле (Borland Package Library — файл `.bpl`). В Delphi пакеты могут использоваться во время выполнения — это позволяет не связывать их текст в единое целое с текстом приложения еще на этапе компиляции. Поскольку код этих модулей находится в `.bpl`-файлах, а не в файлах `.exe` и `.dll`, размер последних значительно уменьшается.

Пакеты отличаются от библиотек DLL тем, что являются частью библиотеки VCL Delphi. Это означает, что приложения, написанные на других языках, не могут использовать эти пакеты (за исключением C++Builder). Одной из причин введения пакетов являлась необходимость преодолеть ограничения, свойственные Delphi версий 1 и 2. В этих двух версиях Delphi подпрограммы библиотеки VCL увеличивали размер каждого выполняемого файла как минимум на 150–200 Кбайт. Таким образом, если вы помещали даже некоторую часть вашего приложения в библиотеку DLL, то и файл этой библиотеки, и сам выполняемый файл содержали избыточный код. При развертывании на одном компьютере большого программного комплекса, состоящего из нескольких приложений, это выливалось в существенную проблему. Пакеты позволяют уменьшить размеры приложений и предоставляют удобный способ для распространения коллекций компонентов.

Зачем использовать пакеты

Использование пакетов вызвано рядом причин. О трех наиболее важных из них и пойдет речь в следующих разделах.

Сокращение размера кода

Самой главной причиной использования пакетов является стремление уменьшить размер приложений и библиотек DLL. Delphi поставляется в виде нескольких пакетов, среди которых по логическим признакам распределены компоненты библиотеки VCL. Создаваемые приложения можно компилировать с использованием этих уже существующих пакетов Delphi.

Дробление приложений и уменьшение их размеров

Как известно, многие приложения распространяются в Internet в виде полномасштабных версий, загружаемых демо-версии или в виде пакетов обновлений к уже существующим приложениям. Представьте себе преимущества такого режима загрузки: пользователь загружает лишь сокращенную версию приложения, тогда как основная часть этого приложения (установленная во время предыдущей его инсталляции) уже присутствует в его системе.

Разбив приложения на пакеты, можно предоставить пользователям право получать обновления лишь для тех частей приложения, которые им действительно нужны. Необходимо, однако, принять во внимание несколько моментов, касающихся взаимоотношений версий пакетов и приложений. Об этом — чуть позже.

Хранение компонентов

Одной из наиболее распространенных причин использования пакетов является распространение компонентов, созданных сторонними фирмами. Поэтому, если вы занимаетесь поставкой компонентов, то обязаны знать, как создавать пакеты. Кстати, в пакеты можно помещать даже некоторые элементы времени разработки, такие как редакторы свойств и компонентов, мастера и эксперты. Все эти надстройки времени проектирования распространяются в виде пакетов.

Когда не нужно использовать пакеты

Не следует использовать пакеты времени выполнения, если вы не уверены в том, что их будут использовать и другие приложения. Иначе эти пакеты займут больше места на диске, чем в том случае, если вы просто скомпилируете весь исходный код в один выполняемый файл. Почему так получается? При создании среднего приложения с использованием пакетов, выполняемый файл уменьшается приблизительно с 200 Кбайт до 30 Кбайт, и может показаться, что на диске удалось сохранить 170 Кбайт свободного места. Однако при распространении этого приложения к нему присовокупляются используемые им пакеты, в том числе пакет `vcl50.dcp`, имеющий размер, равный почти 2 Мбайт. Как вы понимаете, это совсем не та экономия пространства на диске, на которую вы рассчитывали. По нашему мнению, пакеты стоит применять только в том случае, если ими будут пользоваться многие выполняемые файлы. Помните, что все эти рассуждения относятся лишь к пакетам времени выполнения. Если вы — разработчик компонентов, то должны обеспечить своим пользователям пакет разработки с компонентом, который хотите сделать доступным в интегрированной среде Delphi.

Типы пакетов

Существует четыре типа пакетов, которые можно создать и использовать.

- **Пакеты времени выполнения**. Такие пакеты содержат код, компоненты и другие ресурсы, которые используются приложением во время выполнения. Если приложение создается в расчете на определенный пакет времени выполнения, то в случае отсутствия этого пакета приложение работать не будет.
- **Пакеты разработки**. Эти пакеты содержат компоненты, редакторы свойств или компонентов, эксперты и все прочие элементы, необходимые для разработки приложения в интегрированной среде Delphi. Этот тип пакетов используется только со средой Delphi и никогда не распространяется вместе с создаваемыми приложениями.
- **Пакеты разработки и времени выполнения**. Эти пакеты, объединяющие в себе возможности пакетов и первого и второго типа, обычно используются в случае отсутствия специальных элементов времени разработки наподобие редакторов свойств и компонентов или экспертов. Пакеты этого типа можно создавать для упрощения процесса разработки и развертывания использующих их приложений. Однако, если этот пакет все-таки содержит элементы разработки, их использование в приложениях будет связано с дополнительной нагрузкой, вызванной поддержкой этапа разработки. Рекомендуем пакеты времени выполнения и пакеты разработки создавать отдельно, чтобы отделить специфические элементы времени разработки, если таковые присутствуют.
- **Пакеты, не являющиеся ни пакетами времени выполнения, ни пакетами разработки.** Эта редкая разновидность пакетов служит для использования другими пакетами. Они не предназначены ни для непосредственных ссылок приложения на них, ни для работы в среде разработки. Как видите, пакеты могут использовать или включать другие пакеты.

Файлы пакетов

В табл. 21.5 приводятся и описываются файлы пакетов в зависимости от расширений их имен.

Таблица 21.5. Файлы пакетов

Расширение файла	Тип файла	Описание
.dpk	Исходный файл пакета	Этот файл создается при запуске редактора пакетов. Можно считать его разновидностью файла проекта Delphi .dpr
.dcp	Символьный файл пакета времени выполнения и разработки	Скомпилированная версия пакета с символьной информацией о пакете и его модулях. Кроме того, в нем содержится заголовок с информацией, необходимой среде разработки Delphi
.dcu	Скомпилированный модуль	Скомпилированная версия модуля, содержащегося в пакете. Для каждого модуля, содержащегося в пакете, создается собственный .dcu-файл
.bpl	Пакет библиотеки времени выполнения и разработки	Это пакет времени выполнения или разработки, эквивалент библиотеки DLL Windows. Если это — пакет времени выполнения, необходимо распространять его вместе с теми приложениями, которые его используют. Если это — пакет разработки, он будет распространяться вместе со своим “собратом” времени выполнения среди программистов, использующих его для написания программ. Помните, что, если вы не распространяете исходный код пакета, вам нужно предоставить соответствующий ему .dcp-файл

Активизация использования пакетов в приложении

Выполнить это очень просто — достаточно установить флажок опции Build with Runtime Packages во вкладке Packages диалогового окна Project Options. При компоновке приложения с этой опцией, вместо статической компоновки всех модулей в .exe- или .dll-файл, оно будет динамически связано с пакетами времени выполнения. В результате вы получите более стройное приложение. Однако не забывайте при его распространении передавать пользователям и необходимые пакеты.

Установка пакетов в интегрированную среду разработки Delphi

Это также нетрудно. Вам может понадобиться это сделать, если вы получили новый набор компонентов от сторонних разработчиков. Прежде всего, необходимо поместить файлы пакета в нужное место. Папки, в которых обычно хранятся различные файлы пакетов, приведены в табл. 21.6.

Таблица 21.6. Размещение файлов пакетов

Файл пакета	Местоположение
Пакеты времени выполнения (*.bpl)	Пакеты времени выполнения должны быть помещены в каталог \Windows\System (Windows 95/98) или \WinNT\System32 (Windows NT)
Пакеты разработки (*.bpl)	Поскольку, возможно, у вас может быть сразу несколько пакетов от различных производителей, пакеты разработки должны быть помещены в общую папку — там за ними легче уследить. Например, создайте папку \Delphi 5\Pkg и храните там свои пакеты разработки
Файлы пакетов с символьным кодом (*.dcp)	Их можно поместить туда же, куда и файлы пакетов разработки (*.bpl)
Скомпилированные модули (*.dcu)	Вы должны распространять скомпилированные модули вместе с пакетами времени разработки. Рекомендуем хранить .dcu-файлы сторонних производителей в папке, аналогичной \Delphi 5\Lib; например, это может быть папка \Delphi 5\3PrtyLyb. Ваш путь поиска должен включать эту папку

Для инсталляции пакета нужно перейти во вкладку Packages диалогового окна Project Options, которое раскрывается с помощью команды Component⇨Install Packages.

Щелкните на кнопке Add и выберите нужный файл с расширением .bpl. Его можно найти во вкладке Project. По щелчку на кнопке OK новый пакет будет инсталлирован в среду разработки Delphi. Если в этом пакете содержатся компоненты, то вы обнаружите их в новой вкладке Components палитры компонентов.

Разработка пакетов

Перед созданием нового пакета следует определиться с некоторыми вопросами. Во-первых, необходимо решить, пакет какого типа будет создаваться (времени выполнения, разработки или любого другого). От этого решения зависит выбор одного из сценариев, которые вам будут предложены ниже. Во-вторых, необходимо решить, как назвать новый пакет и где будут храниться файлы проекта. Не следует размещать готовый пакет в той же папке, в которой вы его создаете. И, наконец, следует решить, какие модули будет содержать данный пакет и какие пакеты ему будут необходимы.

Редактор пакетов

Пакеты обычно создаются с помощью редактора пакетов (Package Editor), который вызывается посредством выбора пиктограммы Packages в диалоговом окне New Items (это окно вызывается по команде File⇨New). Окно редактора пакетов содержит две папки — Contains и Requires.

Папка Contains

В папке Contains указываются модули, которые нужно скомпилировать в новый пакет. Существует несколько правил, которые следует соблюдать при помещении модулей в папку Contains пакета.

- Пакет не должен содержаться в разделе contains другого пакета или в разделе uses модуля другого пакета.

- Модули, перечисленные в разделе contains пакета прямо или косвенно (модули в разделе uses модулей раздела contains пакета), не могут быть в указаны в разделе Requires пакета. Эти модули и так будут присутствовать в пакете при его компиляции.
- Нельзя поместить имя модуля в раздел contains пакета, если оно уже находится в разделе contains другого пакета, используемого одним и тем же приложением.

Папка Requires

В этой папке задаются другие пакеты, необходимые новому пакету. Это подобно использованию раздела uses в модулях Delphi. В большинстве случаев создаваемые пакеты будут использовать пакет VCL50, содержащий стандартные компоненты библиотеки VCL Delphi. Поэтому нужно поместить его в раздел requires. Обычно все пользовательские компоненты помещаются в пакет времени выполнения, а затем создается пакет разработки, в раздел requires которого включается этот пакет времени выполнения. Существует несколько правил помещения пакетов в раздел Requires другого пакета.

- Избегайте циклических ссылок: Package1 не может иметь в своем разделе Requires пакет Package1 или другой пакет, содержащий Package1 в своем разделе Requires.
- Цепочка ссылок не должна возвращаться к пакетам, уже указанным в ней.

Редактор пакетов имеет панель инструментов и контекстно-зависимое меню. О назначении каждой из кнопок и команд можно узнать в разделе “Package Editor” интерактивной справочной системы Delphi 5.

Сценарии разработки пакетов

Выше уже отмечалось, что тип разрабатываемого пакета базируется на том или ином сценарии его использования. В этом разделе вам предлагается три возможных сценария использования пакетов времени разработки и/или времени выполнения.

Сценарий 1. Компонентные пакеты разработки и времени выполнения

Сценарий для пакетов разработки и времени выполнения, содержащих компоненты, потребуется разработчикам компонентов для удовлетворения любого из двух приведенных ниже условий.

- Необходимо предоставить программистам Delphi право выбора: компоновать приложение вместе с вашими компонентами или распространять их отдельно от него.
- Не желательно заставлять пользователей компилировать в код приложения элементы времени разработки (редакторы свойств, компонентов и т.п.).

Исходя из этого сценария, следует создать пакет одновременно и разработки, и времени выполнения. На рис. 21.4 изображена схема организации такого пакета. Как видите, пакет разработки DDGDsgn50.dpk включает и элементы времени разработки (редакторы свойств, компонентов и т.п.), и пакет времени выполнения DDGStd50.dpk. Этот вложенный пакет содержит только выполняемый код новых компонентов. Подобное устройство пакета организуется включением имени пакета времени выполнения в раздел requires пакета разработки.

Перед компиляцией этого пакета следует выбрать надлежащие опции для каждого из пакетов. Это выполняется в диалоговом окне Package Options, которое выводится с помощью команды Options контекстного меню редактора пакетов. Для пакета времени

выполнения DDGStd50.dpk переключатель Usage Options должен быть установлен в положение Runtime Only. Это гарантирует, что компонент времени выполнения не будет установлен в интегрированную среду как пакет разработки (см. врезку "Безопасность компонентов" ниже в этой главе). Для пакета разработки DDGDsgn50 этот переключатель должен быть установлен в положение Design Time Only. Это обеспечит установку пакета в среде разработки Delphi и предохранить его от использования в качестве пакета времени выполнения.

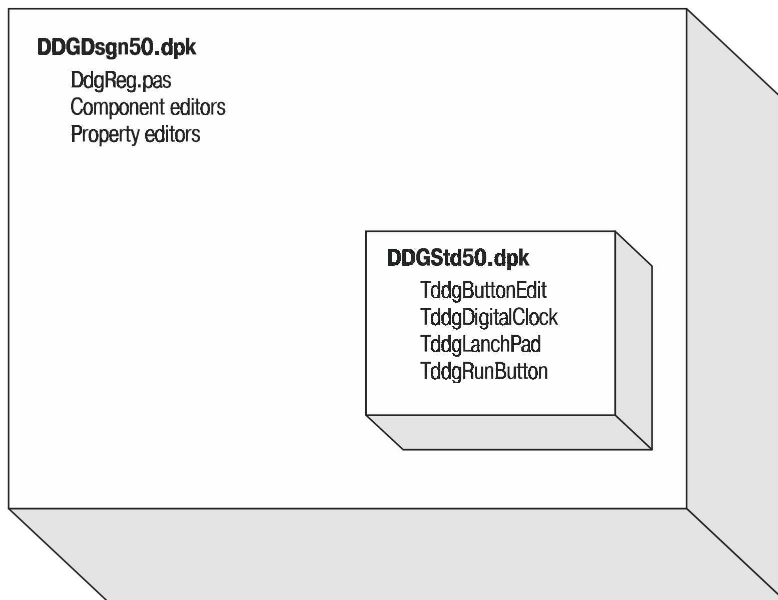


Рис. 21.4. Пакеты разработки содержат элементы разработки и пакеты времени выполнения

Вставка пакета времени выполнения в пакет разработки еще не делает содержащиеся в нем компоненты доступными в IDE Delphi. Компоненты нужно зарегистрировать. Как вы уже знаете, при создании компонента Delphi автоматически вставляет в его модуль процедуру `Register()`, которая, в свою очередь, вызывает процедуру `RegisterComponents()`. Это и есть процедура, регистрирующая любой компонент в среде IDE Delphi при его установке. При работе с пакетами рекомендуем выделить процедуру `Register()` из компонентов в отдельный модуль регистрации. Этот модуль будет регистрировать все компоненты пакета путем вызова процедуры `RegisterComponents()`. Это не только упрощает управление регистрацией компонентов, но и позволяет избежать нелегальной установки и использования вашего пакета в Delphi, так как незарегистрированные компоненты будут не доступны в IDE.

В частности, все использованные в этой книге компоненты мы поместили в пакет времени выполнения DDGStd50.dpk. Редакторы свойств, редакторы компонентов и регистрационный модуль DdgReg.pas этих компонентов находятся в пакете разработки DDGDSgn50.dpk. В разделе `requires` этого пакета указан и пакет DDGStd50.dpk. В листинге 21.17 показано, как может выглядеть регистрационный модуль.

Листинг 21.17. Регистрационный модуль компонентов этой книги

```
unit DDGReg;

interface

procedure Register;

implementation

uses Classes, ExptIntf, DsgnIntf, TrayIcon, AppBars, ABExpt, Worthless,
  RunBtn, PwDlg, Planets, LbTab, HalfMin, DDGClock, ExMemo, MemView,
  Marquee, PlanetPE, RunBtnPE, CompEdit, DefProp, Wavez,
  WavezEd, LnchPad, LPadPE, Cards, ButtonEdit, Planet, DrwPnel;

procedure Register;
begin

  // Регистрация компонентов
  RegisterComponents('DDG',
  [ TddgTrayNotifyIcon, TddgDigitalClock, TddgHalfMinute, TddgButtonEdit,
    TddgExtendedMemo, TddgTabListbox, TddgRunButton, TddgLaunchPad,
    TddgMemView, TddgMarquee, TddgWaveFile, TddgCard, TddgPasswordDialog,
    TddgPlanet, TddgPlanets, TddgWorthLess, TddgDrawPanel,
    TComponenTEditorSample, TDefinePropTest]);

  // Регистрация редакторов свойств
  RegisterPropertyEditor(TypeInfo(TRunButtons), TddgLaunchPad, '',
    TRunButtonsProperty);
  RegisterPropertyEditor(TypeInfo(TWaveFileString), TddgWaveFile, 'WaveName',
    TWaveFileStringProperty);
  RegisterComponenTEditor(TddgWaveFile, TWaveEditor);
  RegisterComponenTEditor(TComponenTEditorSample, TSampleEditor);
  RegisterPropertyEditor(TypeInfo(TPlanetName), TddgPlanet,
    'PlanetName', TPlanetNameProperty);
  RegisterPropertyEditor(TypeInfo(TCommandLine), TddgRunButton, '',
     TCommandLineProperty);

  // Регистрация пользовательских модулей и экспертов библиотек
  RegisterCustomModule(TAppBar, TCustomModule);
  RegisterLibraryExpert(TAppBarExpert.Create);

end;

end.
```

Безопасность компонентов

Можно зарегистрировать компоненты, даже имея только пакет времени выполнения. Для этого создается собственный регистрационный модуль, в котором регистрируются компоненты данного пакета. Затем этот модуль добавляется в отдельный пакет, содержащий пакет времени выполнения в

разделе `requires`. После этого достаточно установить новый пакет в среду Delphi, для того чтобы все компоненты появились в палитре компонентов. Тем не менее, по-прежнему невозможно скомпилировать приложение, использующее эти компоненты, пока будут отсутствовать соответствующие файлы `.dcu` модулей этих компонентов.

Распространение пакетов

При поставке пакетов разработчикам компонентов без их исходного текста, необходимо поместить в дистрибуцию файлы обоих откомпилированных пакетов (`DDGDsgn50.bpl` и `DDGStd50.bpl`), а также оба `*.dcp`-файла и любые скомпилированные модули (`*.dcu`), необходимые для компиляции поставляемых компонентов. Программисты, распространяющие пакеты времени выполнения своих приложений, в которых они используют компоненты из ваших пакетов, должны также вместе со своими приложениями распространять пакет `DDGStd50.bpl` и любые другие пакеты времени выполнения, используемые этими приложениями.

Сценарий 2. Только пакет разработки компонентов

Этот сценарий используется, когда не требуется распространять компоненты в пакетах времени выполнения. В этом случае все компоненты, редакторы компонентов, редакторы свойств, модуль регистрации компонентов помещается в один файл пакета.

Распространение пакетов

При распространении такого пакета без исходного текста следует включить в дистрибутив скомпилированный файл пакета `DDGDsgn50.bpl`, файл `DDGDsgn50.dcp` и все скомпилированные модули (`*.dcu`-файлы), необходимые для компиляции распространяемых компонентов. Программисты, использующие ваши компоненты, обязаны скомпилировать их в свои приложения. Они не смогут распространять ваши компоненты в пакетах времени выполнения.

Сценарий 3. Расширение среды Delphi элементами разработки без компонентов

Этот сценарий применяется, когда среда разработки Delphi дополняется специальными инструментами, например, такими как эксперты. Распространяемые эксперты регистрируются в специальном регистрационном модуле. Принцип распространения такого сценария очень прост — поставляется только `*.bpl`-файл.

Сценарий 4. Дробление приложений

Этот сценарий применяется при дроблении приложения на отдельные логические части, каждая из которых может распространяться отдельно. Существует несколько причин использования такого сценария.

- Этот сценарий легче поддерживать.
- Пользователи могут приобрести лишь минимально необходимую им функциональность. Позднее, когда им потребуется расширить функциональность, они получат пакет, размер которого значительно меньше целого приложения.
- Вносить исправления в части приложения значительно легче, что позволяет пользователям обойтись без повторной загрузки полной версии приложения.

В этом сценарии распространяются лишь `*.bpl`-файлы, необходимые для работы приложения. Такой сценарий подобен предыдущему, но за одним исключением: вместо пакетов для IDE Delphi пользователю предоставляются пакеты для вашего собственного приложения. При таком дроблении приложения следует особое внимание уделить вопросам, связанным с версиями пакетов. Об этом пойдет речь в следующем разделе.

Версии пакетов

Эта тема обычно неверно понимается читателями. Версии пакетов следует рассматривать в том же ключе, что и версии модулей. Другими словами, любой пакет вашего приложения должен быть скомпилирован с использованием той же версии Delphi, что и само приложение. Таким образом, пакет, написанный в Delphi 5, нельзя использовать с приложением Delphi 4. Разработчики фирмы Inprise обычно ссылаются на версию пакета как на *базу кода*. Поэтому пакет, написанный в Delphi 5, имеет базу кода 5.0. Эту идею следует отразить в соглашении об именах, используемых для файлов пакета.

Директивы компилятора, предназначенные для пакетов

Предусмотрено несколько специальных директив компилятора, которые можно вставить в исходный код ваших пакетов. Одни из них относятся к модулям пакетов, другие — к файлам пакетов. Эти директивы приведены в табл. 21.7 и 21.8.

Таблица 21.7. Директивы компилятора для модулей пакета

Директива	Значение
`{$G}` или `{$IMPORTEDDATA OFF}`	Используется для предотвращения помещения модуля в пакет, т.е. в том случае, когда требуется, чтобы этот модуль был непосредственно скомпилирован с приложением. Противоположна директиве `{$WEAKPACKAGEUNIT}`, которая позволяет модулю входить в состав пакета, но код которой статически связан с приложением
`{$DENYPACKAGEUNIT}`	То же самое, что и директива `{$G}`
`{$WEAKPACKAGEUNIT}`	См. раздел "Дополнительная информация о директиве {$WEAKPACKAGEUNIT}"

Таблица 21.8. Директивы компилятора для пакетного файла с расширением .dpk

Директива	Значение
`{$DESIGNONLY ON}`	Компилирует пакет только как пакет разработки
`{$RUNONLY ON}`	Компилирует пакет только как пакет времени выполнения
`{$IMPLICITBUILD OFF}`	Предотвращает дальнейшую перекомпоновку пакета. Используйте эту опцию в пакетах, которые не придется часто изменять

Дополнительная информация о директиве `{$WEAKPACKAGEUNIT}`

Идея *слабого пакета* (weak package) очень проста. Обычно она используется, если пакет ссылается на библиотеки DLL, которые могут отсутствовать. Например, `Vcl50` вызывает программы ядра Win32 API, входящие в состав операционной системы Windows. Многие из этих вызываемых подпрограмм находятся в библиотеках DLL, которые присутствуют далеко не каждом компьютере. Эти вызовы распознаются модулями, содержащими директиву

{$WEAKPACKAGEUNIT}. Включение этой директивы вызывает помещение исходного текста данного модуля в пакет, но не в BPL-, а в DCP-файл (файл .DCP — аналог файла .DCU, а файл .BPL — аналог файла .DLL). Таким образом, любые ссылки на функции этих "слабо пакетированных" модулей статически связываются с приложением вместо использования динамических "пакетных" ссылок.

Эта директива — одна из реже всего употребляемых (возможно, она вам вообще не понадобится). Она позволяет разработчикам Delphi обрабатывать специфические ситуации. Предположим, существует два компонента (каждый в отдельном пакете), ссылающихся на один и тот же интерфейсный модуль библиотеки DLL. Если приложение использует оба компонента, это приведет к загрузке двух экземпляров библиотеки DLL, что вызовет конфликты инициализации и использования глобальных переменных. Решением этой проблемы будет помещение интерфейсного модуля в один из стандартных пакетов Delphi (типа Vcl50.bpl). Однако такой подход не позволяет избавиться от других проблем, связанных с отсутствием специализированных библиотек DLL (например PENWIN.DLL). Если библиотека DLL, указанная в интерфейсном модуле, отсутствует, то пакет Vcl50.bpl (а значит, и Delphi) становится бесполезным. Разработчики Delphi борются с этим, позволяя Vcl50.bpl содержать интерфейсный модуль в отдельном пакете, который присоединяется статически и не загружается динамически при работе с Vcl50.bpl в среде Delphi.

Как уже отмечалось, вы вряд ли прибегнете к этой директиве, если только не предвидите ситуацию, с которой довелось столкнуться создателям Delphi, или если не захотите удостовериться в том, что определенный модуль включается в пакет, но при этом статически связывается с использующим его приложением. Это можно делать, например, с целью оптимизации работы приложения. Обратите внимание на то, что любые слабо пакетированные модули не могут иметь глобальные переменные или код в своих разделах инициализации/завершения. Кроме того, для слабо пакетированных модулей вместе с пакетами следует распространять и *.dcu-файлы.

Соглашения об именах пакетов

Как уже упоминалось, версии пакетов должны отражаться в их именах. Настоятельно рекомендуем (хотя вы и не обязаны это делать) использовать соглашение об именах пакетов с добавлением базы кода. Например, компоненты этой книги находятся в пакете времени выполнения DDGStd50.dpk, имя которого содержит базу кода Delphi 5.0 — 50. То же самое можно сказать и о пакете разработки DDGStd50.dpk. Предыдущей версией этого пакета была бы DdgStd40.dpk. Используя это соглашение, вы избавляете своих пользователей от неприятностей, связанных с согласованием версии пакета и компилятора Delphi. Имя нашего пакета начинается с трехсимвольного идентификатора автора/компании (Ddg), за которым следуют символы Std (для пакета времени выполнения) или Dsgn (для пакета разработки). Вы можете следовать любому соглашению, отвечающему вашим требования, но все же включайте в имена пакетов номер версии Delphi!

Пакеты надстроек

Пакеты надстроек (add-in packages) позволяют дробить приложения на части (или модули), которые будут распространяться отдельно от основного приложения. Такая схема оказывается особенно привлекательной, поскольку дает возможность расширять функциональность приложения без повторной компиляции и внесения изменений в разработку всего приложения. Однако подобный подход требует тщательного архитектурного планирования. Углубле-

ние в вопросы такой разработки выходит за рамки этой книги. За дополнительной информацией о пакетах надстроек и о том, как они вписываются в общую структуру приложений, обращайтесь к статьям, которые можно найти по адресу: http://www.xapware.com.

В данном разделе мы приведем простую иллюстрацию этого метода. Мы покажем, как можно добавить форму к приложению, не переписывая это приложение целиком. Более сложные примеры можно получить по упомянутому выше URL.

Генерирование форм надстроек

В главе 4, "Строение приложения и концепции конструирования" (том I), обсуждалась структура приложений. Мы разработали приложение, формы которого являлись потомками базового класса TChildForm. Теперь мы воспользуемся тем же приложением для иллюстрации методов создания приложения-оболочки, которому известен лишь класс TChildForm, но, тем не менее, оно может работать и с потомками этого класса, которые доступны через пакеты надстроек.

На заметку

Если формы, используемые в демонстрационном приложении из главы 4, вы установили в хранилище объектов (Object Repository), то их придется удалить оттуда перед загрузкой проекта данного приложения.

Это приложение разбито на три логические части: главное приложение (ChildTest.exe), пакет, содержащий класс TChildForm (AIChildForm50.bpl), и конкретные классы, производные от класса TChildForm, каждый из которых расположен в своем собственном пакете.

Главное приложение в основном совпадает с приложением из главы 4, но с небольшой модификацией. Пакет AIChildForm50.bpl содержит *абстрактный* класс TChildForm. В других пакетах содержатся потомки класса TChildForm или *конкретные* классы типа TChildForm. Мы будем называть эти пакеты соответственно *абстрактным пакетом* и *конкретными пакетами*.

Главное приложение использует абстрактный пакет (AIChildForm50.bpl). Каждый конкретный пакет также использует абстрактный пакет. Для обеспечения корректной работы главное приложение должно быть скомпилировано с пакетами времени выполнения, включая пакет AIChildForm50.dcp. Аналогично, каждый конкретный пакет должен использовать пакет AIChildForm50.dcp. Мы не будем приводить здесь текст исходного класса TChildForm или его конкретных потомков, поскольку они мало отличаются от текстов, приведенных в главе 4. Единственное существенное отличие состоит в том, что каждый модуль класса-потомка TChildForm должен включать блоки initialization и finalization, которые имеют следующий вид:

```
initialization
  RegisterClass(TCF2Form);
finalization
  UnRegisterClass(TCF2Form);
```

Обращение к процедуре RegisterClass() вызвано необходимостью сделать потомок класса TChildForm доступным системе обработки потоков главного приложения при загрузке главным приложением своего пакета. Эта процедура аналогична процедуре RegisterComponents(), которая делает компоненты доступными IDE Delphi. Когда пакет не загружен, необходимо обратиться к процедуре UnRegisterClass() для удаления зарегистри-

рованного класса. Заметим, что процедура RegisterClass() делает соответствующий класс доступным только главному приложению. Причем главному приложению все еще остается неизвестным имя класса. Возникает вопрос: как же главное приложение создает экземпляр класса, имени которого оно не знает? Является ли целью данного упражнения сделать эти формы доступными главному приложению без жестко закодированных имен классов в исходном коде главного приложения? В листинге 21.18 представлен исходный текст главной формы главного приложения, в котором мы обратим ваше внимание на реализацию форм надстроек с помощью пакетов надстроек.

Листинг 21.18. Главная форма главного приложения, использующая пакеты надстроек

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls, ChildFrm, Menus;

const
  { Дочерняя форма регистрируется в системном реестре Windows. }
  cCFRegLocation = 'Software\Delphi 5 Developer''s Guide';
  cCFRegSection  = 'ChildForms';  // Раздел инициализации данных модуля

  FMainCaption   = 'Delphi 5 Developer''s Guide Child Form Demo';

type

  TChildFormClass = class of TChildForm;

  TMainForm = class(TForm)
    pnlMain: TPanel;
    Splitter1: TSplitter;
    pnlParent: TPanel;
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiExit: TMenuItem;
    mmiHelp: TMenuItem;
    mmiForms: TMenuItem;
    procedure mmiExitClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    // ссылка на дочернюю форму.
    FChildForm: TChildForm;
    // список доступных дочерних форм, используемых для построения меню.
    FChildFormList: TStringList;
    // Индекс к меню Close Form (закрыть форму) для смещения позиции.
    FCloseFormIndex: Integer;
    // Дескриптор загруженного в данный момент пакета.
```

```
    FCurrentModuleHandle: HModule;
    // метод создания меню для доступных дочерних форм.
    procedure CreateChildFormMenus;
    // Обработчик загрузки дочерней формы и ее пакета.
    procedure LoadChildFormOnClick(Sender: TObject);
    // Обработчик выгрузки дочерней формы и ее пакета.
    procedure CloseFormOnClick(Sender: TObject);
    // Метод считывания имени потомка класса TChildForm
    function GetChildFormClassName(const AModuleName: String): String;
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation
uses Registry;

{$R *.DFM}

function RemoveExt(const AFileName: String): String;
{ Вспомогательная функция удаления расширения из имени файла. }
begin
  if Pos('.', AFileName) <> 0 then
    Result := Copy(AFileName, 1, Pos('.', AFileName)-1)
  else
    Result := AFileName;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  FChildFormList := TStringList.Create;
  CreateChildFormMenus;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  FChildFormList.Free;
  // Выгружаем любые загруженные дочерние формы.
  if FCurrentModuleHandle <> 0 then
    CloseFormOnClick(nil);
end;

procedure TMainForm.CreateChildFormMenus;
```

```
{ Все доступные дочерние формы регистрируются в системном реестре Windows.
  Здесь мы используем эту информацию для создания элементов меню, обеспечивающих
  загрузку всех дочерних форм. }
var
  IniFile: TRegIniFile;
  MenuItem: TMenuItem;
  i: integer;
begin
  inherited;

  { Считываем список всех дочерних форм и строим меню на базе
    записей в системном реестре. }
  IniFile := TRegIniFile.Create(cCFRegLocation);
  try
    IniFile.ReadSectionValues(cCFRegSection, FChildFormList);
  finally
    IniFile.Free;
  end;

  { Добавляем элементы меню для каждого модуля. ОБРАТИТЕ ВНИМАНИЕ на то,
    что свойство mmMain.AutoHotKeys должно быть установлено
    равным значению maAutomatic }

for i := 0 to FChildFormList.Count - 1 do
  begin
    MenuItem := TMenuItem.Create(mmMain);
    MenuItem.Caption := FChildFormList.Names[i];
    MenuItem.OnClick := LoadChildFormOnClick;
    mmiForms.Add(MenuItem);
  end;

  // Создаем разделитель
  MenuItem := TMenuItem.Create(mmMain);
  MenuItem.Caption := '-';
  mmiForms.Add(MenuItem);

  // Создаем элемент меню Close Form (закрыть форму)
  MenuItem := TMenuItem.Create(mmMain);
  MenuItem.Caption := '&Close Form';
  MenuItem.OnClick := CloseFormOnClick;
  MenuItem.Enabled := False;
  mmiForms.Add(MenuItem);

  { Сохраняем ссылку на индекс элемента меню, необходимый для
    закрытия дочерней формы. Она будет использована в другом методе. }
  FCloseFormIndex := MenuItem.MenuIndex;
end;

procedure TMainForm.LoadChildFormOnClick(Sender: TObject);
var
```

```
  ChildFormClassName: String;
  ChildFormClass: TChildFormClass;
  ChildFormName: String;
  ChildFormPackage: String;
begin

  // Заголовок меню представляет имя модуля.
  ChildFormName := (Sender as TMenuItem).Caption;
  // Получаем реальное имя файла пакета.
  ChildFormPackage := FChildFormList.Values[ChildFormName];

  // Выгружаем любые ранее загруженные пакеты.
  if FCurrentModuleHandle <> 0 then
    CloseFormOnClick(nil);

  try
    // Загружаем заданный пакет
    FCurrentModuleHandle := LoadPackage(ChildFormPackage);

    // Возвращаем имя класса, необходимое для создания экземпляра
    ChildFormClassName := GetChildFormClassName(ChildFormPackage);

    { Создаем экземпляр класса с помощью процедуры FindClass(). Заметьте,
      нужно, чтобы этот класс уже был зарегистрирован
      в системе обработки потоков с помощью процедуры RegisterClass().
      Это реализуется в разделе инициализации дочерней формы
      для каждого пакета дочерней формы. }
    ChildFormClass := TChildFormClass(FindClass(ChildFormClassName));
    FChildForm := ChildFormClass.Create(self, pnlParent);
    Caption := FChildForm.GetCaption;
    FChildForm.Show;

    mmiForms[FCloseFormIndex].Enabled := True;
  except
    on E: Exception do
    begin
      CloseFormOnClick(nil);
      raise;
    end;
  end;
end;

function TMainForm.GetChildFormClassName(const AModuleName: String): String;
{ Имя реального потомка класса TChildForm находится в системном
  реестре. Этот метод считывает имя этого класса. }
var
  IniFile: TRegIniFile;
begin
  IniFile := TRegIniFile.Create(cCFRegLocation);
  try
```

```
    Result := IniFile.ReadString(RemoveExt(AModuleName), 'ClassName',
      EmptyStr);
  finally
    IniFile.Free;
  end;
end;

procedure TMainForm.CloseFormOnClick(Sender: TObject);
begin
  if FCurrentModuleHandle <> 0 then
  begin
    if FChildForm <> nil then
    begin
      FChildForm.Free;
FChildForm := nil;
    end;

    // Отмена регистрации любых классов модуля
    UnRegisterModuleClasses(FCurrentModuleHandle);
    // Выгрузка пакета дочерней формы
    UnloadPackage(FCurrentModuleHandle);

    FCurrentModuleHandle := 0;
    mmiForms[FCloseFormIndex].Enabled := False;
    Caption := FMainCaption;
  end;
end;

end.
```

Логика этого приложения довольно проста. В нем для определения доступных пакетов используется системный реестр. Имена найденных пактов помещаются в строки меню, создаваемого для предоставления команды загрузки каждого пакета. Кроме того, считывается имя класса формы, содержащейся в каждом пакете.

На заметку

На компакт-диск мы поместили файл D5DG.Reg (на его имени можно дважды щелкнуть в окне Windows Explorer). Он импортирует параметры реестра, благодаря чему демонстрационный пакет надстройки будет работать надлежащим образом.

Основную нагрузку несет на себе обработчик события LoadChildFormOnClick(). После определения имени пакета этот метод загружает нужный пакет с помощью функции LoadPackage(). (Функция LoadPackage() принципиально ничем не отличается от функции LoadLibrary(), используемой для загрузки библиотек DLL.) Затем этот метод определяет имя класса для формы, содержащейся в загруженном пакете.

Для создания класса нужна ссылка на имя класса (например, TButton или TForm1). Однако в этом главном приложении отсутствует жестко закодированное имя класса конкретной дочерней формы типа TChildForm. Поэтому мы считываем имя класса из системного реестра. Главное приложение может передать это имя класса функции FindClass() и при этом получить ссылку на заданный класс, который уже был зарегистрирован системой обработки пото-

ков. Помните, что регистрация выполняется в разделе инициализации модуля конкретной формы, который вызывается при загрузке соответствующего пакета. Затем мы создаем класс, используя следующие строки программы:

```
ChildFormClass := TChildFormClass(FindClass(ChildFormClassName));
FChildForm := ChildFormClass.Create(self, pnlParent);
```

Переменная ChildFormClass представляет собой заранее определенную ссылку на класс TChildForm и может послужить ссылкой на потомок класса TChildForm.

Обработчик события CloseFormOnClick() просто закрывает дочернюю форму и выгружает ее пакет. Остальная часть кода используется в основном для создания меню пакетов и чтения информации из системного реестра.

Более углубленное изучение этой технологии позволит создавать очень гибкие и слабо связанные структуры приложений.

Резюме

Знание принципов работы компонентов — основа понимания Delphi. В этой книге вам еще предстоят встречи со многими пользовательскими компонентами. Надеемся, что теперь компоненты не представляются вам "черными ящиками". В следующей главе вы познакомитесь с еще более сложными способами создания компонентов.

Глава

22 Сложные методики работы с компонентами

Псевдовизуальные компоненты	131
Анимированные компоненты	135
Создание редакторов свойств	149
Редакторы компонентов	159
Работа непубликуемых компонентов с потоками данных	164
Категории свойств	174
Списки компонентов: классы `TCollection` и `TCollectionItem`	179
Резюме	196

В предыдущей главе рассматривались вопросы разработки пользовательских компонентов Delphi. В данной главе вашему вниманию предлагаются некоторые более сложные методики разработки компонентов, освоив которые вы сможете поднять свое мастерство в этой области на более высокий уровень. Использование продвинутых методик демонстрируется на примерах построения псевдовизуальных компонентов, детализированных редакторов свойств, редакторов компонентов и коллекций.

Псевдовизуальные компоненты

Вы уже знакомы с визуальными компонентами, такими как `TButton` и `TEdit`, и с невизуальными компонентами, такими как `TTable` и `TTimer`. В этом разделе вы познакомитесь с компонентами, занимающими промежуточное положение между визуальными и невизуальными, — назовем их *псевдовизуальными компонентами.*

Расширенные подсказки

Примером псевдовизуального компонента, рассматриваемого в этом разделе, является расширение всплывающего окна подсказки (hint) Delphi. Этот компонент назван псевдовизуальным, поскольку палитра компонентов во время разработки не воспринимает его как визуальный объект, но во время выполнения этот компонент отображается как всплывающее окно подсказки.

Замена обычного стиля окна подсказки новым требует выполнения следующих действий.

- Создание потомка класса `THintWindow`.
- Уничтожение старого класса окна подсказки.
- Назначение нового класса окна подсказки.
- Создание нового класса окна подсказки.

Создание потомка класса `THintWindow`

Прежде чем приступить к написанию кода потомка класса `THintWindow`, следует решить, чем поведение нового класса окна подсказки будет отличаться от поведения обычного окна подсказки. В нашем случае вместо обычного прямоугольного создано эллиптическое окно подсказки. Попутно вы ознакомитесь с интересной методикой — с созданием непрямоугольных окон. В листинге 22.1 приводится текст модуля `RndHint.pas`, содержащего компонент `TDDGHintWindow`, который является потомком класса `THintWindow`.

Листинг 22.1. Модуль `RndHint.pas` — иллюстрация вывода подсказки в эллиптическом окне

```
unit RndHint;

interface

uses Windows, Classes, Controls, Forms, Messages, Graphics;

type
  TDDGHintWindow = class(THintWindow)
  private
```

```
    FRegion: THandle;
    procedure FreeCurrentRegion;
  public
    destructor Destroy; override;
    procedure ActivateHint(Rect: TRect; const AHint: string); override;
    procedure Paint; override;
    procedure CreateParams(var Params: TCreateParams); override;
  end;

implementation

destructor TDDGHintWindow.Destroy;
begin
  FreeCurrentRegion;
  inherited Destroy;
end;

procedure TDDGHintWindow.FreeCurrentRegion;
{ Области окна, подобно другим объектам API, должны быть освобождены
  по завершении их использования. Запомните, что нельзя удалить
  в окне область, являющуюся текущей, поэтому перед удалением объекта
  области в данном методе область окна устанавливается равной значению 0. }
begin
  if FRegion <> 0 then begin       // Если область жива...
    SetWindowRgn(Handle, 0, True); // Установка области окна равной 0
    DeleteObject(FRegion);         // Уничтожение области
    FRegion := 0;                  // Обнуление значения поля
  end;
end;

procedure TDDGHintWindow.ActivateHint(Rect: TRect; const AHint: string);
{ Вызывается, когда подсказка активизируется при помещении
  указателя мыши поверх соответствующего элемента управления. }
begin
  with Rect do
    Right := Right + Canvas.TextWidth('WWWW'); // Добавим немного пространства
  BoundsRect := Rect;
  FreeCurrentRegion;
  with BoundsRect do
    { Создание прямоугольной области со скругленными углами,
      предназначенной для отображения окна подсказки. }
    FRegion := CreateRoundRectRgn(0, 0, Width, Height, Width, Height);
  if FRegion <> 0 then
    SetWindowRgn(Handle, FRegion, True); // Установка области окна
  inherited ActivateHint(Rect, AHint);   // Вызов унаследованного метода
end;

procedure TDDGHintWindow.CreateParams(var Params: TCreateParams);
{ Необходимо удалить обрамление окна, созданное на уровне
  Windows API при прорисовке окна. }
```

```
begin
  inherited CreateParams(Params);
  Params.Style := Params.Style and not ws_Border;  // Удаление обрамления
end;

procedure TDDGHintWindow.Paint;
{ Этот метод вызывается обработчиком события WM_PAINT.
  Он отвечает за отображение окна подсказки. }
var
  R: TRect;
begin
  R := ClientRect;                  // Получает границы прямоугольника
  Inc(R.Left, 1);                   // Сдвигает немного влево
  Canvas.Font.Color := clInfoText;  //Устанавливает надлежащий цвет
  { Рисуем строку посредине скругленного прямоугольника. }
  DrawText(Canvas.Handle, PChar(Caption), Length(Caption), R,
         DT_NOPREFIX or DT_WORDBREAK or DT_CENTER or DT_VCENTER);
end;

initialization
  Application.ShowHint := False;     // Удаление старого окна подсказки
  HintWindowClass := TDDGHintWindow; // Присвоение нового окна подсказки
  Application.ShowHint := True;      // Создание нового окна подсказки
end.
```

Организация переопределенных методов `CreateParams()` и `Paint()` очевидна. Метод `CreateParams()` позволяет настроить структуру стилей окна перед тем, как оно будет создано на уровне API. В этом методе с помощью операции `not` маскируется стиль `WS_BORDER` для того, чтобы вокруг окна не рисовалась прямоугольная рамка. Метод `Paint()` отвечает за прорисовку окна. В нашем случае метод должен поместить свойство подсказки `Caption` в центр заголовка окна. Для текста установлен цвет `clInfoText`, который является определенным системой цветом подсказки.

Эллиптическое окно

Вся магия создания непрямоугольных окон заключена в методе `ActivateHint()`. В действительности же это вовсе не магия, а просто два вызова функций Win32 API: `CreateRoundRectRgn()` и `SetWindowRgn()`.

Функция `CreateRoundRectRgn()` определяет прямоугольную область со скругленными углами внутри заданного окна. *Область* (region) — это специальный объект API, позволяющий выполнить прорисовку, закрашивание, обрезание и анализ необходимости вывода подсказки в некоторой области. Помимо функции `CreateRoundRectRgn()`, существуют и другие функции Win32 API, предназначенные для создания областей различных типов:

- `CreateEllipticRgn()`;
- `CreateEllipticRgnIndirect()`;
- `CreatePolygonRgn()`;
- `CreatePolyPolygonRgn()`;
- `CreateRectRgn()`;

- CreateRectRgnIndirect();
- CreateRoundRectRgn();
- ExtCreateRegion().

Кроме того, можно использовать функцию CombineRgn() для объединения нескольких различных областей в один сложный участок. Все эти функции подробно описаны в интерактивной справке Win32 API.

Затем вызывается функция SetWindowRgn(), которой в качестве параметра передается дескриптор вновь созданной области. Эта функция заставляет операционную систему стать владельцем области, и все последующие перерисовки для данного окна будут происходить только внутри указанной области. Таким образом, если область определена как скругленный прямоугольник, перерисовка будет выполняться только в нем.

Вам следует знать о двух побочных эффектах применения функции SetWindowRgn(). Во-первых, так как перерисовывается только часть окна, у этого окна, вероятно, не будет рамки и заголовка. Вы должны быть готовы к тому, что придется обеспечить пользователя альтернативным способом перемещения, изменения размера и закрытия окна — без помощи рамки или строки заголовка. Во-вторых, поскольку операционная система владеет областью, определенной в функции SetWindowRgn(), необходимо осторожно обращаться с ней, т.е. не изменять и не удалять эту область, пока она находится в работе. В компоненте TDDGHintWindow сначала вызывается метод FreeCurrentRegion() и лишь затем уничтожается старое или создается новое окно.

Активизация потомка класса THintWindow

Код инициализации модуля RndHint делает компонент TDDGHintWindow активным окном подсказки всего приложения. Установка свойства Application.ShowHint равным значению False приводит к уничтожению старого окна подсказки. Далее следует присвоить класс, являющийся потомком класса THintWindow, глобальной переменной HintWindowClass. Теперь можно установить свойство Application.ShowHint равным значению True. В результате чего будет создано новое окно, теперь уже представляющее экземпляр вашего класса-потомка.

На рис. 22.1 компонент TddgHintWindow представлен в действии.

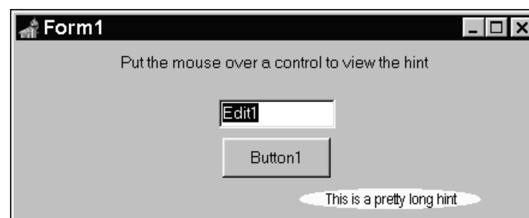


Рис. 22.1. Внешний вид компонента окна подсказки TddgHintWindow

Использование компонента TddgHintWindow

Использование псевдовизуального компонента отличается от методов использования обычных визуальных или невизуальных компонентов. Поскольку вся работа по созданию экземпляра компонента выполняется в разделе инициализации (initialization) его модуля, компонент не может быть добавлен в пакет разработки для помещения его в палитру компонентов. Поэтому необходимо просто добавлять имя модуля в секции uses исходных файлов проекта.

Анимированные компоненты

Завершая разработку очередного приложения в Delphi, мы были вполне удовлетворены полученными результатами, вот только окно About казалось нам несколько “скучным”. Требовалось что-то придумать. Внезапно кому-то из нас пришла в голову мысль создать новый компонент, который позволил бы включать в диалоговое окно About бегущую полосу с титрами.

Компонент бегущих титров

Давайте уделим некоторое время анализу того, как работает компонент бегущих титров. Он может прокручивать пачку строк в окне компонента как настоящую бегущую ленту с титрами. Для компонента `TddgMarquee` в качестве базового следует использовать класс `TCustomPanel`, так как он имеет необходимую базовую функциональность и красивую трехмерную рамку.

Компонент `TddgMarquee` переводит текстовые строки в растровое изображение, находящееся в памяти, и затем копирует части этого изображения на собственную канву с целью имитации эффекта прокрутки. Для этого используется функция API `BitBlt()`, копирующая часть изображения в памяти, соответствующую размеру элемента управления, на его канву, начиная с верхней части. Затем эта область перемещается на несколько пикселей вниз и снова копируется на канву элемента. Она многократно поочередно смещается и копируется в окно, пока не доходит до конца изображения в памяти.

Теперь нужно решить, какие дополнительные классы следует интегрировать в компонент `TddgMarquee` для того, чтобы оживить его. Во-первых, для хранения прокручиваемых строк потребуется объект `TStringList`. Во-вторых, в памяти должно содержаться растровое изображение со строками, которые нужно прокручивать. Для этого вполне подойдет компонент VCL `TBitmap`.

Написание кода компонента

Как и для предыдущих компонентов, сначала нужно составить план написания текста класса `TddgMarquee`. Разобьем задачу на несколько подзадач. Логически компонент `TddgMarquee` может быть разделен на следующие пять частей.

- Механизм переноса текста в растровое изображение, хранящееся в памяти.
- Механизм копирования текста из изображения в памяти — в окно титров.
- Таймер контроля процесса прокрутки.
- Конструктор, деструктор класса и связанные с ними методы.
- Завершающие штрихи в виде вспомогательных свойств и методов.

Работа с изображением в памяти

Создавая экземпляр класса `TBitmap`, необходимо знать, какого он должен быть размера, чтобы вмещать весь список строк в памяти. Этот размер определяется путем умножения высоты строки на общее число строк. Функция API `GetTextMetrics()`, которой в качестве параметра передается дескриптор канвы, предназначена для определения размеров строки текста, набранного конкретным шрифтом. В процессе выполнения эта функция заполняет данными запись `TTextMetric`:

```
var
  Metrics: TTextMetric;
begin
  GetTextMetrics(Canvas.Handle, Metrics);
```

На заметку

Функция API GetTextMetrics() соответствующим образом модифицирует запись типа TTextMetric, содержащую много полезной информации о выбранном в данный момент шрифте контекста устройства. Эта функция предоставляет информацию не только о высоте и ширине шрифта, но и о его начертании: полужирном, курсивном, перечеркнутом, а также предоставляет информацию о названии шрифта.
Метод TextHeight() класса TCanvas здесь не работает. Этот метод определяет только размеры конкретной строки текста, а не шрифта в целом.

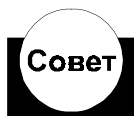
Высота ячейки символа в текущем шрифте канвы хранится в поле tmHeight записи Metric. Если сложить это значение со значением поля tmInternalLeading, хранящим междустрочное расстояние выбранного шрифта, то получим высоту строки текста в растровом изображении в памяти:

```
LineHi := Metrics.tmHeight + Metrics.tmInternalLeading;
```

Общая высота изображения в памяти определяется умножением значения LineHi на количество строк и добавлением удвоенной высоты канвы элемента управления TddgMarquee для создания пустого изображения в начале и конце титров. Предположим, что элемент типа TStringList, в котором находятся все строки, носит имя FItems, и поместим общие размеры изображения в структуру TRect:

```
var
  VRect: TRect;
begin
  { Прямоугольник VRect - растровое изображение в памяти }
 VRect := Rect(0, 0, Width, LineHi * FItems.Count + Height * 2);
end;
```

После того как экземпляр растрового изображения создан и его размеры изменены требуемым образом, выполняется его дальнейшая инициализация с установкой шрифта равным значению свойства TddgMarquee.Font, заливкой фона цветом, определенным свойством TddgMarquee.Color, и установкой свойства Brush.Style равным значению bsClear.



При размещении текста в объекте класса TCanvas цвет фона определяется значением свойства TCanvas.Brush. Для того чтобы фон был невидимым, нужно установить свойство TCanvas.Brush.Style равным значению bsClear.

Итак, основная работа сделана, и теперь можно поместить текст в растровое изображение в памяти. В главе 8, “GDI, шрифты и графика” (том I), рассматривалось несколько способов помещения текста в канву. Наиболее очевидный из них — это использование метода TextOut() класса TCanvas, однако б*о*льшие возможности по управлению форматированием текста предоставляет более сложная функция API DrawText(). Поскольку нам необходимо управлять выравниванием, в компоненте TddgMarquee будет использоваться именно функция DrawText(). Перечислимый тип идеально подходит для представления выравнивания текста:

```
type
  TJustification = (tjCenter, tjLeft, tjRight);
```

В следующем фрагменте программы показан метод PaintLine() класса TddgMarquee, позволяющий поместить текст в изображение в памяти. В этом методе используется переменная FJust — экземпляр типа TJustification. Вот этот код:

```
procedure TddgMarquee.PaintLine(R: TRect; LineNum: Integer);
{ Этот метод вызывается для помещения каждой строки
  текста в область памяти MemBitmap. }
const
  Flags: array[TJustification] of DWORD = (DT_CENTER, DT_LEFT, DT_RIGHT);
var
  S: string;
begin
  { Для упрощения работы скопируем строку в локальную переменную }
  S := FItems.Strings[LineNum];
  { Помещает строку текста в растровое изображение в памяти }
  DrawText(MemBitmap.Canvas.Handle, PChar(S), Length(S), R,
    Flags[FJust] or DT_SINGLELINE or DT_TOP);
end;
```

Отображение компонента

Теперь вы уже знаете, как создать растровое изображение в памяти и поместить в него текст. Следующим шагом будет копирование этого текста в канву TddgMarquee.

Метод Paint() компонента вызывается в ответ на сообщение Windows WM_PAINT. Именно метод Paint() оживляет компонент — он рисует и раскрашивает его изображение, формируя визуальное представление компонента.

Задача метода TddgMarquee.Paint() заключается в копировании строк изображения в памяти на канву компонента TddgMarquee. Для этого используется функция Win32 API BitBlt(), копирующая содержимое контекста одного устройства в контекст другого.

Для определения текущего состояния компонента TddgMarquee используется булева переменная FActive, указывающая, активизирован ли в данный момент процесс прокрутки. Очевидно, что метод Paint() должен работать по-разному, в зависимости от того, активен данный компонент или нет:

```
procedure TddgMarquee.Paint;
{ Этот виртуальный метод вызывается в ответ на сообщение Windows. }
begin
  if FActive then
    { Копирование содержимого растрового изображения из памяти на экран. }
    BitBlt(Canvas.Handle, 0, 0, InsideRect.Right, InsideRect.Bottom,
      MemBitmap.Canvas.Handle, 0, CurrLine, srcCopy)
  else
    inherited Paint;
end;
```

Если компонент активен, то с помощью функции BitBlt() часть хранимого в памяти изображения отображается на экране в канве объекта TddgMarquee. Обратите внимание на переменную CurrLine, передаваемую функции BitBlt() как предпоследний параметр. Значение этого параметра определяет, какую часть изображения в памяти нужно выводить на экран. Постоянно увеличивая или уменьшая значение CurrLine, можно добиться эффекта прокрутки текста в компоненте TddgMarquee вниз или вверх.

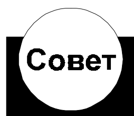
Анимация полосы титров

Визуальные аспекты работы компонента TddgMarquee нами уже реализованы. Для того чтобы наш компонент заработал, осталось сделать совсем немного. Нужно разработать механизм, изменяющий значение CurrLine во времени и включающий в нужный момент перерисовку компонента. Этого легко добиться, используя компонент Ttimer Delphi.

Естественно, перед тем как использовать компонент TTimer, следует создать и инициализировать экземпляр этого класса. Компонент TddgMarquee будет включать собственный экземпляр класса TTimer — переменную FTimer, которая инициализируется в процедуре DoTimer:

```
procedure DoTimer;
{ Процедура настраивает таймер компонента TddgMarquee. }
begin
  FTimer := TTimer.Create(Self);
  with FTimer do
  begin
    Enabled := False;
    Interval := TimerInterval;
    OnTimer := DoTimerOnTimer;
  end;
end;
```

В этой процедуре вначале создается неактивный объект FTimer. Его свойству Interval присваивается значение константы TimerInterval. Наконец, событию FTimer.OnTimer присваивается метод TddgMarquee.DoTimerOnTimer. Этот метод будет вызываться при наступлении каждого события OnTimer.



Присваивая значения событию в коде, вы должны следовать двум правилам.

- Процедура, которая присваивается событию, должна быть методом экземпляра некоторого объекта. Это не может быть самостоятельная процедура или функция.
- Метод, который вы присваиваете событию, должен иметь тот же список параметров, что и тип события. Например, событие OnTimer компонента TTimer имеет тип TNotifyEvent. Поскольку у TNotifyEvent — всего один параметр, Sender типа TObject, то любой метод, присваиваемый событию OnTimer, должен иметь один параметр типа TObject.

Метод DoTimerOnTimer() определяется следующим образом:

```
procedure TddgMarquee.DoTimerOnTimer(Sender: TObject);
{ Этот метод выполняется в ответ на событие таймера. }
begin
  IncLine;
  { Перерисовка только в пределах прямоугольной области. }
  InvalidateRect(Handle, @InsideRect, False);
end;
```

В этом методе вызывается процедура IncLine(), увеличивающая или уменьшающая значение CurrLine. Затем функция Win32 API InvalidateRect() вызывается для перерисовки внутренней части компонента. Мы предпочли использовать функцию InvalidateRect() вместо ме-

тода Invalidate() класса TCanvas, поскольку последний приводит к перерисовке всей канвы, а выбранная нами функция — только ее части. Это позволяет избавиться от неприятного мерцания, связанного с перерисовкой всего окна компонента. Запомните: мерцания следует избегать!

Исходный текст метода IncLine(), обновляющего значение CurrLine и определяющего конец прокрутки, приводится ниже.

```
procedure TddgMarquee.IncLine;
{ Этот метод вызывается для приращения индекса текущей строки. }
begin
  if not FScrollDown then  // Титры прокручиваются вверх,
  begin
    { Проверка конца прокрутки. }
    if FItems.Count * LineHi + ClientRect.Bottom -
      ScrollPixels  >= CurrLine then
      { Если нет, происходит приращение индекса текущей строки. }
      Inc(CurrLine, ScrollPixels)
    else SetActive(False);
  end
  else begin         // Титры прокручиваются вниз
    { Проверка конца прокрутки. }
    if CurrLine >= ScrollPixels then
      { Еще не в конце, происходит уменьшение индекса текущей строки. }
      Dec(CurrLine, ScrollPixels)
    else SetActive(False);
  end;
end;
```

Конструктор класса TddgMarquee довольно прост. Он вызывает унаследованный метод Create(), создает экземпляр класса TStringList, устанавливает значение поля FTimer и значения по умолчанию для переменных экземпляра. Еще раз напоминаем о необходимости использовать в компонентах унаследованный конструктор Create(). Без этого ваши компоненты будут лишены такой важной и полезной функциональности, как создание дескриптора и канвы, возможность работы с потоками данных и взаимодействие с сообщениями Windows. Вот исходный код конструктора TddMarquee.Create():

```
constructor TddgMarquee.Create(AOwner: TComponent);
{ Конструктор класса TddgMarquee. }

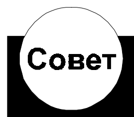
  procedure DoTimer;
  { Процедура установки таймера TddgMarquee. }
  begin
    FTimer := TTimer.Create(Self);
    with FTimer do
    begin
      Enabled := False;
      Interval := TimerInterval;
      OnTimer := DoTimerOnTimer;
    end;
  end;

begin
```

```
  inherited Create(AOwner);
  FItems := TStringList.Create;  { Создание экземпляра списка строк }
  DoTimer;                       { Установка таймера }
  { Установка переменным экземпляра значений по умолчанию }
  Width := 100;
  Height := 75;
  FActive := False;
  FScrollDown := False;
  FJust := tjCenter;
  BevelWidth := 3;
end;
```

Деструктор TddgMarquee еще проще, он деактивизирует компонент, передавая в метод SetActive() параметр False, освобождает таймер и список строк и затем вызывает унаследованный метод Destroy():

```
destructor TddgMarquee.Destroy;
{ Деструктор класса TddgMarquee }
begin
  SetActive(False);
  FTimer.Free;  // Освобождение памяти, выделенной для объектов
  FItems.Free;
  inherited Destroy;
end;
```



Советуем придерживаться правила, согласно которому при переопределении конструкторов унаследованный конструктор вызывается в начале, а при переопределении деструкторов унаследованный деструктор вызывается в самом конце. Это гарантирует, что класс будет создан до его модификации и что все зависимые ресурсы будут освобождены до освобождения класса.
Естественно, из этого правила, как и из любого другого, есть исключения. Однако рекомендуем вам строго придерживаться его, если только нет веских причин его нарушить.

Метод SetActive(), служащий методом записи свойства Active и вызываемый методом IncLine() и деструктором, является механизмом, обеспечивающим начало и останов прокрутки титров в канве:

```
procedure TddgMarquee.SetActive(Value: Boolean);
{ Вызывается для активизации/деактивизации прокрутки титров }
begin
  if Value and (not FActive) and (FItems.Count > 0) then
  begin
    FActive := True;        // Установка флага активности
    MemBitmap := TBitmap.Create;
    FillBitmap;             // Помещение Image в растровое изображение
    FTimer.Enabled := True; // Запуск таймера
  end
  else if (not Value) and FActive then
  begin
```

```
    FTimer.Enabled := False; // Отключение таймера
    if Assigned(FOnDone)     // Обработка события OnDone
      then FOnDone(Self);
    FActive := False;        // Установка переменной FActive равной False
    MemBitmap.Free;          // Освобождение изображения в памяти
    Invalidate;              // Очистка окна элемента управления
  end;
end;
```

У компонента TddgMarquee пока отсутствует важное событие, которое сообщало бы пользователю о конце прокрутки. Ничего страшного, это событие FOnDone легко добавить в наш компонент. Первым шагом будет объявление переменной экземпляра события некоторого типа в разделе private определения класса. Событие FOnDone будет иметь тип TNotifyEvent:

```
FOnDone: TNotifyEvent;
```

Затем это событие следует объявить в качестве свойства в разделе published определения класса:

```
property OnDone: TNotifyEvent read FOnDone write FOnDone;
```

Вспомним, что директивы read и write здесь определяют функции или переменные для установки и получения значения свойства.

Этих двух строк уже достаточно, чтобы свойство OnDone появилось во вкладке Events окна инспектора объектов во время разработки. Осталось лишь вызвать пользовательский обработчик события OnDone (как метод, назначенный событию OnDone), что и делается в тексте метода Deactivate() компонента TddgMarquee:

```
if Assigned(FOnDone) then FOnDone(Self); // Генерация события OnDone
```

Эту строку нужно понимать следующим образом: "Если пользователь компонента присвоил событию OnDone какой-то метод, нужно вызвать этот метод и передать ему экземпляр класса TddgMarquee — Self — в качестве параметра".

В листинге 22.2 приведен полный исходный текст модуля Marquee. Обращаем ваше внимание на то, что, поскольку рассматриваемый компонент является потомком класса TCustom*XXX*, необходимо сделать публикуемыми многие свойства компонента TCustomPanel.

Листинг 22.2. Исходный код модуля Marquee.pas — реализация компонента TddgMarquee

```
unit Marquee;

interface

uses
  SysUtils, Windows, Classes, Forms, Controls, Graphics,
  Messages, ExtCtrls, Dialogs;

const
  ScrollPixels = 3;   // Количество пикселей для каждой прокрутки
```

```
  TimerInterval = 50; // Время между прокрутками в мс

type
  TJustification = (tjCenter, tjLeft, tjRight);

  EMarqueeError = class(Exception);

  TddgMarquee = class(TCustomPanel)
  private
    MemBitmap: TBitmap;
    InsideRect: TRect;
    FItems: TStringList;
    FJust: TJustification;
    FScrollDown: Boolean;
    LineHi : Integer;
    CurrLine : Integer;
    VRect: TRect;
    FTimer: TTimer;
    FActive: Boolean;
    FOnDone: TNotifyEvent;
    procedure SetItems(Value: TStringList);
    procedure DoTimerOnTimer(Sender: TObject);
    procedure PaintLine(R: TRect; LineNum: Integer);
    procedure SetLineHeight;
    procedure SetStartLine;
    procedure IncLine;
    procedure SetActive(Value: Boolean);
  protected
    procedure Paint; override;
    procedure FillBitmap; virtual;
  public
    property Active: Boolean read FActive write SetActive;
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property ScrollDown: Boolean read FScrollDown write FScrollDown;
    property Justify: TJustification read FJust write FJust default tjCenter;
    property Items: TStringList read FItems write SetItems;
    property OnDone: TNotifyEvent read FOnDone write FOnDone;
    { Публикуем унаследованные свойства: }
    property Align;
    property Alignment;
    property BevelInner;
    property BevelOuter;
    property BevelWidth;
    property BorderWidth;
    property BorderStyle;
    property Color;
    property Ctl3D;
    property Font;
```

```
    property Locked;
    property ParentColor;
    property ParentCtl3D;
    property ParentFont;
    property Visible;
    property OnClick;
    property OnDblClick;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnResize;
  end;

implementation

constructor TddgMarquee.Create(AOwner: TComponent);
{ Конструктор класса TddgMarquee }

  procedure DoTimer;
  { Процедура установки таймера TddgMarquee }
  begin
    FTimer := TTimer.Create(Self);
    with FTimer do
    begin
      Enabled := False;
      Interval := TimerInterval;
      OnTimer := DoTimerOnTimer;
    end;
  end;

begin
  inherited Create(AOwner);
  FItems := TStringList.Create;  { Создание экземпляра списка строк }
  DoTimer;                       { Установка таймера }
  { Установка переменным экземпляра значений по умолчанию }
  Width := 100;
  Height := 75;
  FActive := False;
  FScrollDown := False;
  FJust := tjCenter;
  BevelWidth := 3;
end;

destructor TddgMarquee.Destroy;
{ Деструктор класса TddgMarquee }
begin
  SetActive(False);
  FTimer.Free;  // Освобождает объекты, для которых была выделена память
  FItems.Free;
  inherited Destroy;
```

```
end;
procedure TddgMarquee.DoTimerOnTimer(Sender: TObject);
{ Этот метод выполняется в ответ на событие таймера }
begin
  IncLine;
  { Перерисовка только внутри прямоугольной области }
  InvalidateRect(Handle, @InsideRect, False);
end;

procedure TddgMarquee.IncLine;
{ Этот метод вызывается для приращения индекса текущей строки }
begin
  if not FScrollDown then // Титры прокручиваются вверх
  begin
    { Проверка конца прокрутки }
    if FItems.Count * LineHi + ClientRect.Bottom -
      ScrollPixels  >= CurrLine then
      { Если нет, выполняется приращение индекса текущей строки }
      Inc(CurrLine, ScrollPixels)
    else SetActive(False);
  end
  else begin              // Титры прокручиваются вниз
    { Проверка конца прокрутки }
    if CurrLine >= ScrollPixels then
      { Еще не в конце, происходит уменьшение индекса текущей строки }
      Dec(CurrLine, ScrollPixels)
    else SetActive(False);
  end;
end;

procedure TddgMarquee.SetItems(Value: TStringList);
begin
  if FItems <> Value then
    FItems.Assign(Value);
end;

procedure TddgMarquee.SetLineHeight;
{ Этот виртуальный метод устанавливает переменную экземпляра LineHi }
var
  Metrics : TTextMetric;
begin
  { Получаем метрику шрифта }
  GetTextMetrics(Canvas.Handle, Metrics);
  { Настройка высоты строки }
  LineHi := Metrics.tmHeight + Metrics.tmInternalLeading;
end;

procedure TddgMarquee.SetStartLine;
{ Этот виртуальный метод инициализирует переменную CurrLine экземпляра }
begin
```

```
  { Инициализация текущей строки на начало списка, если прокрутка вверх...}
  if not FScrollDown then CurrLine := 0
  { ...или на конец списка, если прокрутка вниз }
  else CurrLine := VRect.Bottom - Height;
end;

procedure TddgMarquee.PaintLine(R: TRect; LineNum: Integer);
{Этот метод вызывается для помещения каждой строки текста в объект MemBitmap }
const
  Flags: array[TJustification] of DWORD = (DT_CENTER, DT_LEFT, DT_RIGHT);
var
  S: string;
begin
  { Для упрощения скопируем строку в локальную переменную }
  S := FItems.Strings[LineNum];
  { Помещаем строку текста в растровое изображение в памяти }
  DrawText(MemBitmap.Canvas.Handle, PChar(S), Length(S), R, Flags[FJust] or
DT_SINGLELINE or DT_TOP);
end;

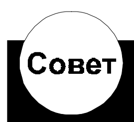
procedure TddgMarquee.FillBitmap;
var
  y, i : Integer;
  R: TRect;
begin
  SetLineHeight;               // Установка высоты каждой строки
  { Прямоугольник VRect представляет всю память растрового изображения }
  VRect := Rect(0, 0, Width, LineHi * FItems.Count + Height * 2);
  { Прямоугольник InsideRect представляет внутреннюю
    часть прямоугольной области }
  InsideRect := Rect(BevelWidth, BevelWidth, Width - (2 * BevelWidth),
    Height - (2 * BevelWidth));
  R := Rect(InsideRect.Left, 0, InsideRect.Right, VRect.Bottom);
  SetStartLine;
  MemBitmap.Width := Width;   // Инициализируем память растрового изображения
  with MemBitmap do
  begin
    Height := VRect.Bottom;
    with Canvas do
    begin
      Font := Self.Font;
      Brush.Color := Color;
      FillRect(VRect);
      Brush.Style := bsClear;
    end;
  end;
  y := Height;
  i := 0;
  repeat
    R.Top := y;
```

```
    PaintLine(R, i);
    { Инкрементируем y на высоту (в пикселях) строки }
    inc(y, LineHi);
    inc(i);
  until i >= FItems.Count;      // То же самое повторяем для всех строк
end;

procedure TddgMarquee.Paint;
{Этот виртуальный метод вызывается в ответ на сообщение Paint от Windows }
begin
  if FActive then
    { Копирование содержимого изображения из памяти на экран }
    BitBlt(Canvas.Handle, 0, 0, InsideRect.Right, InsideRect.Bottom,
      MemBitmap.Canvas.Handle, 0, CurrLine, srcCopy)
  else
    inherited Paint;
end;

procedure TddgMarquee.SetActive(Value: Boolean);
{ Вызывается для активизации/деактивизации прокрутки титров }
begin
  if Value and (not FActive) and (FItems.Count > 0) then
  begin
    FActive := True;         // Установка флага активности
    MemBitmap := TBitmap.Create;
    FillBitmap;              // Помещает Image в растровое изображение
    FTimer.Enabled := True;  // Запуск таймера
  end
  else if (not Value) and FActive then
  begin
    FTimer.Enabled := False; // Отключение таймера
    if Assigned(FOnDone)     // Генерация события OnDone
      then FOnDone(Self);
    FActive := False;        // Установка переменной FActive равной False
    MemBitmap.Free;          // Освобождение изображения в памяти
    Invalidate;              // Очистка окна элемента управления
  end;
end;

end.
```



Обратите внимание на использование директивы `default` со свойством `Justify` компонента `TddgMarquee`. Использование директивы `default` оптимизирует работу компонента с потоками данных, что повышает производительность времени разработки. Вы можете создавать значения по умолчанию для свойств любого упорядоченного типа (`Integer`, `Word`, `Longint`, а также, например, для перечислимых типов), но для свойств неупорядоченных типов, таких как строки, числа с плавающей точкой, массивы, записи и классы, значения по умолчанию не могут быть созданы.
Вы должны инициализировать ваши значения по умолчанию для свойств в конструкторе. В противном случае возможны проблемы при работе с потоками данных.

Тестирование компонента TddgMarquee

Хотя компонент уже практически написан и находится в стадии тестирования, охладите ваш пыл и пока не устанавливайте его в палитру компонентов. Сначала нужно его отладить. Необходимо выполнить полное предварительное тестирование компонента, разработав проект, создающий и использующий динамический экземпляр нового компонента. В листинге 22.3 содержится текст главного модуля проекта `TestMarq`, предназначенного для тестирования компонента `TddgMarquee`. Этот простой проект состоит из формы с двумя кнопками.

Листинг 22.3. Исходный текст модуля `TestU.pas`, предназначенного для тестирования компонента `TddgMarquee`

```
unit Testu;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Marquee, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    Marquee1: TddgMarquee;
    procedure MDone(Sender: TObject);
  public
    { Открытые объявления }
end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.MDone(Sender: TObject);
begin
  Beep;
end;

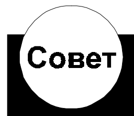
procedure TForm1.FormCreate(Sender: TObject);
begin
  Marquee1 := TddgMarquee.Create(Self);
  with Marquee1 do
```

```
  begin
    Parent := Self;
    Top := 10;
    Left := 10;
    Height := 200;
    Width := 150;
    OnDone := MDone;
    Show;
    with Items do
 begin
      Add('Greg');
      Add('Peter');
      Add('Bobby');
      Add('Marsha');
      Add('Jan');
      Add('Cindy');
    end;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Marquee1.Active := True;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Marquee1.Active := False;
end;

end.
```



Обязательно создавайте тестовые проекты для всех новых компонентов. Никогда не пытайтесь выполнить пробное тестирование компонента добавлением его в палитру компонентов. Занимаясь отладкой компонента, находящегося в палитре компонентов, вы не только потеряете много времени на бесполезную перекомпоновку пакета, но и, возможно, столкнетесь с зависанием интегрированной среды Delphi из-за ошибки в вашем новом компоненте.

На рис. 22.2 показан проект TestMarq в действии.

Только после устранения всех ошибок в новом компоненте, его можно добавить в палитру компонентов. Как вы помните, это довольно просто: нужно выбрать команду главного меню Component⇨Install Component, а затем ввести имена модуля и пакета в раскрывшемся диалоговом окне Install Component. Щелкните на кнопке OK — и Delphi перекомпонует тот пакет, который помещается новый компонент, и обновит вид палитры компонентов. Конечно, для того чтобы новой компонент мог оказаться в палитре компонентов, ему потребуется процедура Register(). Компонент TddgMarquee регистрируется в модуле DDGReg.pas пакета DDGDsgn, который находится на прилагаемом компакт-диске.

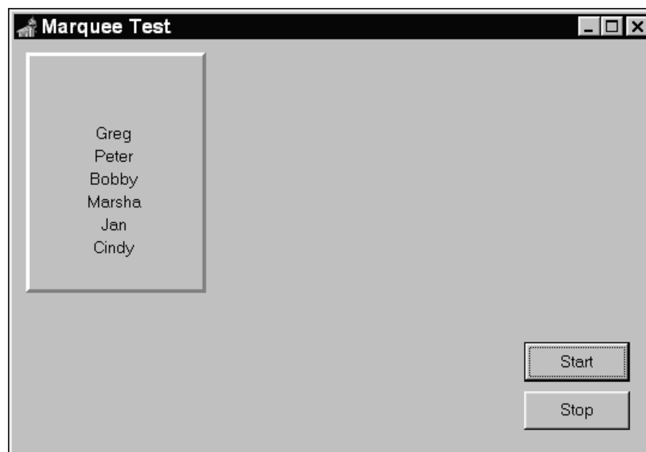


Рис. 22.2. Тестирование компонента `TddgMarquee`

Создание редакторов свойств

В главе 21, “Создание пользовательских компонентов в Delphi”, показано, как в окне инспектора объектов выполняется редактирование свойств самых распространенных типов. Средство, с помощью которого выполняется редактирование свойства, называется *редактором свойства*. Для уже существующих свойств предусмотрено несколько заранее созданных редакторов. Но возможна ситуация, когда ни один из уже существующих редакторов применить не удастся — например, при создании пользовательского свойства. В этом случае потребуется разработать собственный редактор свойства.

Редактировать свойства в окне инспектора объектов можно двумя способами. Один заключается в предоставлении пользователю возможности редактирования свойства как строки текста. Другой требует создания специального диалогового окна, в котором и выполняется редактирование свойства. В некоторых случаях вам потребуется использовать оба способа для редактирования одного свойства.

Перечислим основные этапы создания редактора свойства.

1. Создание объекта-потомка редактора свойств.
2. Редактирование свойства как текста.
3. Редактирование свойства в диалоговом окне (необязательный этап).
4. Определение атрибутов редактора свойств.
5. Регистрация редактора свойств.

Все эти этапы подробно рассматриваются в следующих разделах.

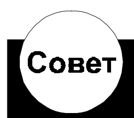
Создание объекта-потомка редактора свойств

Delphi определяет несколько редакторов свойств в модуле `DsgnIntf.pas`, причем все они происходят от базового класса `TPropertyEditor`. Создаваемый вами редактор свойства также должен происходить от него или его потомков. В табл. 22.1 перечислены потомки класса `TPropertyEditor`, используемые для редактирования свойств уже существующих типов.

Таблица 22.1. Редакторы свойств, определенные в модуле DsgnIntf.pas

Редакторы свойств	Описание
`TOrdinalProperty`	Базовый класс для всех редакторов свойств порядковых типов, таких как `TIntegerProperty`, `TEnumProperty`, `TCharProperty` и т.п.
`TIntegerProperty`	Редактор целочисленных свойств произвольных размеров
`TCharProperty`	Редактор свойств типа `char` и `char`-поддиапазонов (например, 'A'..'Z')
`TEnumProperty`	Редактор свойств для всех перечислимых типов, определенных пользователем
`TFloatProperty`	Редактор свойств вещественного типа (чисел с плавающей точкой)
`TStringProperty`	Редактор свойств строковых типов
`TSetElementProperty`	Редактор свойств, являющихся отдельными элементами множества. Каждый элемент рассматривается как независимый булев элемент
`TSetProperty`	Редактор свойств-множеств. Множество редактируется путем редактирования его отдельных элементов
`TClassProperty`	Редактор свойств, представляющих собой объекты
`TMethodProperty`	Редактор свойств, являющихся указателями на методы, т.е. *событиями*
`TComponentProperty`	Редактор свойств, ссылающихся на компоненты. Это не то же самое, что редактор `TClassProperty`. Этот редактор позволяет пользователю задать компонент, на который ссылается свойство, т.е. `ActiveControl`
`TColorProperty`	Редактор свойств типа `Tcolor` (цвет)
`TFontNameProperty`	Редактор свойств, содержащих имя шрифта. Этот редактор отображает раскрывающийся список названий шрифтов, доступных в данной системе
`TFontProperty`	Редактор свойств типа `TFont`, позволяющий редактировать подсвойства, поскольку он является производным от класса `TClassProperty`

Выбор предка для создаваемого редактора свойства основывается на желаемом поведении свойства при его редактировании. Например, ваше свойство может обладать той же функциональностью, которая присуща компоненту `TIntegerProperty`, но при этом требует дополнительной логики редактирования. Следовательно, в качестве предка для вашего редактора свойства целесообразно использовать класс `TIntegerProperty`.



Иногда нет никакой необходимости в создании редактора свойства, специфического для типа вашего свойства. Например, границы типов-поддиапазонов автоматически контролируются соответствующим редактором (скажем, значения `1..10` находятся под контролем компонента `TIntegerProperty`), для перечислимых типов автоматически используются раскрывающиеся списки и т.д. Лучше применять определения типов, а не пользовательские редакторы свойств, так как они отлично поддерживаются и языком программирования во время компиляции, и используемыми по умолчанию редакторами свойств.

Редактирование свойства в виде текста

Редактор свойства используется для двух основных целей. Первая достаточно ясна и состоит в том, чтобы предоставить пользователю возможность редактировать свойства. Другая, не столь очевидна, — обеспечить строковое представление значения свойства в окне инспектора объектов.

При создании класса-потомка редактора свойства необходимо переопределить методы GetValue() и SetValue(). Метод GetValue() возвращает строковое представление значения свойства для отображения в окне инспектора объектов. Метод SetValue(), основываясь на подобном представлении значения свойства, присваивает это значение для отображения нового значения свойства в окне инспектора объектов.

В качестве примера рассмотрим определение класса TIntegerProperty в модуле DsgnInfo.pas:

```
TIntegerProperty = class(TOrdinalProperty)
public
  function GetValue: string; override;
  procedure SetValue(const Value: string); override;
end;
```

Как видите, методы GetValue() и SetValue() переопределены. Вот реализация метода GetValue():

```
function TIntegerProperty.GetValue: string;
begin
  Result := IntToStr(GetOrdValue);
end;
```

А вот реализация метода SetValue():

```
procedure TIntegerProperty.SetValue(const Value: String);
var
  L: Longint;
begin
  L := StrToInt(Value);
  with GetTypeData(GetPropType)^ do
    if (L < MinValue) or (L > MaxValue) then
      raise EPropertyError.CreateResFmt(SOutOfRange, [MinValue, MaxValue]);
  SetOrdValue(L);
end;
```

Метод GetValue() возвращает строковое представление целого свойства. Инспектор объектов использует эту строку для отображения значения свойства. Метод GetOrdValue(), определенный в классе TPropertyEditor, служит для получения значения свойства, с которым работает редактор.

Метод SetValue() принимает строковое значение, введенное пользователем, и присваивает его свойству в нужном формате. Метод SetValue() выполняет контроль ошибок выхода за пределы диапазона, определенного для данного свойства. Это полезный пример реализации контроля ошибок в редакторах-потомках. Метод SetOrdValue() присваивает свойству, с которым работает редактор, введенное пользователем значение.

В классе TPropertyEditor определено несколько методов, подобных методу GetOrdValue(), предназначенных для получения строкового представления различных типов. К тому же, компонент TPropertyEditor содержит аналогичные Set-методы для установки значений в соответствующих форматах. Потомки класса TPropertyEditor наследуют все эти методы, используемые для получения и установки значений свойств, с которыми работают редакторы свойств. Эти методы приведены в табл. 22.2.

Таблица 22.2. Методы чтения/записи свойств, определенные в классе TPropertyEditor

Тип свойства	Метод Get	Метод Set
Число с плавающей точкой	GetFloatValue()	SetFloatValue()
Событие	GetMethodValue()	SetMethodValue()
Порядковый тип	GetOrdValue()	SetOrdValue()
Строковый тип	GetStrValue()	SetStrValue()
Вариантный	GetVarValue()	SetVarValue(),SetVarValueAt()

Для иллюстрации создания нового редактора свойства мы вернемся к примеру с планетами Солнечной системы, приведенному в предыдущей главе. Прошлый раз был создан простой компонент TPlanet, представляющий одну планету. Компонент TPlanet содержит свойство PlanetName. Его внутреннее поле типа Integer хранит позицию планеты в Солнечной системе, но в окне инспектора объектов оно должно будет отображаться как название планеты.

Это совсем просто, поэтому мы добавим одно усложнение. Допустим, требуется предоставить пользователю возможность выбрать один из двух способов задания планеты. Он может набрать строку названия планеты, например Венера или ВЕНЕРА, или ВеНеРа, или ввести позицию планеты в Солнечной системе. Так, для Венеры эта позиция будет равной значению 2.

Ниже приводится исходный код компонента TPlanet:

```
type
  TPlanetName = type Integer;

  TPlanet = class(TComponent)
  private
    FPlanetName: TPlanetName;
  published
    property PlanetName: TPlanetName read FPlanetName write FPlanetName;
  end;
```

Как видите, этот компонент совсем небольшой. У него есть только одно свойство PlanetName типа TPlanetName. Специальное определение типа TPlanetName позволяет ему иметь собственную информацию о типах времени выполнения и оставаться при этом целочисленным типом.

Основная функциональность находится не в компоненте TPlanet, а в редакторе его свойства типа TPlanetName. Текст этого редактора приведен в листинге 22.4.

Листинг 22.4. Исходный текст редактора TPlanetNameProperty

```
unit PlanetPE;

interface

uses
  Windows, SysUtils, DsgnIntF;
type
  TPlanetNameProperty = class(TIntegerProperty)
  public
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
```

```
  end;

implementation

const
  { Объявление массива-константы с названиями планет }
  PlanetNames: array[1..9] of String[8] =
   ('Меркурий', 'Венера', 'Земля', 'Марс', 'Юпитер', 'Сатурн',
    'Уран', 'Нептун', 'Плутон');

function TPlanetNameProperty.GetValue: string;
begin
  Result := PlanetNames[GetOrdValue];
end;

procedure TPlanetNameProperty.SetValue(const Value: String);
var
  PName: string[8];
  i, ValErr: Integer;
begin
  PName := UpperCase(Value);
  i := 1;
  { Сравнение Value с названием каждой планеты в массиве PlanetNames.
    Если найдено соответствие, переменная i принимает значение, меньшее 10.}
  while (PName <> UpperCase(PlanetNames[i])) and (i < 10) do
    inc(i);
  { Если i меньше 10, введено правильное название планеты.
    Установка значения и выход из процедуры. }
  if i < 10 then  // Название планеты введено правильно
  begin
    SetOrdValue(i);
    Exit;
  end
  { Если i больше 10, пользователь ввел недопустимый номер планеты или
    несуществующее название планеты. Используйте функцию Val для проверки
    введенного значения на число. Если ValErr не равен нулю, название
    планеты введено неправильно. В противном случае проверьте введенный
    номер на вмещение в диапазон (0 < i < 10). }
  else begin
    Val(Value, i, ValErr);
    if ValErr <> 0 then
      raise Exception.Create(Format('Извините, мы никогда не'+
           ' слышали о планете %s.', [Value]));
    if (i <= 0) or (i >= 10) then
       raise Exception.Create('Извините, но в НАШЕЙ Солнечной системе'+
          ' такой планеты нет');
    SetOrdValue(i);
  end;
end;

end.
```

Вначале создается редактор свойства `TPlanetNameProperty`, являющийся потомком компонента `TIntegerProperty`. Кстати, в раздел `uses` нашего модуля обязательно нужно включить модуль `DsgnIntf`.

Мы определили массив строковых констант для представления планет Солнечной системы, в зависимости от их положения относительно Солнца. Эти строки будут отвечать за строковое представление планет в окне инспектора объектов.

Как уже отмечалось, мы должны переопределить методы `GetValue()` и `SetValue()`. Метод `GetValue()` возвращает строку из массива `PlanetNames`. Этот массив индексируется значением свойства. Конечно же, значение свойства должно находиться в пределах диапазона 1–9. Мы добиваемся этого, запрещая пользователю вводить в метод `SetValue()` номер, выходящий за пределы этого диапазона.

Метод `SetValue()` получает строку, введенную в окне инспектора объектов. Эта строка может быть как названием планеты, так и номером, определяющим позицию планеты. Логика кода определяет, правильно ли введено название планеты или ее номер, и если да, то соответствующее значение присваивается свойству методом `SetOrdValue()`. Если пользователь ввел неправильное название планеты или недопустимый номер, генерируется соответствующее исключение.

Вот и все определение редактора свойства. Впрочем, нет, еще не все — вы должны его зарегистрировать для того, чтобы свойство "знало" о своем новом редакторе.

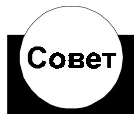
Регистрация нового редактора свойства

Для регистрации редактора свойства следует воспользоваться процедурой `RegisterPropertyEditor()`. Этот метод объявляется следующим образом:

```
procedure RegisterPropertyEditor(PropertyType: PTypeInfo;
   ComponentClass: TClass; const PropertyName: string;
   EditorClass: TPropertyEditorClass);
```

Первый параметр `PropertyType` — это указатель на информацию о типах времени выполнения редактируемого свойства. Данную информацию можно получить с помощью функции `TypeInfo()`. Параметр `ComponentClass` используется для обозначения класса, к которому применяется редактор свойства. `PropertyName` определяет имя свойства компонента, а параметр `EditorClass` — тип используемого редактора свойства. Для свойства `TPlanet.PlanetName` эта функция выглядит следующим образом:

```
RegisterPropertyEditor(TypeInfo(TPlanetName), TPlanet, 'PlanetName',
TPlanetNameProperty);
```



Пока (в иллюстративных целях) этот конкретный редактор свойств регистрируется только для использования со свойством `PlanetName` компонента `TPlanet`. Однако вы можете зарегистрировать этот редактор таким образом, чтобы он был пригоден для работы с любым свойством типа `TPlanetName`. Для этого следует в качестве параметра `ComponentClass` ввести значение `nil`, а в качестве параметра `PropertyName` — `''`.

Регистрацию редактора свойств можно оформить вместе с регистрацией компонента в модуле компонента, как показано в листинге 22.5.

Листинг 22.5. Модуль `Planet.pas`, содержащий компонент `TPlanet`

```
unit Planet;

interface

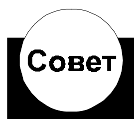
uses
  Classes, SysUtils;

type
  TPlanetName = type Integer;

  TPlanet = class(TComponent)
  private
    FPlanetName: TPlanetName;
  published
    property PlanetName: TPlanetName read FPlanetName write FPlanetName;
  end;

implementation

end.
```



Регистрация редактора свойств в процедуре `Register()` модуля компонента связывает код редактора с компонентом при помещении последнего в пакет. Для сложных компонентов средства времени разработки зачастую занимают больше места, чем код самих компонентов. Несмотря на то что размер кода не играет роли для таких маленьких компонентов, как этот, имейте в виду, что все перечисленное в интерфейсном разделе модуля вашего компонента (в том числе и процедура `Register()` вместе со всеми классами, ею регистрируемыми, такими как тип класса редактора свойств) будет скомпилировано в пакет вместе с компонентом. Поэтому регистрацию вашего редактора свойств нужно осуществлять в отдельном регистрационном модуле. Некоторые разработчики компонентов создают для своих компонентов как пакеты разработки, так и пакеты времени выполнения, причем редакторы свойств и другие средства времени разработки присутствуют только в пакете разработки. Программный код, представляет собой примеры, обсуждаемые в данной книге, содержится в пакете времени выполнения `DdgStd5` и пакете разработки `DdgDsgn5`.

Редактирование свойства в специальном диалоговом окне

Иногда требуются более широкие возможности редактирования, чем редактирование на месте в окне инспектора объектов. В этом случае в качестве редактора свойств можно использовать диалоговое окно). Например, большинство компонентов Delphi имеют свойство `Font`. Конечно, создатели Delphi могли заставить пользователей вручную набирать имя шрифта и сопутствующую информацию. Но было бы неразумно ожидать, что пользователь знает, какую информацию ему следует вводить. Гораздо проще обеспечить

пользователя диалоговым окном, в котором он сможет установить необходимые атрибуты, относящиеся к шрифту, и увидеть результат их применения до выбора окончательного варианта.

Чтобы проиллюстрировать использование диалогового окна для редактирования свойства, мы собираемся расширить функциональность компонента TddgRunButton, создание которого описано в главе 21, "Создание пользовательских компонентов в Delphi". При установке значения свойства CommandLine в окне инспектора объектов пользователю будет достаточно щелкнуть на кнопке с многоточием, чтобы открыть диалоговое окно Open File и выбрать в нем файл для работы с компонентом TddgRunButton.

Пример диалогового редактора свойств: расширение возможностей компонента TddgRunButton

Текст компонента TddgRunButton приведен в главе 21, "Создание пользовательских компонентов в Delphi", в листинге 21.13. Мы не будем полностью повторять его здесь, но хотим обратить ваше внимание на несколько моментов. Свойство TddgRunButton.CommandLine определено, как имеющее тип TCommandLine. Этот тип определяется следующим образом:

```
TCommandLine = type string;
```

И вновь заметим, специальное объявление типа используется для того, чтобы иметь уникальную информацию о типах времени выполнения. Это позволяет определить редактор свойств специально для типа TCommandLine. Кроме того, поскольку тип TCommandLine аналогичен строковому, для типа TCommandLine применим редактор свойств, предназначенный для строковых типов.

Описывая редактор свойств типа TCommandLine, не забывайте, что проверка наличия ошибок присвоения свойству уже включена в его методы доступа. Следовательно, нет необходимости повторять ее в логике редактора свойств.

В листинге 22.6 приведен текст редактора свойств TCommandLineProperty.

Листинг 22.6. Модуль RunBtnPE.pas: редактор свойств TCommandLineProperty

```
unit runbtnpe;

interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons, DsgnIntF, TypInfo;

type

  { Редактор происходит от класса TStringProperty и унаследовал
    его возможности для редактирования строк. }
  TCommandLineProperty = class(TStringProperty)
    function GetAttributes: TPropertyAttributes; override;
    procedure Edit; override;
  end;
```

```
implementation

function TCommandLineProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paDialog]; // Отображение диалогового окна, вызываемого методом Edit
end;

procedure TCommandLineProperty.Edit;
{ Метод Edit отображает окно TOpenDialog, из которого пользователь получает имя
выполняемого файла, присваиваемое свойству. }
var
  OpenDialog: TOpenDialog;
begin
  { Создаем окно TOpenDialog }
  OpenDialog := TOpenDialog.Create(Application);
  try
    { Отображаем только выполняемые файлы }
    OpenDialog.Filter := 'Executable Files|*.EXE';
    { Если пользователь выбрал файл, присвоить его свойству. }
    if OpenDialog.Execute then
      SetStrValue(OpenDialog.FileName);
  finally
    OpenDialog.Free // Освобождаем экземпляр класса TOpenDialog.
  end;
end;

end.
```

Ознакомившись с компонентом TCommandLineProperty, вы наверняка поняли, что этот редактор свойств очень прост. Он происходит от класса TStringProperty и, следовательно, поддерживает средства редактирования строк. Поэтому необязательно вызывать диалоговое окно в инспекторе объектов. Пользователь может вручную набрать командную строку. К тому же, мы не переопределяли методы SetValue() и GetValue(), так как в компоненте TStringProperty уже это сделано надлежащим образом. Тем не менее необходимо переопределить метод GetAttributes() для того, чтобы инспектор свойств "знал", что свойство можно редактировать с помощью диалогового окна. Метод GetAttributes() заслуживает более подробного обсуждения.

Задание атрибутов редактора свойств

Каждый редактор свойств должен сообщить инспектору объектов о способе редактирования свойства и об использовании специальных атрибутов, если таковые имеются. В большинстве случаев достаточно атрибутов, унаследованных от редактора-потомка. Однако в некоторых случаях вам придется переопределить метод TPropertyEditor.GetAttributes(), возвращающий множество флагов атрибутов свойства (компонент TPropertyAttributes). Это множество указывает, для каких атрибутов используются специальные редакторы свойств. Различные флаги типа TPropertyAttribute приведены в табл. 22.3.

Таблица 22.3. Флаги типа TPropertyAttribute

Атрибут	Как редактор свойств работает с инспектором объектов
paValueList	Возвращает перечислимый список значений свойства. Метод GetValue() заполняет список. Справа от значения свойства находится кнопка с изображением стрелки, направленной вниз. Применяется для таких перечислимых свойств, как TForm.BorderStyle, и таких групп целочисленных констант, как TColor и TCharSet
paSubProperties	Подсвойства отображаются в контурном формате с отступом вниз от текущего свойства. Атрибут paValueList также должен быть установлен. Устанавливается для свойств типа "множество" или типа "класс", таких как TOpenDialog.Options и TForm.Font
paDialog	Появляется кнопка с многоточием, по щелчку на которой метод Edit() вызывает диалоговое окно. Применяется для свойств наподобие TForm.Font
paMultiSelect	Свойства отображаются, даже если в конструкторе форм выбрано больше одного компонента; это позволяет пользователю менять значения свойств сразу нескольких компонентов. Некоторые свойства, например Name, не подходят для использования этой возможности
paAutoUpdate	Метод SetValue() вызывается при каждом изменении свойства. Если этот флаг не установлен, SetValue() вызывается при нажатии клавиши <Enter> или при выходе из свойства в окне инспектора объектов. Применяется для таких свойств, как Tform.Caption
paFullWidthName	Сообщает инспектору объектов о том, что это значение не нуждается в дополнительном представлении, а посему его имя следует отображать, используя всю ширину окна инспектора объектов
paSortList	Инспектор объектов сортирует список, возвращаемый функцией GetValues()
paReadOnly	Значение свойства не может быть изменено
paRevertable	Свойству можно восстановить его прежнее значение. Некоторые свойства, например составные (типа TFont), не нужно восстанавливать

Установка атрибута paDialog для свойства типа TCommandLineProperty

Поскольку компонент TCommandLineProperty используется для отображения диалогового окна, необходимо сообщить инспектору объектов об этой возможности, установив атрибут paDialog в методе TCommandLineProperty.GetAttributes(). В этом случае справа от значения свойства CommandLine в окне инспектора объектов появится кнопка с многоточием. Если пользователь щелкнет на этой кнопке, будет вызван метод TCommandLineProperty.Edit().

Регистрация редактора свойств TCommandLineProperty

Завершающим этапом реализации редактора свойств TCommandLineProperty является его регистрация с помощью описанной выше в этой главе процедуры RegisterPropertyEditor(). Эта процедура была добавлена в процедуру Register() в модуле DDGReg.pas, входящем в состав пакета DDGDsgn:

```
RegisterComponents('DDG', [TddgRunButton]);
  RegisterPropertyEditor(TypeInfo(TCommandLine), TddgRunButton, '',
TCommandLineProperty);
```

Помните о том, что в раздел uses необходимо добавить модули DsgnIntf и RunBtnPE.

Редакторы компонентов

Редакторы компонентов расширяют поведение компонентов во время разработки, позволяя добавлять элементы в контекстное меню, связанное с конкретным компонентом, а также изменять стандартные действия, выполняемые по двойному щелчку мышью на компоненте в конструкторе форм. Вполне возможно, что вы уже использовали редакторы полей компонентов TTable, TQuery и TStoredProc, и, следовательно, сами того не ведая, работали с редакторами компонентов.

Компонент TComponentEditor

Быть может, вам не известно, что для каждого компонента, выбранного в конструкторе форм, создается особый редактор. Тип созданного редактора компонентов зависит от типа компонентов, хотя все редакторы компонентов происходят от класса TComponentEditor. Этот класс определен в модуле DsgnIntf следующим образом:

```
type
  TComponentEditor = class(TInterfacedObject, IComponentEditor)
  private
    FComponent: TComponent;
    FDesigner: IFormDesigner;
  public
    constructor Create(AComponent: TComponent; ADesigner: IFormDesigner);
      virtual;
    procedure Edit; virtual;
    procedure ExecuteVerb(Index: Integer); virtual;
    function GetIComponent: IComponent;
    function GetDesigner: IFormDesigner;
    function GetVerb(Index: Integer): string; virtual;
    function GetVerbCount: Integer; virtual;
    procedure Copy; virtual;
    property Component: TComponent read FComponent;
    property Designer: IFormDesigner read GetDesigner;
  end;
```

Свойства

Свойство TComponentEditor.Component — это экземпляр редактируемого компонента. Так как это свойство имеет общий тип TComponent, для доступа к полям, введенным классами-потомками, необходимо выполнить преобразование типов.

Свойство Designer — это экземпляр компонента TFormDesigner, управляющий приложением во время разработки. Полное определение этого класса можно найти в модуле DsgnIntf.pas.

Методы

Метод Edit() вызывается в результате двойного щелчка на компоненте во время разработки. Чаще всего этот метод вызывает тот или иной тип диалогового окна разработки. По умолчанию алгоритм этого метода предполагает вызов функции ExecuteVerb(0), если функция GetVerbCount() возвращает значение, большее или равное 1. Если в методе Edit() или в любом другом методе компонент подвергается изменениям, необходимо вызвать метод Designer.Modified().

Метод GetVerbCount() вызывается для получения количества элементов, добавляемых в контекстное меню.

Метод GetVerb() принимает целый параметр Index и возвращает строку, содержащую текст, появляющийся в соответствующей позиции контекстного меню.

После выбора элемента в контекстном меню вызывается метод ExecuteVerb(). Этому методу в параметре Index передается номер выбранного пункта контекстного меню (считая от нуля). В ответ должно быть выполнено действие, определяемое выбранным пунктом меню.

Метод Paste() вызывается при помещении компонента в буфер обмена. Delphi помещает в буфер обмена ресурсный (записанный в поток) образ компонента. Этот метод можно также использовать для помещения в буфер обмена данных различного формата.

Стандартный редактор компонентов TDefaultEditor

Если для определенного компонента собственный редактор компонентов не зарегистрирован, для его редактирования будет использоваться стандартный редактор, имеющий класс TDefaultEditor. Этот редактор переопределяет поведение метода Edit() так, что он ищет свойства компонента и генерирует (или находит) событие OnCreate, OnChanged или OnClick (в зависимости от того, какое из них будет найдено первым).

Пример простого компонента

Рассмотрим следующий простой пользовательский компонент:

```
type
  TComponentEditorSample = class(TComponent)
  protected
    procedure SayHello; virtual;
    procedure SayGoodbye; virtual;
  end;

procedure TComponentEditorSample.SayHello;
begin
  MessageDlg('Привет!', mtInformation, [mbOk], 0);
end;

procedure TComponentEditorSample.SayGoodbye;
```

```
begin
  MessageDlg('Пока!', mtInformation, [mbOk], 0);
end;
```

Как видите, этот небольшой компонент может делать немногое. Это невизуальный компонент, происходящий непосредственно от класса `TComponent` и содержащий два метода — `SayHello()` и `SayGoodbye()`, — просто отображающих окна сообщений.

Пример редактора для простого компонента

Для того чтобы компонент смотрелся солиднее, его можно снабдить редактором. Этот редактор будет вызываться во время разработки для выполнения методов компонента. Для этого в нем придется переопределить, по крайней мере, три метода компонента `TComponentEditor`: методы `ExecuteVerb()`, `GetVerb()` и `GetVerbCount()`. Приведем исходный код этого компонента:

```
type
  TSampleEditor = class(TComponentEditor)
  private
    procedure ExecuteVerb(Index: Integer); override;
    function GetVerb(Index: Integer): string; override;
    function GetVerbCount: Integer; override;
  end;

procedure TSampleEditor.ExecuteVerb(Index: Integer);
begin
  case Index of
    0: TComponentEditorSample(Component).SayHello;    // Вызов функции
    1: TComponentEditorSample(Component).SayGoodbye;  // Вызов функции
  end;
end;

function TSampleEditor.GetVerb(Index: Integer): string;
begin
  case Index of
    0: Result := 'Привет';  // Возвращает строку приветствия
    1: Result := 'Пока';    // Возвращает строку прощания
  end;
end;

function TSampleEditor.GetVerbCount: Integer;
begin
  Result := 2;      // Существует две возможные команды
end;
```

Метод `GetVerbCount()` возвращает число 2, а это означает, что редактор компонентов готов выполнить две команды. `GetVerb()` возвращает строку каждой из команд, появляющихся в контекстном меню. Метод `ExecuteVerb()` вызывает соответствующий метод компонента, основываясь на передаваемом ему в качестве параметра значении индекса.

Регистрация редактора компонентов

Подобно компонентам и редакторам свойств, редакторы компонентов также должны быть зарегистрированы в интегрированной среде Delphi — в методе `Register()` модуля. Для регистрации редактора компонентов вызывается процедура с соответствующим именем — `RegisterComponentEditor()`, — объявленная следующим образом:

```
procedure RegisterComponentEditor(ComponentClass: TComponentClass;
  ComponentEditor: TComponentEditorClass);
```

Первый параметр данной функции определяет тип компонента, для которого вы хотите зарегистрировать редактор; второй параметр задает имя этого редактора.

В листинге 22.7 приведен код модуля `CompEdit.pas`, который включает компонент, его редактор и регистрационный вызов. На рис. 22.3 показано локальное меню, связанное с компонентом `TComponentEditorSample`, а на рис. 22.4 — результат выбора команды в этом меню.

Листинг 22.7. Модуль `CompEdit.pas` — редактор компонентов `TComponentEditorSample`

```
unit CompEdit;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls, Forms, Dialogs,
  DsgnIntf;

type
  TComponentEditorSample = class(TComponent)
  protected
    procedure SayHello; virtual;
    procedure SayGoodbye; virtual;
  end;

  TSampleEditor = class(TComponentEditor)
  private
    procedure ExecuteVerb(Index: Integer); override;
    function GetVerb(Index: Integer): string; override;
    function GetVerbCount: Integer; override;
  end;

implementation

{ Компонент TComponentEditorSample }

procedure TComponentEditorSample.SayHello;
begin
  MessageDlg('Hello, there!', mtInformation, [mbOk], 0);
end;

procedure TComponentEditorSample.SayGoodbye;
begin
```

```
  MessageDlg('See ya!', mtInformation, [mbOk], 0);
end;

{ Редактор TSampleEditor }

const
  vHello = 'Hello';
  vGoodbye = 'Goodbye';

procedure TSampleEditor.ExecuteVerb(Index: Integer);
begin
  case Index of
    0: TComponentEditorSample(Component).SayHello;    // Вызов функции
    1: TComponentEditorSample(Component).SayGoodbye;  // Вызов функции
  end;
end;

function TSampleEditor.GetVerb(Index: Integer): string;
begin
  case Index of
    0: Result := vHello;    // Возвращает строку приветствия
    1: Result := vGoodbye;  // Возвращает строку прощания
  end;
end;

function TSampleEditor.GetVerbCount: Integer;
begin
  Result := 2;      // Две возможные команды
end;

end.
```

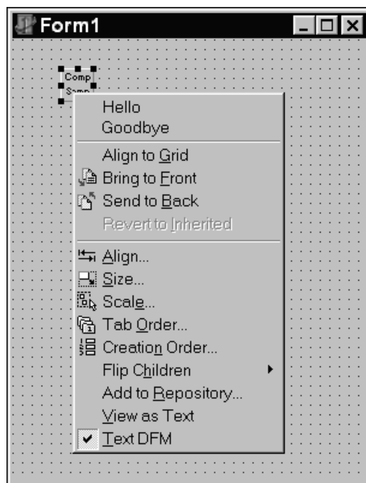


Рис. 22.3. Контекстное меню компонента `TcomponentEditorSample`

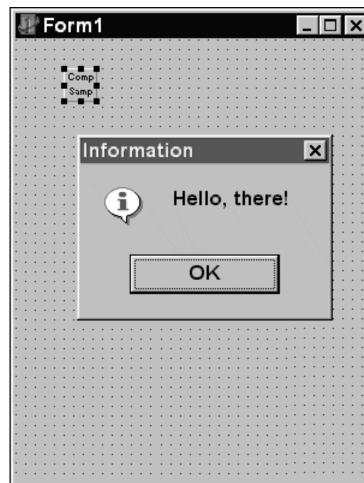


Рис. 22.4. Результат выполнения команды

Работа непубликуемых компонентов с потоками данных

В главе 21, “Создание пользовательских компонентов в Delphi”, отмечалось, что интегрированная среда Delphi автоматически записывает и считывает публикуемые (published) свойства компонента из DFM-файла. Что же делать, если в DFM-файле требуется сохранять и непубликуемые данные? К счастью, компоненты Delphi содержат механизм, позволяющий записывать и считывать определенные программистом данные из DFM-файла.

Определение свойств

Первым шагом определения сохраняемых непубликуемых свойств является переопределение метода `DefineProperties()` компонента. Этот метод любой компонент наследует от класса `TPersistent`, в котором он определяется следующим образом:

```
procedure DefineProperties(Filer: TFiler); virtual;
```

По умолчанию этот метод управляет чтением и записью публикуемых свойств в DFM-файл. Можно переопределить этот метод и после вызова унаследованного метода вызвать определенные в компоненте `TFiler` методы `DefineProperty()` или `DefineBinaryProperty()` — один раз для каждой порции данных, которые нужно поместить в DFM-файл. Определение этих методов приводится ниже.

```
procedure DefineProperty(const Name: string; ReadData: TReaderProc;
    WriteData: TWriterProc; HasData: Boolean); virtual;

procedure DefineBinaryProperty(const Name: string; ReadData, WriteData:
    TStreamProc; HasData: Boolean); virtual;
```

Функция `DefineProperty()` используется для того, чтобы сделать сохраняемыми такие стандартные типы данных, как строковые, целые, булевы, символьные типы, числа с плавающей точкой и перечислимые типы. Метод `DefineBinaryProperty()` используется для обеспечения доступа к необработанным двоичным данным (графическим или звуковым), записанным в DFM-файл.

В обеих этих функциях параметр `Name` идентифицирует имя свойства, которое должно быть записано в DFM-файл. Оно может не совпадать с внутренним именем поля, содержащего наши данные. Параметры `ReadData` и `WriteData` функции `DefineProperty()` отличаются от соответствующих параметров функции `DefineBinaryProperty()` по типу, но предназначены для одного и того же: они вызываются для записи или считывания данных из DFM-файла. (Мы рассмотрим их подробнее чуть позже). Параметр `HasData` определяет, имеет ли свойство данные, которые необходимо сохранить.

Параметры `ReadData` и `WriteData()` функции `DefineProperty()` имеют типы `TReaderProc` и `TWriterProc` соответственно. Эти типы определяются следующим образом:

```
type
  TReaderProc = procedure(Reader: TReader) of object;
  TWriterProc = procedure(Writer: TWriter) of object;
```

Классы TReader и TWriter (специализированные потомки класса Tfiler) имеют дополнительные методы чтения и записи своих типов. Методы этих типов "наводят мосты" между публикуемыми данными компонента и DFM-файлом.

Параметры ReadData и WriteData метода DefineBinaryProperty() имеют тип TStreamProc, определенный так:

```
type
  TStreamProc = procedure(Stream: TStream) of object;
```

Поскольку методам типа TStreamProc передается только один параметр типа TStream, то двоичные данные легко можно считывать и записывать в поток. Подобно типам других методов, методы этого типа связывают с DFM-файлом нестандартные данные.

Пример использования функции DefineProperty()

Для того чтобы свести воедино всю изложенную информацию по данному вопросу, в листинге 22.8 приведен текст модуля DefProp.pas. Этот модуль иллюстрирует использование функции DefineProperty() с целью сохранения содержимого двух полей данных, объявленных в разделе private: строкового и целого.

Листинг 22.8. Модуль DefProp.pas — пример использования функции DefineProperty()

```
unit DefProp;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
  TDefinePropTest = class(TComponent)
  private
    FString: String;
    FInteger: Integer;
    procedure ReadStrData(Reader: TReader);
    procedure WriteStrData(Writer: TWriter);
    procedure ReadIntData(Reader: TReader);
    procedure WriteIntData(Writer: TWriter);
  protected
    procedure DefineProperties(Filer: TFiler); override;
  public
    constructor Create(AOwner: TComponent); override;
  end;

implementation
```

```
constructor TDefinePropTest.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  { Помещение данных в закрытые поля }
  FString := 'Ответом будет номер...';
  FInteger := 42;
end;

procedure TDefinePropTest.DefineProperties(Filer: TFiler);
begin
  inherited DefineProperties(Filer);
  { Определение новых свойств и их методов чтения/записи }
  Filer.DefineProperty('StringProp', ReadStrData, WriteStrData,
    FString <> '');
  Filer.DefineProperty('IntProp', ReadIntData, WriteIntData, True);
end;

procedure TDefinePropTest.ReadStrData(Reader: TReader);
begin
  FString := Reader.ReadString;
end;

procedure TDefinePropTest.WriteStrData(Writer: TWriter);
begin
  Writer.WriteString(FString);
end;

procedure TDefinePropTest.ReadIntData(Reader: TReader);
begin
  FInteger := Reader.ReadInteger;
end;

procedure TDefinePropTest.WriteIntData(Writer: TWriter);
begin
  Writer.WriteInteger(FInteger);
end;

end.
```

Всегда для чтения и записи строковых данных используйте методы `ReadString()` и `WriteString()` классов `TReader` и `TWriter`. Никогда не используйте похожие на них методы `ReadStr()` и `WriteStr()`, потому что они могут разрушить DFM-файл.

Для проверки работоспособности кода была создана форма с компонентом `TDefinePropTest`, добавленным в текстовом виде в окне редактора программного кода (Code Editor), как показано на рис. 22.5.

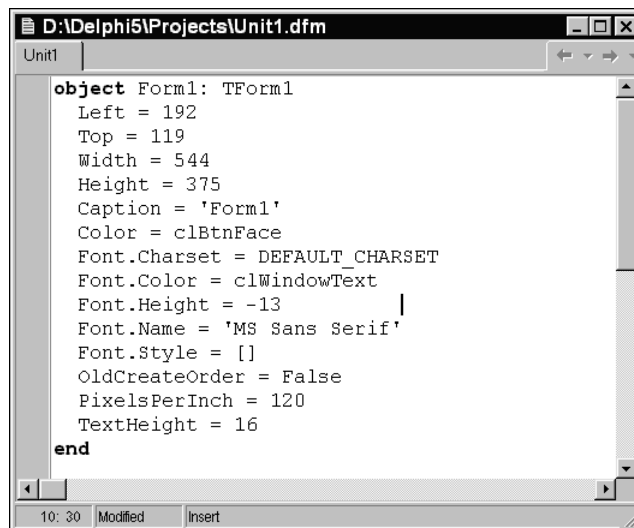


Рис. 22.5. Просмотр формы со свойствами в текстовом виде

Компонент TddgWaveFile: пример использования функции DefineBinaryProperty()

Как уже упоминалось выше, функцию DefineBinaryProperty() лучше всего использовать для сохранения вместе с компонентом графической или звуковой информации. Фактически, в библиотеке VCL эта методика используется для сохранения изображений, связанных с компонентом, например значка (Glyph) компонента TBitBtn или пиктограммы (Icon) компонента TForm. В этом разделе вы узнаете, как использовать подобную методику для сохранения звуковых данных, связанных с компонентом TddgWaveFile.

На заметку ddgWaveFile — это полнофункциональный компонент, содержащий пользовательское свойство, редактор этого свойства и редактор компонента и позволяющий воспроизводить звуки во время разработки. Об этом речь пойдет несколько позже, а пока рассмотрим механизм сохранения двоичного свойства.

Ниже приведен исходный код метода DefineProperties() компонента TddgWaveFile.

```
procedure TddgWaveFile.DefineProperties(Filer: TFiler);
{ Определяет двоичное свойство "Data" для поля FData. Благодаря этому
  значение поля FData может быть считано и записано в DFM-файл. }

  function DoWrite: Boolean;
  begin
    if Filer.Ancestor <> nil then
      Result := not (Filer.Ancestor is TddgWaveFile) or
```

```
    not Equal(TddgWaveFile(Filer.Ancestor))
  else
    Result := not Empty;
end;

begin
  inherited DefineProperties(Filer);
  Filer.DefineBinaryProperty('Data', ReadData, WriteData, DoWrite);
end;
```

Этот метод определяет двоичное свойство Data, которое считывается и записывается с помощью методов ReadData() и WriteData() компонента. При этом данные записываются только в том случае, когда функция DoWrite() возвращает значение True. Подробнее мы рассмотрим эту функцию чуть позже.

Методы ReadData() и WriteData() определяются следующим образом:

```
procedure TddgWaveFile.ReadData(Stream: TStream);
{ Читает WAV-данные из DFM-потока. }
begin
  LoadFromStream(Stream);
end;

procedure TddgWaveFile.WriteData(Stream: TStream);
{ Записывает WAV-данные в DFM-поток }
begin
  SaveToStream(Stream);
end;
```

Как видите, оба этих небольших метода просто вызывают методы LoadFromStream() и SaveToStream(), которые также определяются компонентом TddgWaveFile. Вот исходный код метода LoadFromStream():

```
procedure TddgWaveFile.LoadFromStream(S: TStream);
{ Загружает WAV-данные из потока S. Эта процедура освобождает
  память, предварительно выделенную для данных FData. }
begin
  if not Empty then
    FreeMem(FData, FDataSize);
  FDataSize := 0;
  FData := AllocMem(S.Size);
  FDataSize := S.Size;
  S.Read(FData^, FDataSize);
end;
```

Этот метод вначале определяет, не была ли ранее выделена память (посредством проверки значения поля FDataSize). Если это так (т.е. это значение больше нуля), то память, на которую указывает значение поля FData, освобождается. Затем для FData запрашивается новый блок памяти, и полю FDataSize присваивается размер потока поступающих данных. Содержимое потока затем считывается по указателю поля FData.

Метод SaveToStream() значительно проще вышеописанного. Приведем его определение.

```
procedure TddgWaveFile.SaveToStream(S: TStream);
{ Сохраняет WAV-данные в поток S. }
begin
  if FDataSize > 0 then
    S.Write(FData^, FDataSize);
end;
```

Этот метод записывает данные, на которые ссылается указатель `FData`, в `TStream`-поток `S`.

Локальная функция `DoWrite()`, вызываемая внутри метода `DefineProperties()`, определяет необходимость записи в поток значения свойства `Data`. Конечно, если поле `FData` "пустует", то ничего в поток записывать не нужно. Кроме того, следует принять дополнительные меры для того, чтобы компонент корректно работал при наследовании формы, — необходимо убедиться, что свойство `Ancestor` объекта `TFiler` не равно `nil`. Если это так, и свойство указывает на некоторую версию предка текущего компонента, необходимо также проверить, что данные, приготовленные для записи, отличаются от данных предка. Если вы не выполните эти дополнительные тесты, копия данных (в нашем случае `wav`-файла) будет записана в каждую форму потомка, и изменение `wav`-файла предка не приведет к изменению форм потомков.

В свете вышеизложенного можно отметить, что метод `DefineProperties()` — это область, в которой наиболее резко проявляются отличия между 16- и 32-разрядными версиями Delphi. В большинстве случаев компания Borland пыталась сделать наследование форм прозрачным для разработчиков компонентов, но в случае с методом `DefineProperties()` это оказалось невозможным. Хотя компоненты Delphi 1.0 функционируют и в 32-разрядной версии Delphi, они не способны без модификаций передавать по наследству обновления, связанные с наследованием форм.

На рис. 22.6 показано окно редактора кода Delphi с текстовым представлением формы, содержащей компонент `TddgWaveFile`.

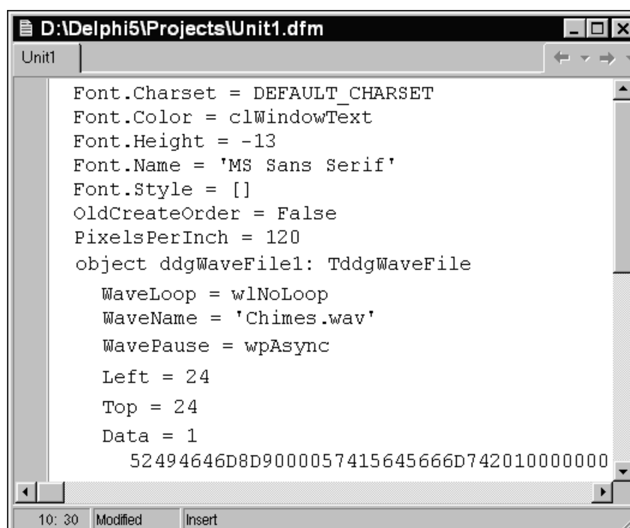


Рис. 22.6. Просмотр свойства `Data` *в окне редактора кода*

В листинге 22.9 представлен модуль `Wavez.pas`, содержащий полный исходный текст этого компонента.

Листинг 22.9. Модуль `Wavez.pas` — иллюстрирация компонента `TddgWaveFile`, инкапсулирующего `WAV`-файл

```
unit Wavez;

interface

uses
  SysUtils, Classes;

type
  { Специальный "потомок" строки используется для создания редактора свойства. }
  TWaveFileString = type string;

  EWaveError = class(Exception);

  TWavePause = (wpAsync, wpsSync);
  TWaveLoop = (wlNoLoop, wlLoop);

  TddgWaveFile = class(TComponent)
  private
    FData: Pointer;
    FDataSize: Integer;
    FWaveName: TWaveFileString;
    FWavePause: TWavePause;
    FWaveLoop: TWaveLoop;
    FOnPlay: TNotifyEvent;
    FOnStop: TNotifyEvent;
    procedure SetWaveName(const Value: TWaveFileString);
    procedure WriteData(Stream: TStream);
    procedure ReadData(Stream: TStream);
  protected
    procedure DefineProperties(Filer: TFiler); override;
  public
    destructor Destroy; override;
    function Empty: Boolean;
    function Equal(Wav: TddgWaveFile): Boolean;
    procedure LoadFromFile(const FileName: String);
    procedure LoadFromStream(S: TStream);
    procedure Play;
    procedure SaveToFile(const FileName: String);
    procedure SaveToStream(S: TStream);
    procedure Stop;
  published
    property WaveLoop: TWaveLoop read FWaveLoop write FWaveLoop;
    property WaveName: TWaveFileString read FWaveName write SetWaveName;
    property WavePause: TWavePause read FWavePause write FWavePause;
    property OnPlay: TNotifyEvent read FOnPlay write FOnPlay;
```

```
    property OnStop: TNotifyEvent read FOnStop write FOnStop;
  end;

implementation

uses MMSystem, Windows;

{ TddgWaveFile }

destructor TddgWaveFile.Destroy;
{ Обеспечивает освобождение всей выделенной ранее памяти. }
begin
  if not Empty then
    FreeMem(FData, FDataSize);
  inherited Destroy;
end;

function StreamsEqual(S1, S2: TMemoryStream): Boolean;
begin
  Result := (S1.Size = S2.Size) and CompareMem(S1.Memory, S2.Memory, S1.Size);
end;

procedure TddgWaveFile.DefineProperties(Filer: TFiler);
{ Определяет двоичное свойство "Data" для поля FData. Благодаря этому данные,
  на которые указывает поле Fdata, могут быть считаны и записаны в DFM-файл. }
function DoWrite: Boolean;
  begin
    if Filer.Ancestor <> nil then
      Result := not (Filer.Ancestor is TddgWaveFile) or
        not Equal(TddgWaveFile(Filer.Ancestor))
    else
      Result := not Empty;
  end;

begin
  inherited DefineProperties(Filer);
  Filer.DefineBinaryProperty('Data', ReadData, WriteData, DoWrite);
end;

function TddgWaveFile.Empty: Boolean;
begin
  Result := FDataSize = 0;
end;

function TddgWaveFile.Equal(Wav: TddgWaveFile): Boolean;
var
  MyImage, WavImage: TMemoryStream;
begin
  Result := (Wav <> nil) and (ClassType = Wav.ClassType);
  if Empty or Wav.Empty then
```

```
  begin
    Result := Empty and Wav.Empty;
    Exit;
  end;
  if Result then
  begin
    MyImage := TMemoryStream.Create;
    try
      SaveToStream(MyImage);
      WavImage := TMemoryStream.Create;
      try
        Wav.SaveToStream(WavImage);
        Result := StreamsEqual(MyImage, WavImage);
      finally
        WavImage.Free;
      end;
    finally
      MyImage.Free;
    end;
  end;
end;

procedure TddgWaveFile.LoadFromFile(const FileName: String);
{ Загружает WAV-данные из FileName. Обратите внимание на то, что
  эта процедура не присваивает значение свойству WaveName. }
var
  F: TFileStream;
begin
  F := TFileStream.Create(FileName, fmOpenRead);
  try
    LoadFromStream(F);
  finally
    F.Free;
  end;
end;

procedure TddgWaveFile.LoadFromStream(S: TStream);
{ Загружает WAV-данные из потока S. Эта процедура освобождает
  память, перед этим выделенную для FData. }
begin
  if not Empty then
    FreeMem(FData, FDataSize);
  FDataSize := 0;
  FData := AllocMem(S.Size);
  FDataSize := S.Size;
  S.Read(FData^, FDataSize);
end;

procedure TddgWaveFile.Play;
{ Воспроизводит WAV-звук из FData с использованием параметров,
  содержащихся в FWaveLoop и FWavePause. }
```

```
const
  LoopArray: array[TWaveLoop] of DWORD = (0, SND_LOOP);
  PauseArray: array[TWavePause] of DWORD = (SND_ASYNC, SND_SYNC);
begin
  { Убедимся, что компонент содержит данные. }
  if Empty then
    raise EWaveError.Create('Нет wave-данных');
  if Assigned(FOnPlay) then FOnPlay(Self);    // произошло событие
  {Попытка воспроизведения wave-звука. }
  if not PlaySound(FData, 0, SND_MEMORY or PauseArray[FWavePause] or
                LoopArray[FWaveLoop]) then
    raise EWaveError.Create('Ошибка при воспроизведении звука');
end;

procedure TddgWaveFile.ReadData(Stream: TStream);
{ Считывает WAV-данные из потока DFM. }
begin
  LoadFromStream(Stream);
end;

procedure TddgWaveFile.SaveToFile(const FileName: String);
{ Сохраняет WAV-данные в файле FileName. }
var
  F: TFileStream;
begin
  F := TFileStream.Create(FileName, fmCreate);
  try
    SaveToStream(F);
  finally
    F.Free;
  end;
end;

procedure TddgWaveFile.SaveToStream(S: TStream);
{ Сохраняет WAV-данные в потоке S. }
begin
  if not Empty then
    S.Write(FData^, FDataSize);
end;

procedure TddgWaveFile.SetWaveName(const Value: TWaveFileString);
{ Метод записи свойства WaveName. Этот метод отвечает за установку
  свойства WaveName и загрузку WAV-данных из файла Value. }
begin
  if Value <> '' then begin
    FWaveName := ExtractFileName(Value);
    { Не загружать из файла при загрузке из DFM-потока,
      так как DFM-поток уже содержит данные. }
    if (not (csLoading in ComponentState)) and FileExists(Value) then
      LoadFromFile(Value);
  end
```

```
  else begin
    { Если Value - пустая строка, необходимо освободить память,
      выделенную для WAV-данных. }
    FWaveName := '';
    if not Empty then
      FreeMem(FData, FDataSize);
    FDataSize := 0;
  end;
end;

procedure TddgWaveFile.Stop;
{ Останавливает воспроизведение текущего WAV-звука. }
begin
  if Assigned(FOnStop) then FOnStop(Self);  // Произошло событие
  PlaySound(Nil, 0, SND_PURGE);
end;

procedure TddgWaveFile.WriteData(Stream: TStream);
{ Записывает WAV-данные в DFM-поток. }
begin
  SaveToStream(Stream);
end;

end.
```

Категории свойств

Как мы уже отмечали в главе 1, "Программирование для Windows в Delphi 5" (том I), новинкой Delphi 5 являются *категории свойства*. Теперь свойства компонентов библиотеки VCL можно отнести к той или иной категории, а инспектор объектов получает возможность рассортировать свойства по категориям. Принадлежность свойства к определенной категории можно зарегистрировать с помощью функций `RegisterPropertyInCategory()` и `RegisterPropertiesInCategory()`, объявленных в модуле `DsgnIntf`. Первая из названных функций позволяет отнести к заданной категории одно свойство, а вторая — одновременно зарегистрировать сразу несколько свойств.

Функция `RegisterPropertyInCategory()` является перегружаемой, в результате чего можно использовать четыре ее версии, предназначенные для удовлетворения различных потребностей программистов. Всем версиям этой функции передается в качестве первого параметра класс `TPropertyCategoryClass`, описывающий требуемую категорию. Что касается остальной части списка передаваемых параметров, то каждой из версии передается различная комбинация параметров (состоящая из имени свойства, типа свойства и класса компонента). Это позволяет программисту выбрать наилучший метод регистрации созданных им свойств. Приводим объявления различных версий функции `RegisterPropertyInCategory()`:

```
function RegisterPropertyInCategory(ACategoryClass: TPropertyCategoryClass;
  const APropertyName: string): TPropertyFilter; overload;
function RegisterPropertyInCategory(ACategoryClass: TPropertyCategoryClass;
  AComponentClass: TClass; const APropertyName: string): TPropertyFilter
```

```
  overload;
function RegisterPropertyInCategory(ACategoryClass: TPropertyCategoryClass;
  APropertyType: PTypeInfo; const APropertyName: string): TPropertyFilter;
  overload;
function RegisterPropertyInCategory(ACategoryClass: TPropertyCategoryClass;
  APropertyType: PTypeInfo): TPropertyFilter; overload;
```

Эти функции достаточно интеллектуальны, чтобы надлежащим образом воспринимать символы шаблона, благодаря чему можно отнести к определенной категории все свойства, имена которых совпадают, к примеру, со строкой `'Data*'`. Чтобы получить полный список поддерживаемых Delphi символов шаблона и информацию об их поведении, обратитесь к интерактивной справочной системе Delphi (найдите раздел, описывающий класс `TMask`).

Предусмотрено три варианта перегружаемой функции `RegisterPropertiesInCategory()`:

```
function RegisterPropertiesInCategory(ACategoryClass: TPropertyCategoryClass;
  const AFilters: array of const): TPropertyCategory; overload;
function RegisterPropertiesInCategory(ACategoryClass: TPropertyCategoryClass;
  AComponentClass: TClass; const AFilters: array of string): TPropertyCategory;
  overload;
function RegisterPropertiesInCategory(ACategoryClass: TPropertyCategoryClass;
  APropertyType: PTypeInfo; const AFilters: array of string): TPropertyCategory;
  overload;
```

Классы категорий

Тип `TPropertyCategoryClass` представляет собой ссылку на класс категории свойства. Для всех стандартных категорий свойств в библиотеке VCL базовым классом является класс `TPropertyCategory`. Существует 12 стандартных категорий свойств, и все они описаны в табл. 22.4.

Таблица 22.4. Классы стандартных категорий свойств

Имя класса	Описание
`TactionCategory`	Свойства, связанные с действиями времени выполнения. К этой категории относятся свойства `Enabled` и `Hint` компонента `TControl`
`TDatabaseCategory`	Свойства, относящиеся к операциям с базами данных. К этой категории относятся свойства `DatabaseName` и `SQL` компонента `TQuery`
`TDragNDropCategory`	Свойства, связанные с операциями перетаскивания и пристыковки. К этой категории относятся свойства `DragCursor` и `DragKind` компонента `Tcontrol`
`THelpCategory`	Свойства, связанные с использованием интерактивной справки и подсказок. К этой категории относятся свойства `HelpContext` и `Hint` компонента `TwinControl`
`TLayoutCategory`	Свойства, связанные с визуальным отображением элемента управления во время разработки. К этой категории относятся свойства `Top` и `Left` компонента `Tcontrol`
`TLegacyCategory`	Свойства, связанные с уже устаревшими операциями. К этой категории относятся свойства `Ctl3D` и `ParentCtl3D` компонента `TwinControl`

Окончание табл. 22.4

Имя класса	Описание
TLinkageCategory	Свойства, имеющие отношение к присоединению или связыванию одного компонента с другим. К этой категории относится свойство DataSet компонента TDataSource
TLocaleCategory TLocalizableCategory	Свойства, связанные с локализациями международного характера. К этой категории относятся свойства BiDiMode и ParentBiDiMode компонента TControl
TMiscellaneousCategory	Свойства, которые либо не подходят ни под одну из существующих категорий, либо не нуждаются в причислении к какой-либо категории, либо не зарегистрированы в явном виде в определенной категории. К этой категории принадлежат свойства AllowAllUp и Name компонента TSpeedButton
TVisualCategory	Свойства, связанные с визуальным отображением элемента управления во время выполнения. К этой категории относятся свойства Align и Visible компонента TControl
TInputCategory	Свойства, связанные с вводом данных (они необязательно имеют отношение к операциям с базами данных). К этой категории относятся свойства Enabled и ReadOnly компонента TEdit

Допустим, что создан компонент с именем TNeato, у которого есть свойство Keen, и требуется зарегистрировать это свойство в качестве члена категории Action, представленной классом TActionCategory. Это можно сделать путем добавления в процедуру Register() обращения к функции RegisterPropertyInCategory(), как показано ниже.

```
RegisterPropertyInCategory(TActionCategory, TNeato, 'Keen');
```

Пользовательские категории

Как вы уже знаете, категория свойства представляется в программном коде как класс, который является потомком класса TPropertyCategory. Но насколько трудно создать свои собственные категории свойств? Оказывается, это не так уж и сложно. В большинстве случаев для этого достаточно переопределить виртуальные функции Name() и Description() класса TPropertyCategory, и тогда вы получите информацию, относящуюся к вашей категории.

В качестве примера мы создадим новую категорию Sound, к которой можно причислить некоторые свойства компонента TddgWaveFile (с которым мы уже познакомились выше в этой главе). Объявление класса новой категории, TSoundCategory, представлено в листинге 22.10. Этот листинг содержит текст файла WavezEd.pas, включающего определение категории компонента, редактор его свойств и редактор самого компонента.

Листинг 22.10. Модуль WavezEd.pas — редактор свойства для компонента TddgWaveFile

```
unit WavezEd;

interface

uses DsgnIntf;
```

```
type
  { Категория для некоторых свойств компонента TddgWaveFile }
  TSoundCategory = class(TPropertyCategory)
  public
    class function Name: string; override;
    class function Description: string; override;
  end;

  { Редактор для свойства WaveName компонента TddgWaveFile }
  TWaveFileStringProperty = class(TStringProperty)
  public
    procedure Edit; override;
    function GetAttributes: TPropertyAttributes; override;
  end;

  { Редактор для компонента TddgWaveFile. Позволяет воспроизводить и
    останавливать звучание WAV-данных с помощью локального меню в среде IDE. }
  TWaveEditor = class(TComponentEditor)
  private
    procedure EditProp(PropertyEditor: TPropertyEditor);
  public
    procedure Edit; override;
    procedure ExecuteVerb(Index: Integer); override;
    function GetVerb(Index: Integer): string; override;
    function GetVerbCount: Integer; override;
  end;

implementation

uses TypInfo, Wavez, Classes, Controls, Dialogs;

{ TSoundCategory }

class function TSoundCategory.Name: string;
begin
  Result := 'Sound';
end;

class function TSoundCategory.Description: string;
begin
  Result := 'Свойства, связанные с воспроизведением звуков'
end;

{ TWaveFileStringProperty }

procedure TWaveFileStringProperty.Edit;
{ Выполняется по щелчку на кнопке с многоточием рядом со
  строкой свойства WavName в окне инспектора объектов. Этот метод позволяет
  пользователю выбрать файл в диалоговом окне OpenDialog и задать
  его в качестве значения свойства. }
begin
```

```
  with TOpenDialog.Create(nil) do
    try
      { Устанавливаем свойства диалогового окна }
      Filter := 'Wav files|*.wav|All files|*.*';
      DefaultExt := '*.wav';
      { Помещаем текущее значение в свойство FileName диалогового окна }
      FileName := GetStrValue;
      { Выполняем диалог и устанавливаем значение свойства, если это
        диалоговое окно закрывается по щелчку на кнопке OK }
      if Execute then
        SetStrValue(FileName);
    finally
      Free;
    end;
end;

function TWaveFileStringProperty.GetAttributes: TPropertyAttributes;
{ Указывает на то, что редактор свойств будет вызывать диалоговое окно. }
begin
  Result := [paDialog];
end;

{ TWaveEditor }

const
  VerbCount = 2;
  VerbArray: array[0..VerbCount - 1] of string[7] = ('Play', 'Stop');

procedure TWaveEditor.Edit;
{ Вызывается, когда пользователь дважды щелкает на компоненте во время
  разработки. Этот метод вызывает метод GetComponentProperties,
  чтобы обратиться к методу Edit редактора свойства WaveName. }
var
  Components: TDesignerSelectionList;
begin
  Components := TDesignerSelectionList.Create;
  try
    Components.Add(Component);
    GetComponentProperties(Components, tkAny, Designer, EditProp);
  finally
    Components.Free;
  end;
end;

procedure TWaveEditor.EditProp(PropertyEditor: TPropertyEditor);
{ Вызывается один раз для каждого свойства в ответ на вызов функции
  GetComponentProperties. Этот метод ищет редактор свойства WaveName
  и вызывает свой метод Edit. }
begin
  if PropertyEditor is TWaveFileStringProperty then begin
    TWaveFileStringProperty(PropertyEditor).Edit;
```

```
    Designer.Modified;    // Сообщает конструктору о модификации
  end;
end;

procedure TWaveEditor.ExecuteVerb(Index: Integer);
begin
  case Index of
    0: TddgWaveFile(Component).Play;
    1: TddgWaveFile(Component).Stop;
  end;
end;

function TWaveEditor.GetVerb(Index: Integer): string;
begin
  Result := VerbArray[Index];
end;

function TWaveEditor.GetVerbCount: Integer;
begin
  Result := VerbCount;
end;

end.
```

После определения класса категории остается лишь зарегистрировать свойства для этой категории, используя одну из функций регистрации. Это делается в процедуре `Register()` для компонента `TddgWaveFile` с помощью следующих строк кода:

```
RegisterPropertiesInCategory(TSoundCategory, TddgWaveFile,
    ['WaveLoop', 'WaveName', 'WavePause']);
```

На рис. 22.7 показано окно инспектора объектов, в котором отображаются свойства компонента `TddgWaveFile`, рассортированные по категориям.

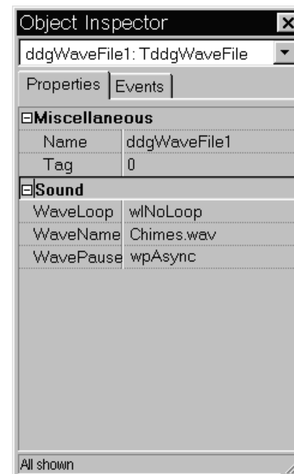


Рис. 22.7. Просмотр свойств компонента `TddgWaveFile` по категориям

Списки компонентов: классы `TCollection` и `TCollectionItem`

Довольно часто компоненты поддерживают или владеют списками некоторых элементов — типов данных, записей, объектов и даже других компонентов. В некоторых случаях удобно инкапсулировать такой список в специальный объект и сделать этот объект свойством владельца компонента. Примером подобного подхода служит свойство `Lines` компонента

TMemo. Это свойство имеет тип TStrings, инкапсулирующий список строк. При этом объект TStrings отвечает за механизм работы с потоками, используемый для записи строк в файл формы при сохранении этой формы.

Как быть, если требуется сохранить список элементов компонентного или объектного типов, которые еще не инкапсулированы существующим классом, таким как TStrings? В этом случае можно было бы создать класс, обеспечивающий работу элементов списка с потоками данных, и сделать его свойством компонента-владельца. В качестве альтернативного варианта можно переопределить стандартный механизм работы с потоками данных компонента-владельца так, чтобы он знал, как работать с этим списком элементов. Однако самое лучшее решение — воспользоваться преимуществами, предоставляемыми классами TCollection и TCollectionItem.

Класс TCollection — это объект, используемый для хранения списка объектов типа TCollectionItem. Сам класс TCollection — это не компонент, а просто потомок класса TPersistent. Обычно класс TCollection связан с существующим компонентом.

Чтобы использовать класс TCollection для хранения списка элементов, необходимо создать класс, производный от класса TCollection, назвав его, например, TNewCollection. Этот класс будет служить типом свойства компонента. Затем следует создать новый класс из класса TCollectionItem, который можно назвать TNewCollectionItem. Класс TNewCollection будет вести список объектов типа TNewCollectionItem. Вся прелесть такого подхода состоит в том, что для того, чтобы принадлежащие классу TNewCollectionItem данные были обработаны обычными средствами работы с потоками данных, достаточно их опубликовать (объявить в разделе published компонента TNewCollectionItem). Delphi знает, как записывать опубликованные свойства в поток и считывать их из него.

Примером использования класса TCollection может служить компонент TStatusBar, являющийся потомком класса TWinControl. Одно из его свойств — Panels, которое имеет тип TStatusPanels. Класс TStatusPanels является потомком класса TCollection и определяется следующим образом:

```
type
  TStatusPanels = class(TCollection)
  private
    FStatusBar: TStatusBar;
    function GetItem(Index: Integer): TStatusPanel;
    procedure SetItem(Index: Integer; Value: TStatusPanel);
  protected
    procedure Update(Item: TCollectionItem); override;
  public
    constructor Create(StatusBar: TStatusBar);
    function Add: TStatusPanel;
    property Items[Index: Integer]: TStatusPanel read GetItem
       write SetItem; default;
  end;
```

Класс TStatusPanels хранит список потомков класса TCollectionItem — объектов класса TStatusPanel. Ниже приведен код определения этого класса.

```
type
  TStatusPanel = class(TCollectionItem)
  private
    FText: string;
    FWidth: Integer;
```

```
    FAlignment: TAlignment;
    FBevel: TStatusPanelBevel;
    FStyle: TStatusPanelStyle;
    procedure SetAlignment(Value: TAlignment);
    procedure SetBevel(Value: TStatusPanelBevel);
    procedure SetStyle(Value: TStatusPanelStyle);
    procedure SetText(const Value: string);
    procedure SetWidth(Value: Integer);
  public
    constructor Create(Collection: TCollection); override;
    procedure Assign(Source: TPersistent); override;
  published
    property Alignment: TAlignment read FAlignment
      write SetAlignment default taLeftJustify;
    property Bevel: TStatusPanelBevel read FBevel
      write SetBevel default pbLowered;
    property Style: TStatusPanelStyle read FStyle write SetStyle
      default psText;
    property Text: string read FText write SetText;
    property Width: Integer read FWidth write SetWidth;
  end;
```

Свойства типа TStatusPanel, объявленные в разделе published класса, автоматически получают возможность работы с потоками в Delphi. В классе TStatusPanel предусмотрена передача параметра типа TCollection конструктору Create(), благодаря чему обеспечивается связь с классом TCollection. В нашем случае конструктору класса TStatusPanel передается компонент TStatusBar, с которым объект TStatusPanel должен быть связан. Класс TCollection обладает механизмом работы с потоками данных, являющихся компонентами класса TCollectionItem, а также определяет некоторые методы и свойства для манипулирования элементами, поддерживаемыми в классе TCollection. Их описание можно найти в интерактивной справке.

Для иллюстрации использования этих двух новых классов нами создан компонент TddgLaunchPad, разрешающий пользователю хранить список компонентов TddgRunButton (компонент TddgRunButton описан в предыдущей главе).

Компонент TddgLaunchPad — потомок компонента TScrollBox. Одно из свойств нового компонента — RunButtons, является потомком класса Tcollection и поддерживает список компонентов TRunBtnItem. Компонент TRunBtnItem — это потомок класса TCollectionItem, свойства которого используются для создания компонента TddgRunButton, помещенного в компонент TddgLaunchPad. В следующих разделах мы рассмотрим важные моменты, связанные с его созданием.

Определение класса типа TCollectionItem: компонент TRunBtnItem

Вначале нужно определить элементы, которые будут содержаться в списке. Для компонента TddgLaunchPad это будут компоненты типа TddgRunButton. Таким образом, каждый экземпляр компонента TRunBtnItem должен быть связан с компонентом TddgRunButton. Ниже приводится часть определения класса TRunBtnItem:

```
type
  TRunBtnItem = class(TCollectionItem)
  private
```

```
    FCommandLine: String;   // Хранит командную строку
    FLeft: Integer;         // Хранят позиционные свойства
    FTop: Integer;          // компонента TddgRunButton
    FRunButton: TddgRunButton; // Указатель на компонент TddgRunButton
     ...
  public
    constructor Create(Collection: TCollection); override;
  published
    { Эти свойства получают возможность работы с потоками данных }
    property CommandLine: String read FCommandLine write SetCommandLine;
    property Left: Integer read FLeft write SetLeft;
    property Top: Integer read FTop write SetTop;
  end;
```

Обратите внимание на то, что компонент TRunBtnItem содержит указатель на компонент TddgRunButton и помещает в раздел published только те свойства, которые необходимы для создания компонента TddgRunButton. Можно было бы подумать, что если компонент TRunBtnItem связывается с компонентом TddgRunButton, то достаточно просто опубликовать этот компонент и предоставить остальную работу механизму работы с потоками. Однако при этом возникают определенные проблемы, связанные с тем, что механизм работы с потоками по-разному обрабатывает классы TComponent и TPersistent. Основное правило обработки здесь заключается в том, что система работы с потоками данных отвечает за создание новых экземпляров каждого найденного в потоке класса, произошедшего от класса TComponent; в то же время предполагается, что экземпляры класса TPersistent уже существуют и в создании новых необходимости нет. Следуя этому правилу, мы записываем в поток информацию, необходимую для компонента TddgRunButton, а затем создаем экземпляр компонента TddgRunButton в конструкторе класса TRunBtnItem.

Определение класса типа TCollection: компонент TRunButtons

Следующим шагом будет определение объекта, обслуживающего список компонентов TRunBtnItem. Как уже отмечалось, этот объект должен быть потомком класса TCollection. Мы назвали этот класс TRunButtons. Ниже следует его определение.

```
type
  TRunButtons = class(TCollection)
  private
    FLaunchPad: TddgLaunchPad; // Хранит указатель на TddgLaunchPad
    function GetItem(Index: Integer): TRunBtnItem;
    procedure SetItem(Index: Integer; Value: TRunBtnItem);
  protected
    procedure Update(Item: TCollectionItem); override;
  public
    constructor Create(LaunchPad: TddgLaunchPad);
    function Add: TRunBtnItem;
    procedure UpdateRunButtons;
    property Items[Index: Integer]: TRunBtnItem read GetItem
      write SetItem; default;
  end;
```

Класс TRunButtons связывает себя с компонентом TddgLaunchPad, который будет рассмотрен чуть ниже. Это происходит в конструкторе Create(), который, как вы можете видеть, принимает компонент TddgLaunchPad в качестве параметра. Обратите внимание на различные свойства и методы, добавление которых позволило компоненту манипулировать отдельными классами TRunBtnItem, в частности свойством Items, представляющим собой массив для списка компонентов TRunBtnItem.

Использование классов TRunBtnItem и TRunButtons станет более понятным после рассмотрения реализации компонента TddgLaunchPad.

Реализация классов TddgLaunchPad, TRunBtnItem и TRunButtons

Компонент TddgLaunchPad имеет свойство типа TRunButtons. Его реализация, как и реализация компонентов TRunBtnItem и TRunButtons, приведена в листинге 22.11.

Листинг 22.11. Модуль LnchPad.pas — иллюстрация реализации компонента TddgLaunchPad

```
unit LnchPad;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, RunBtn, ExtCtrls;

type
  TddgLaunchPad = class;

  TRunBtnItem = class(TCollectionItem)
  private
    FCommandLine: String;   // Хранит командную строку
    FLeft: Integer;         // Хранят позиционные свойства
    FTop: Integer;          // компонента TddgRunButton
    FRunButton: TRunButton; // Указатель на TRunButton
    FWidth: Integer;        // Хранят размеры
    FHeight: Integer;
    procedure SetCommandLine(const Value: string);
    procedure SetLeft(Value: Integer);
    procedure SetTop(Value: Integer);
  public
    constructor Create(Collection: TCollection); override;
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); override;
    property Width: Integer read FWidth;
    property Height: Integer read FHeight;
  published
    { Эти свойства получают возможность работы с потоками данных }
    property CommandLine: String read FCommandLine
```

```
      write SetCommandLine;
    property Left: Integer read FLeft write SetLeft;
    property Top: Integer read FTop write SetTop;
  end;

  TRunButtons = class(TCollection)
  private
    FLaunchPad: TddgLaunchPad; // Содержит указатель на TddgLaunchPad
    function GetItem(Index: Integer): TRunBtnItem;
    procedure SetItem(Index: Integer; Value: TRunBtnItem);
  protected
    procedure Update(Item: TCollectionItem); override;
  public
    constructor Create(LaunchPad: TddgLaunchPad);
    function Add: TRunBtnItem;
    procedure UpdateRunButtons;
    property Items[Index: Integer]: TRunBtnItem read
      GetItem write SetItem; default;
  end;

  TddgLaunchPad = class(TScrollBox)
  private
    FRunButtons: TRunButtons;
    TopAlign: Integer;
    LeftAlign: Integer;
    procedure SetRunButtons(Value: TRunButtons);
    procedure UpdateRunButton(Index: Integer);
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure GetChildren(Proc: TGetChildProc; Root: TComponent); override;
  published
    property RunButtons: TRunButtons read FRunButtons write SetRunButtons;
  end;

implementation

{ TRunBtnItem }

constructor TRunBtnItem.Create(Collection: TCollection);
{ Конструктор принимает в качестве параметра коллекцию типа TCollection,
  которая владеет этим элементом TRunBtnItem. }
begin
  inherited Create(Collection);
  { Создает экземпляр компонента FRunButton. Делает объект панели загрузки
    владельцем и родителем. Затем инициализирует его различные свойства. }
  FRunButton := TRunButton.Create(TRunButtons(Collection).FLaunchPad);
  FRunButton.Parent := TRunButtons(Collection).FLaunchPad;
  FWidth := FRunButton.Width;   // Сохраняет размеры
  FHeight := FRunButton.Height;
```

```
end;

destructor TRunBtnItem.Destroy;
begin
  FRunButton.Free;   // Уничтожение экземпляра TddgRunButton
  inherited Destroy; // Вызов унаследованного деструктора Destroy
end;

procedure TRunBtnItem.Assign(Source: TPersistent);
{ Необходимо переопределить метод TCollectionItem.Assign таким образом,
  чтобы он знал, как копировать данные из одного экземпляра TRunBtnItem
  в другой. Если это так, то вызывать унаследованный метод Assign() не нужно. }
begin
  if Source is TRunBtnItem then
  begin
    {Вместо присвоения командной строки полю FCommandLine выполняем
     присвоение значения соответствующему свойству путем вызова метода
     доступа. Метод доступа осуществляет все необходимые побочные действия. }
    CommandLine := TRunBtnItem(Source).CommandLine;
    { Копирование значений в остальные поля и выход из процедуры. }
    FLeft := TRunBtnItem(Source).Left;
    FTop := TRunBtnItem(Source).Top;
    Exit;
  end;
  inherited Assign(Source);
end;

procedure TRunBtnItem.SetCommandLine(const Value: string);
{ Это метод записи для свойства TRunBtnItem.CommandLine. Он гарантирует,
  что экземпляру FRunButton компонента TddgRunButton будет присвоено
  заданное значение строки из параметра Value. }
begin
  if FRunButton <> nil then
  begin
    FCommandLine := Value;
    FRunButton.CommandLine := FCommandLine;
    { Это приводит к вызову метода TRunButtons.Update
      для каждого элемента TRunBtnItem. }
    Changed(False);
  end;
end;

procedure TRunBtnItem.SetLeft(Value: Integer);
{ Метод доступа к свойству TRunBtnItem.Left. }
begin
  if FRunButton <> nil then
  begin
    FLeft := Value;
    FRunButton.Left := FLeft;
   end;
```

```
end;

procedure TRunBtnItem.SetTop(Value: Integer);
{ Метод доступа к свойству TRunBtnItem.Top. }
begin
  if FRunButton <> nil then
  begin
    FTop := Value;
    FRunButton.Top := FTop;
   end;
end;

{ TRunButtons }

 constructor TRunButtons.Create(LaunchPad: TddgLaunchPad);
{ Конструктор заставляет FLaunchPad ссылаться на параметр типа TddgLaunchPad.
  LaunchPad является владельцем этой коллекции, и для доступа к ней
  необходимо иметь соответствующий указатель. }
begin
  inherited Create(TRunBtnItem);
  FLaunchPad := LaunchPad;
end;

function TRunButtons.GetItem(Index: Integer): TRunBtnItem;
{ Метод доступа TRunButtons.Items, возвращающий
  экземпляр компонента TRunBtnItem. }
begin
  Result := TRunBtnItem(inherited GetItem(Index));
end;

procedure TRunButtons.SetItem(Index: Integer; Value: TRunBtnItem);
{ Метод доступа к свойству TRunButton.Items, позволяющий
  выполнить присвоение элементу с заданным индексом. }
begin
  inherited SetItem(Index, Value)
end;

procedure TRunButtons.Update(Item: TCollectionItem);
{ Метод TCollection.Update вызывается объектами типа TCollectionItem
  при изменении одного из элементов коллекции. Изначально это - абстрактный
  метод. Он должен быть переопределен для включения необходимой логики
  обработки изменения элемента TCollectionItem. Мы используем его для
  перерисовки элемента путем вызова метода TddgLaunchPad.UpdateRunButton.}
begin
  if Item <> nil then
    FLaunchPad.UpdateRunButton(Item.Index);
end;

procedure TRunButtons.UpdateRunButtons;
{ UpdateRunButtons - открытая процедура, предоставляющая пользователю
```

```
  компонентов TRunButtons возможность принудительной перерисовки всех
  кнопок. Этот метод вызывает метод TddgLaunchPad.UpdateRunButton для
  каждого экземпляра TRunBtnItem. }
var
  i: integer;
begin
  for i := 0 to Count - 1 do
    FLaunchPad.UpdateRunButton(i);
end;

function TRunButtons.Add: TRunBtnItem;
{ Этот метод должен быть переопределен таким образом, чтобы возвращать
  экземпляр TRunBtnItem при вызове унаследованного метода Add. Это
  достигается путем приведения типа для результата унаследованного метода. }
begin
  Result := TRunBtnItem(inherited Add);
end;

{ TddgLaunchPad }

constructor TddgLaunchPad.Create(AOwner: TComponent);
{ Инициализация экземпляра TRunButtons и внутренних переменных, используемых
  для позиционирования элемента TRunBtnItem при его отображении. }
begin
  inherited Create(AOwner);
  FRunButtons := TRunButtons.Create(Self);
  TopAlign := 0;
  LeftAlign := 0;
end;

destructor TddgLaunchPad.Destroy;
begin
  FRunButtons.Free;  // Освобождение экземпляра TRunButtons
  inherited Destroy; // Вызов унаследованного деструктора
end;

procedure TddgLaunchPad.GetChildren(Proc: TGetChildProc; Root: TComponent);
{ Переопределяем метод GetChildren, чтобы компонент TddgLaunchpad игнорировал
  любые принадлежащие ему элементы TRunButtons, поскольку они не нуждаются
  в потоковой обработке в контексте компонента TddgLaunchPad. Вся информация,
  необходимая для создания экземпляров TRunButton, уже записана в поток в виде
  публикуемых свойств потомка класса TCollectionItem - класса TRunBtnItem.
  Этот метод предотвращает двойную запись в поток компонента TRunButton. }
var
  I: Integer;
begin
  for I := 0 to ControlCount - 1 do
    { Игнорирует кнопки запуска и поле прокрутки. }
    if not (Controls[i] is TRunButton) then
      Proc(TComponent(Controls[I]));
```

```
end;

procedure TddgLaunchPad.SetRunButtons(Value: TRunButtons);
{ Метод доступа для свойства RunButtons. }
begin
  FRunButtons.Assign(Value);
end;

procedure TddgLaunchPad.UpdateRunButton(Index: Integer);
{ Этот метод отвечает за отображение экземпляров TRunBtnItem. Он гарантирует,
  что элементы TRunBtnItem не будут выходить за пределы панели
  TddgLaunchPad - при необходимости могут создаваться новые строки
  таких элементов. Это происходит только при добавлении/удалении элементов
  TRunBtnItems. Пользователь имеет возможность так изменить размеры панели
  TddgLaunchPad, что она будет меньше ширины строки с элементами TRunBtnItem. }
begin
  { Вначале рисуется первый элемент, обе его позиции обнуляются. }
  if Index = 0 then
  begin
    TopAlign := 0;
    LeftAlign := 0;
  end;
  { Если ширина текущей строки элементов TRunBtnItem больше ширины
    панели TddgLaunchPad, создается новая строка элементов TRunBtnItem. }
  if (LeftAlign + FRunButtons[Index].Width) > Width then
  begin
    TopAlign := TopAlign + FRunButtons[Index].Height;
    LeftAlign := 0;
  end;
  FRunButtons[Index].Left := LeftAlign;
  FRunButtons[Index].Top := TopAlign;
  LeftAlign := LeftAlign + FRunButtons[Index].Width;
end;

end.
```

Реализация компонента TRunBtnItem

Конструктор TRunBtnItem.Create() создает экземпляр компонента TddgRunButton. Каждый элемент TRunBtnItem в коллекции обслуживает собственный экземпляр TddgRunButton. Следующие две строки конструктора требуют пояснения:

```
FRunButton := TddgRunButton.Create(TRunButtons(Collection).FLaunchPad);
FRunButton.Parent := TRunButtons(Collection).FLaunchPad;
```

При выполнении первой строки кода создается экземпляр компонента TddgRunButton — FRunButton. Владельцем экземпляра FRunButton является переменная FLaunchPad, которая представляет собой экземпляр компонента TddgLaunchPad и поле объекта TCollection, передаваемого в качестве параметра. Необходимо использовать экземпляр FLaunchPad в качестве

владельца экземпляра FRunButton, потому что ни экземпляр компонента TRunBtnItem, ни объект TRunButton не могут быть владельцами (так как они являются потомками класса TPersistent). Запомните, что владельцем может быть только потомок класса TComponent!

Хотелось бы обратить ваше внимание на проблему, возникающую при использовании экземпляра класса FLaunchPad в качестве владельца экземпляра FRunButton. Мы сделали экземпляр FLaunchPad владельцем FRunButton во время разработки. Обычное поведение механизма работы с потоками заставит Delphi при сохранении формы записать в поток экземпляр FRunButton как компонент, принадлежащий экземпляру FLaunchPad. Нам это не нужно, поскольку экземпляр FRunButton создается в конструкторе компонента TRunBtnItem на основе информации, записанной в поток в контексте компонента TRunBtnItem. Ниже вы увидите, каким образом мы предотвратили запись в поток компонентов TddgRunButton компонентом TddgLaunchPad, чтобы избежать подобного нежелательного поведения.

Вторая строка приведенного выше фрагмента кода назначает экземпляр FLaunchPad родителем экземпляра FRunButton, чтобы тот отвечал за отображение компонента FRunButton.

Деструктор TRunBtnItem.Destroy() освобождает экземпляр FRunButton перед вызовом унаследованного деструктора.

При определенных обстоятельствах становится необходимостью переопределение метода TRunBtnItem.Assign(). Примером таких обстоятельств может служить ситуация, когда приложение первый раз запускается, и форма считывается из потока. Именно с помощью метода Assign() экземпляру компонента TRunBtnItem выдается указание присвоить потоковые значения своих свойств свойствам компонента, которого он контролирует (в данном случае это компонент TddgRunButton).

Другие методы — это просто методы доступа к различным свойствам компонента TRunBtnItem (они разъясняются в комментариях, содержащихся в листинге).

Реализация компонента TRunButtons

Метод TRunButtons.Create() присваивает указателю FLaunchPad ссылку на параметр типа TddgLaunchPad — LaunchPad, для того чтобы на него можно было ссылаться в дальнейшем.

Метод TRunButtons.Update() вызывается при изменении одного из экземпляров компонента TRunBtnItem. Этот метод содержит определенный алгоритм, выполняющийся при такого рода изменении, и используется для вызова метода компонента TddgLaunchPad, который перерисовывает экземпляры TRunBtnItem. Мы также добавили открытый метод UpdateRunButtons(), который принудительно (по желанию пользователя) перерисовывает элементы.

Остальные методы компонента TRunButtons являются методами доступа к свойствам, описанных в комментариях к листингу 22.11.

Реализация компонента TddgLaunchPad

Конструктор и деструктор компонента TddgLaunchPad не отличаются большой сложностью. Метод TddgLaunchPad.Create() создает экземпляр объекта TRunButtons и передает ему себя в качестве параметра конструктора. Деструктор TddgLaunchPad.Destroy() освобождает экземпляр компонента TRunButtons.

Переопределение метода TddgLaunchPad.GetChildren() — очень важный момент. Благодаря ему предотвращается запись в поток экземпляров компонента TddgRunButton, хранимых в коллекции, как компонентов с владельцем TddgLaunchPad. Запомните: это необходимо, так

как они должны создаваться не в контексте объекта TddgLaunchPad, а в контексте экземпляров компонента TRunBtnItem. Поскольку ни один компонент TddgRunButton не передается в процедуру Proc, они не будут записываться или считываться из потока.

Метод TddgLaunchPad.UpdateRunButton() используется для отображения экземпляров TddgRunButton, которые содержатся в коллекции. Логика, реализованная в этом методе, гарантирует, что кнопки никогда не "выйдут" за пределы панели TddgLaunchPad. Поскольку компонент TddgLaunchPad — потомок класса TScrollBox, прокрутка будет осуществляться в вертикальном направлении.

Остальные методы — просто методы доступа к свойствам, описанным в комментариях листинга 22.11.

Наконец, мы регистрируем редактор свойств класса коллекции TRunButtons в процедуре Register() данного модуля. В следующем разделе как раз и рассматривается этот редактор свойств, а также один из способов редактирования списка компонентов в диалоговом окне редактора свойств.

Редактирование списка компонентов TCollectionItem в диалоговом окне редактора свойств

Теперь, когда уже определен компонент TddgLaunchPad, класс коллекции TRunButtons и класс коллекции TRunBtnItem, необходимо обеспечить пользователя способом добавления компонентов TddgRunButton в коллекцию TRunButtons. Лучше всего для этого подходит редактор свойств списка, обслуживаемого коллекцией TRunButtons.

Редактор свойства представляет собой диалоговое окно, показанное на рис. 22.8.

Рис. 22.8. Окно редактора коллекции RunButtons — компонента TddgLaunchPad

Это диалоговое окно предназначено для проведения непосредственных манипуляций компонентами TRunBtnItem, поддерживаемыми коллекцией RunButtons типа TddgLaunchPad. В списке PathListBox отображаются различные строки CommandLine для каждого элемента TddgRunButton, "вложенного" в компонент TRunBtnItem. В компоненте TddgRunButton отображается текущий выбранный элемент списка. Диалоговое окно также содержит кнопки, позволяющие пользователю добавлять и удалять элемент, принимать изменения и отменять операции. По мере того как пользователь вносит изменения в диалоговое окно, они отображаются в компоненте TddgLaunchPad.

Совет

Для редакторов свойств используется соглашение о включении в них кнопки Apply для подтверждения изменения формы. Мы не рассматриваем это включение здесь, но вы можете самостоятельно, в качестве упражнения, попытаться добавить такую кнопку в редактор свойства RunButtons. Чтобы посмотреть, как работает кнопка Apply, обратитесь к исходному коду редактора свойства Panels компонента TStatusBar, расположенного во вкладке Win32 палитры компонентов.

На рис. 22.9 показано окно компонента TLaunchPadEditor — редактора свойства RunButtons с находящимися в нем несколькими элементами. Здесь же присутствует сам компонент TddgLaunchPad, в котором размещаются компоненты TddgRunButton, перечисленные в окне редакторе свойства RunButtons.

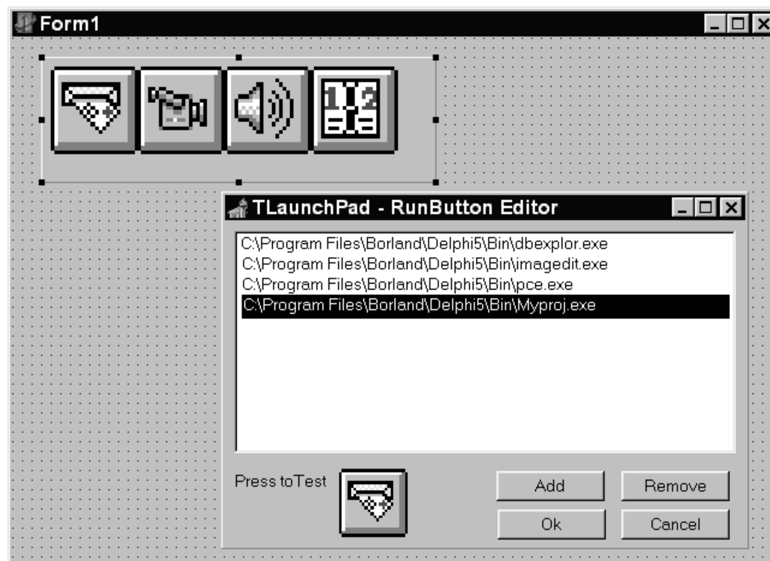


Рис. 22.9. Окно редактора свойства RunButton компонента TddgLaunchPad с компонентами TRunBtnItem

В листинге 22.12 представлен исходный код редактора свойства RunButtons.TddgLaunchPad и его диалогового окна.

Листинг 22.12. Модуль LPadPE.pas: редактор свойств TRunButtons

```
unit LPadPE;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Buttons, RunBtn, StdCtrls, LnchPad, DsgnIntF, TypInfo, ExtCtrls;

type

  { Вначале объявляем диалоговое окно редактора. }
  TLaunchPadEditor = class(TForm)
```

```
    PathListBox: TListBox;
    AddBtn: TButton;
    RemoveBtn: TButton;
    CancelBtn: TButton;
    OkBtn: TButton;
    Label1: TLabel;
    pnlRBtn: TPanel;
    procedure PathListBoxClick(Sender: TObject);
    procedure AddBtnClick(Sender: TObject);
    procedure RemoveBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure CancelBtnClick(Sender: TObject);
  private
    TestRunBtn: TddgRunButton;
    FLaunchPad: TddgLaunchPad; // Для создания резервной копии компонента
    FRunButtons: TRunButtons; // Ссылается на реальный компонент TRunButtons
    Modified: Boolean;
    procedure UpdatePathListBox;
  end;

  { Объявление потомка TPropertyEditor и переопределение требуемых методов. }
  TRunButtonsProperty = class(TPropertyEditor)
    function GetAttributes: TPropertyAttributes; override;
    function GetValue: string; override;
    procedure Edit; override;
  end;

{ Эта функция будет вызываться редактором свойства. }
function EditRunButtons(RunButtons: TRunButtons): Boolean;

implementation

{$R *.DFM}

function EditRunButtons(RunButtons: TRunButtons): Boolean;
{ Создает экземпляр диалогового окна TLaunchPadEditor для
  непосредственной модификации коллекции TRunButtons. }
begin
  with TLaunchPadEditor.Create(Application) do
    try
      FRunButtons := RunButtons; // Ссылается на действительный TRunButtons
      { Копирование компонентов TRunBtnItems в резервный экземпляр
        FLaunchPad, который будет использован, если пользователь
        отменит операцию редактирования. }
      FLaunchPad.RunButtons.Assign(RunButtons);
      { Отображение в окне списка компонентов TRunBtnItem. }
      UpdatePathListBox;
      ShowModal; // Отображает форму
      Result := Modified;
    finally
```

```
      Free;
    end;
end;

{ TLaunchPadEditor }

procedure TLaunchPadEditor.FormCreate(Sender: TObject);
begin
  { Создает резервные экземпляры компонента TLaunchPad для использования
    в том случае, если пользователь отменит редактирование
    элементов TRunBtnItem. }
  FLaunchPad := TddgLaunchPad.Create(Self);

  { Создает экземпляр компонента TddgRunButton и выравнивает его
    относительно содержащей его панели.}
  TestRunBtn := TddgRunButton.Create(Self);
  TestRunBtn.Parent := pnlRBtn;

  TestRunBtn.Width  := pnlRBtn.Width;
  TestRunBtn.Height := pnlRBtn.Height;
end;

procedure TLaunchPadEditor.FormDestroy(Sender: TObject);
begin
  TestRunBtn.Free;
  FLaunchPad.Free; // Освобождает экземпляр TLaunchPad
end;

procedure TLaunchPadEditor.PathListBoxClick(Sender: TObject);
{ Если пользователь щелкает на элементе списка TRunBtnItem,
  тестируется компонент TRunButton, соответствующий выбранному элементу.}
begin
  if PathListBox.ItemIndex > -1 then
    TestRunBtn.CommandLine := PathListBox.Items[PathListBox.ItemIndex];
end;

procedure TLaunchPadEditor.UpdatePathListBox;
{ Повторная инициализация компонента PathListBox для
  обновления списка элементов TRunBtnItem. }
var
  i: integer;
begin
  PathListBox.Clear; // Вначале нужно очистить окно списка
  for i := 0 to FRunButtons.Count - 1 do
    PathListBox.Items.Add(FRunButtons[i].CommandLine);
end;

procedure TLaunchPadEditor.AddBtnClick(Sender: TObject);
{ Если был щелчок на кнопке Add, запускается в работу TOpenDialog
  для получения имени и пути выполняемого файла (этот файл добавляется
  в PathListBox), а также добавляется новый элемент FRunBtnItem. }
```

```
var
  OpenDialog: TOpenDialog;
begin
  OpenDialog := TOpenDialog.Create(Application);
  try
    OpenDialog.Filter := 'Executable Files|*.EXE';
    if OpenDialog.Execute then
    begin
      { Добавление в PathListBox. }
      PathListBox.Items.Add(OpenDialog.FileName);
      FRunButtons.Add; // Создание нового экземпляра TRunBtnItem
      { Новый элемент в списке PathListBox получает фокус. }
      PathListBox.ItemIndex := FRunButtons.Count - 1;
      { Установка командной строки для нового элемента TRunBtnItem
        равной имени файла, заданного значением PathListBox.ItemIndex. }
      FRunButtons[PathListBox.ItemIndex].CommandLine :=
        PathListBox.Items[PathListBox.ItemIndex];
      { Вызов обработчика события PathListBoxClick для тестирования
        компонента TRunButton, соответствующего вновь добавленному элементу. }
      PathListBoxClick(nil);
      Modified := True;
    end;
  finally
    OpenDialog.Free
  end;
end;

procedure TLaunchPadEditor.RemoveBtnClick(Sender: TObject);
{ Удаление выбранного пути/имени файла из PathListBox одновременно
  с удалением соответствующего элемента TRunBtnItem из экземпляра
  коллекции FRunButtons. }
var
  i: integer;
begin
  i := PathListBox.ItemIndex;
  if i >= 0 then
  begin
    PathListBox.Items.Delete(i);  // Удаление элемента из списка
    FRunButtons[i].Free;          // Удаление элемента из коллекции
    TestRunBtn.CommandLine := ''; // Удаление кнопки выполнения тестирования
    Modified := True;
  end;
end;

procedure TLaunchPadEditor.CancelBtnClick(Sender: TObject);
{ Если пользователь отменяет операцию, элементы TRunBtnItem резервного
  экземпляра LaunchPad копируются в исходный экземпляр TLaunchPad, а затем
  форма закрывается - после установки для ModalResult значения mrCancel. }
begin
  FRunButtons.Assign(FLaunchPad.RunButtons);
  Modified := False;
```

```
  ModalResult := mrCancel;
end;

{ TRunButtonsProperty }

function TRunButtonsProperty.GetAttributes: TPropertyAttributes;
{ Сообщает инспектору объектов о том, что данный редактор свойства
  будет использовать диалоговое окно. По щелчку на кнопке
  с многоточием в окне Object Inspector будет вызываться метод Edit. }
begin
  Result := [paDialog];
end;

procedure TRunButtonsProperty.Edit;
{ Вызов метода EditRunButton() и передача ему указателя на редактируемый
  экземпляр компонента TRunButton. Этот указатель получен с помощью
  метода GetOrdValue. Затем следует перерисовка диалогового окна
  LaunchDialog с использованием вызова метода TRunButtons.UpdateRunButtons. }
begin
  if EditRunButtons(TRunButtons(GetOrdValue)) then
    Modified;
  TRunButtons(GetOrdValue).UpdateRunButtons;
end;

function TRunButtonsProperty.GetValue: string;
{ Переопределение метода GetValue таким образом, чтобы тип класса
  редактируемого свойства отображался в окне инспектора объектов. }
begin
  Result := Format('(%s)', [GetPropType^.Name]);
end;

end.
```

В приведенном модуле сначала определяется диалоговое окно TLaunchPadEditor, а затем — редактор свойства TRunButtonsProperty, но мы в первую очередь рассмотрим редактор свойства, так как он и вызывает это диалоговое окно.

Редактор свойства TRunButtonsProperty мало отличается от редактора свойства с диалоговым окном, описанного выше. В нем переопределяются методы GetAttributes(), Edit() и GetValue().

Метод GetAttributes() просто возвращает значение типа TPropertyAttributes, определяющее, что данный редактор вызывает диалоговое окно, а также помещает кнопку с троеточием в окно инспектора объектов.

Метод GetValue() использует функцию GetPropType() для получения указателя на информацию о типах времени выполнения редактируемого свойства. Он возвращает поле имени этой информации, представляющее строку типа нашего свойства. Эта строка отображается внутри скобок в окне инспектора объектов, как это принято в Delphi.

Наконец, метод Edit() вызывает функцию EditRunButtons(), определенную в этом модуле. Ей в качестве параметра передается указатель на свойство TRunButtons (с помощью функции GetOrdValue). При возврате функции вызывается метод UpdateRunButtons() для перерисовки коллекции RunButtons, отражающей любые возможные в ней изменения.

Функция EditRunButtons() создает экземпляр компонента TLaunchPadEditor и помещает в его поле FRunButtons указатель на TRunButtons, переданный в нее в качестве параметра. Этот указатель используется для внесения изменений в коллекцию TRunButtons. Затем функция копирует коллекцию свойств TRunButtons во внутренний экземпляр компонента TddgLaunchPad с именем FLaunchPad. Функция использует этот экземпляр как резервный на тот случай, если пользователь отменит операцию редактирования.

Выше упоминалась возможность добавления кнопки Apply в диалоговое окно. Для этого можно вместо непосредственной модификации реальной коллекции отредактировать экземпляр коллекции RunButtons компонента FLaunchPad. Тогда, если пользователь отменит операцию редактирования, ничего не произойдет; если же он нажмет кнопку Apply или OK — внесенные изменения подтвердятся.

Конструктор формы Create() создает внутренний экземпляр компонента TddgLaunchPad. Деструктор Destroy() освобождает его перед уничтожением формы.

Метод PathListBoxClick() — обработчик события OnClick компонента PathListBox. Этот метод заставляет компонент TestRunBtn (тестовый компонент TddgRunButton) отображать элемент, выбранный в данный момент в списке PathListBox, и путь к выполняемому файлу. Пользователь может нажать кнопку этого экземпляра TddgRunButton для запуска приложения.

Метод UpdatePathListBox() инициализирует список PathListBox с элементами коллекции.

Метод AddButtonClick() — обработчик события OnClick кнопки Add. Этот обработчик события вызывает диалоговое окно File Open, чтобы принять от пользователя имя выполняемого файла. При этом в список PathListBox добавляется путь к этому файлу. Метод AddButtonClick() также создает экземпляр компонента TRunBtnItem в коллекции и присваивает командную строку свойству TRunBtnItems.CommandLine. Это значение тут же передается в компонент TddgRunButton, заключенный в данный компонент TRunBtnItems.

Метод RemoveBtnClick() — обработчик события OnClick кнопки Remove. Он удаляет выбранный элемент из списка PathListBox и экземпляр TRunBtnItem из коллекции.

Метод CancelBtnClick() — обработчик события OnClick кнопки Cancel. Он копирует резервную коллекцию из экземпляра FLaunchPad в действительную коллекцию TRunButtons и закрывает форму.

Объекты TCollection и TCollectionItem могут оказаться весьма полезными во многих отношениях. Внимательно их изучите, и как только вам потребуется хранить список компонентов, у вас уже будет готовое решение.

Резюме

В этой главе вы познакомились с более сложными методиками и приемами, используемыми при разработке компонентов в Delphi. Здесь были продемонстрированы способы расширения подсказки, анимированные компоненты, редакторы компонентов, редакторы свойств и коллекции компонентов. Вооружившись всей этой информацией, но не забывая при этом основ, изложенных в главе 21, “Создание пользовательских компонентов в Delphi”, вы сможете разработать любой необходимый вам компонент. Прочитав следующую главу — “COM-ориентированные технологии”, вы сможете еще глубже погрузиться в мир компонентно-ориентированного программирования.

COM-ориентированные технологии

Глава

23

Основы COM 198

Object Pascal и COM 202

COM-объекты и фабрики классов 210

Агрегирование 215

Распределенная модель COM 215

Автоматизация 215

Использование более сложных технологий автоматизации 244

Сервер транзакций Microsoft (MTS) 275

Класс TOleContainer 301

Резюме 313

Полная поддержка COM-ориентированных технологий — одно из основных достоинств Delphi. Под термином *COM-ориентированные технологии* подразумевается набор разнообразных технологий, опирающихся на модель COM как на некий фундамент. В этот набор входят такие технологии, как серверы и клиенты COM, элементы управления ActiveX, связывание и внедрение объектов (Object Linking and Embedding, или OLE), автоматизация (Automation) и сервер транзакций Microsoft (Microsoft Transaction Server, или MTS). Однако эти супермодные технологии могут также несколько озадачить и даже обескуражить разработчика. В этой главе речь пойдет о технологиях, основанных на COM, ActiveX и OLE, а также об их использовании при разработке приложений. Еще несколько лет назад рассмотрение подобных тем касалось, в основном, технологии OLE, которая предоставляет метод совместного использования данных различными приложениями, главным образом, посредством внедрения или связывания данных, ассоциированных с одним приложением, с данными, ассоциированными с другим приложением (например, внедрение электронной таблицы в документ текстового процессора). Однако COM — нечто гораздо большее, чем просто OLE-ориентированные трюки с текстовым процессором!

В этой главе рассматриваются основы COM-ориентированных технологий, в частности — расширения языка Object Pascal и библиотеки VCL, вызванные необходимостью поддержки COM-технологий. Вы узнаете, как применить эти новые средства для управления серверами автоматизации из своих приложений Delphi и как создать собственный сервер автоматизации. Вы также получите представление о высокоорганизованных методах автоматизации и MTS. Наконец, вы познакомитесь с описанием VCL-класса `TOleContainer`, который инкапсулирует ActiveX-контейнеры. Необходимо отметить, что в одной главе невозможно осветить все подробности технологий ActiveX и OLE (это тема для отдельной книги или даже нескольких книг), тем не менее здесь представлены наиболее важные свойства этих технологий и их реализация в Delphi.

Основы COM

Прежде чем приступить к изучению этой темы, читателю необходимо понять основные концепции и терминологию, используемую при описании технологий COM. В этом разделе рассматриваются основные идеи и термины, связанные с новыми технологиями.

COM: Component Object Model

Component Object Model (модель многокомпонентных объектов) — COM — представляет собой “фундамент”, на котором построены технологии ActiveX и OLE. COM определяет API- и двоичный стандарты для связи объектов, не зависящих от языка программирования или платформы (теоретически). COM-объекты подобны уже известным вам VCL-объектам, за исключением того, что они содержат специализированные методы и свойства, а не поля данных.

COM-объект имеет один или несколько *интерфейсов* (interface), которые, по сути, представляют собой таблицы функций, связанных с этим объектом. Методы интерфейса можно вызывать аналогично методам любого объекта Delphi. Более подробно интерфейсы будут описаны ниже в этой главе.

Используемые объекты компонентов могут быть реализованы в любом файле `.exe` или `.dll`. При этом реализация объекта остается для вас, как для пользователя объекта, совершенно прозрачной, благодаря предлагаемому COM сервису, называемому *маршалингом* (marshalling). Механизм маршалинга COM берет на себя всю сложность организации вызова функций между независимыми процессами и даже различными компьютерами, благодаря че-

му становится возможным использование 32-разрядных объектов в 16-разрядных приложениях или доступ к объекту, расположенному на компьютере А, из приложения, запущенного на компьютере Б. Такое межкомпьютерное взаимодействие называется *распределенной COM* (Distributed COM — DCOM). Более детальное описание этого механизма взаимодействия приводится ниже в этой главе.

COM – ActiveX – OLE

"Так в чем же различие между COM, ActiveX и OLE?" — один из наиболее частых (и обоснованных) вопросов, которые задают разработчики при знакомстве с этими технологиями. Резонность этого вопроса объясняется еще и тем, что создатель данных технологий, фирма Microsoft, не прилагает особых усилий, чтобы разъяснить их суть. Выше уже упоминалось, что COM — это API- и двоичный стандарты, которые служат основой для всех остальных "кирпичиков" этого семейства технологий. В 1995 году аббревиатура OLE была общим термином, использовавшимся для описания целого набора технологий, основанных на архитектуре COM. Тогда термин "OLE" использовался для обозначения только тех технологий, которые непосредственно имели дело со связыванием и внедрением, а именно: использование контейнеров и серверов, активизация по месту вставки или внедрения, технология "перетащить и опустить", слияние меню. В 1996 году Microsoft начинает агрессивную маркетинговую кампанию по продвижению в язык разработчиков термина *ActiveX*. ActiveX становится всеобъемлющим термином, используемым для описания не OLE-технологий, основанных на использовании COM. Технология ActiveX включает автоматизацию (ранее называвшуюся *OLE-автоматизацией*), управляющие элементы, документы, контейнеры, сценарии и некоторые Internet-технологии. Поскольку появляется неразбериха, вызванная стремлением использовать термин "ActiveX" для описания всех "семейных любимцев", компания Microsoft слегка "дала задний ход" и сейчас иногда называет любые отличные от OLE технологии, основанные на COM, просто и незатейливо — *COM-ориентированные технологии* (COM-based technologies).

В компьютерной индустрии критический взгляд на творчество этой фирмы получил свое выражение в следующей фразе: мы говорим "OLE" — подразумеваем "замедление работы и увеличение размера приложений". В результате для маркетинговых решений фирме Microsoft потребовалась новая терминология, предназначенная для новых интерфейсов API, которые были положены в основу будущих операционных систем и Internet-технологий. Еще один забавный факт: Microsoft просит называть OLE не *Object Linking and Embedding*, а просто *О-Ле*!

Терминология

COM-технология принесла с собой новую терминологию; поэтому прежде чем погружаться в глубины ActiveX и OLE следует разобраться со значением некоторых терминов.

Экземпляр COM-объекта обычно называют просто *объектом*, а тип, который идентифицирует этот объект, — *компонентным классом* (component class, или coclass). Поэтому для создания экземпляра некоторого COM-*объекта* нужно предоставить идентификатор COM-*класса* (CLSID).

Часть данных, которая совместно используется несколькими приложениями, называется *OLE-объектом* (OLE object). Приложения, способные содержать OLE-объекты, называются *OLE-контейнерами* (OLE container), а приложения, способные содержать собственные данные в OLE-контейнерах, называются *OLE-серверами* (OLE server).

Документ, который содержит один или более OLE-объектов, обычно называют *составным документом* (compound document). Хотя OLE-объекты могут содержаться внутри отдельных документов, полноценные приложения, с которыми можно работать в контексте другого документа, называют *ActiveX-документами* (ActiveX document).

Как следует из названия технологии, OLE-объект может быть *связанным* или *внедренным* в составной документ. Связанные объекты сохраняются в отдельном файле на диске. Благодаря средствам связывания объектов, несколько контейнеров — или даже приложение-сервер — может быть связано с одним и тем же OLE-объектом, расположенным на диске. Если одно из приложений модифицирует связанный объект, внесенные изменения распространяется на все приложения, связанные с данным объектом. Внедренные объекты сохраняются непосредственно в приложениях, являющихся OLE-контейнерами. Только контейнерное приложение будет способно поддерживать редактирование внедренного OLE-объекта. Внедрение не позволяет другим приложениям иметь доступ к импортированным данным (а следовательно, модифицировать их или разрушать), но при этом существенно усложняется управление данными, помещенными в контейнер.

Еще один аспект ActiveX, о котором речь пойдет дальше в этой главе, называется *автоматизацией* (automation). Это средство позволяет приложениям (называемым *контроллерами автоматизации* — automation controller) управлять объектами, ассоциированными с другими приложениями или динамическими библиотеками (называемыми *сервером автоматизации* — automation server). Автоматизация позволяет управлять объектами в другом приложении и, наоборот, — предоставлять функциональные элементы своего приложения другим разработчикам.

Достоинства ActiveX

Самым замечательным свойством технологии ActiveX является простота внедрения средств управления многими типами данных в приложение. Слово “простота” может вызвать у вас улыбку, но это правда! Например, гораздо проще наделить разрабатываемое приложение возможностью использования ActiveX-объектов, чем самостоятельно создать средства, которые обычно применяются при работе с текстовым процессором, электронными таблицами или графическими изображениями.

Технология ActiveX прекрасно вписывается в традиции Delphi многократно использовать уже созданный программный код. Не нужно заново писать программу для управления конкретным типом данных, если уже создано работоспособное приложение OLE-сервера. Какой бы сложной не показалась технология OLE, она все же лучше других альтернативных решений.

Не секрет, что фирма Microsoft инвестировала значительные средства в технологию ActiveX, и теперь разработчики Windows 95/98, Windows NT и последующих операционных систем этой серии вынуждены ближе познакомиться с технологией ActiveX с целью использовать все ее преимущества в своих приложениях. Причем, нравится это кому-то или нет, но модель COM следует воспринимать как объективную реальность и постараться сделать ее удобным помощником в разработке своих приложений.

Стандарты OLE 1 и OLE 2

Одно из основных различий между OLE-объектами, ассоциированными с 16-разрядными серверами OLE 1 и с OLE 2, заключается в способе их активизации. Когда активизируется объект, созданный для сервера OLE 1, запускается и получает фокус ввода приложение-сервер, а OLE-объект появляется в нем в готовом для редактирования виде. Когда активизируется объект OLE 2, приложение-сервер OLE 2 становится активным неявно, “внутри” приложения-контейнера. Это называется *активизацией по месту вставки* (in-place activation), или *визуальным редактированием* (visual editing).

При активизации объекта OLE 2 меню и панели инструментов приложения-сервера заменяют или сливаются с соответствующими элементами приложения-клиента, а часть окна приложения-клиента, по сути, становится окном приложения-сервера. Этот процесс демонстрируется на конкретном примере, приведенном ниже в этой главе.

Структурированное хранилище

Стандарт OLE 2 определяет схему хранения информации на диске, известную как *структурированное хранилище* (structured storage). Эта схема предполагает реализацию на файловом уровне тех же функций, которые в среде DOS обеспечивались при работе с файлами на уровне диска. Структурированное хранилище представляет собой один физический файл на диске, но внутри этот файл подобен каталогу DOS, состоящему из множества других хранилищ (эквивалентных подкаталогам) и потоков (эквивалентных файлам DOS). Иногда структурированные хранилища называют *составными файлами* (compound files).

Единообразная передача данных

В OLE 2 реализована концепция объекта данных, который является базовым объектом, используемым для обмена данными с соблюдением некоторых правил единообразной передачи информации. Принципы *единообразной передачи данных* (uniform data transfer — UDT) определяют передачу данных через буфер обмена (Clipboard) и реализованы в механизмах "перетащить и опустить", DDE и OLE. Объекты данных предоставляют б*о*льшие возможности описания типов содержащихся в них данных, по сравнению с теми, что имелись прежде, недостаточность которых практически выражалась в ограничениях, свойственных существовавшим ранее средствам передачи данных. Фактически, технология UDT предназначена для замены технологии DDE. Объект данных может в процессе работы учитывать значения собственных свойств, таких как размер, цвет и даже тип устройства, в котором он отображается. Попробуйте-ка реализовать вышесказанное в буфере обмена Windows!

Потоковые модели

Каждый COM-объект работает с отдельной потоковой моделью, которая определяет, каким образом объект может функционировать в многопоточной среде. После регистрации COM-сервера в системе каждый из COM-объектов, содержащихся в этом сервере, должен зарегистрировать поддерживаемую им потоковую модель. Для COM-объектов, созданных в Delphi, потоковая модель выбирается при выполнении последовательности действий с помощью мастеров создания объектов автоматизации, элементов управления ActiveX или COM-объектов, тем самым определяется, как будет зарегистрирован элемент управления. Существуют следующие потоковые модели COM.

- *Однопоточная* (single). Весь COM-сервер выполняется в одном потоке.
- *Раздельная* (apartment). Иначе эту модель называют *однопоточно-раздельной* (single-threaded apartment — STA). Каждый COM-объект выполняется в контексте собственного потока и несколько экземпляров одинаковых типов COM-объекта могут выполняться в отдельных потоках. В результате любые данные, распределенные между экземплярами объектов (например, глобальные переменные), должны быть при необходимости защищены объектами синхронизации потоков.
- *Свободная* (free). Иначе эту модель называют *многопоточно-раздельной* (multithreading apartment — MTA). Клиент может вызвать метод объекта в любом потоке и в любое время. В этом случае COM-объект должен защищать даже свои собственные данные (а не только глобальные) от одновременного доступа из нескольких потоков.
- *Обе модели*. Поддерживаются обе потоковые модели ("раздельная" и "свободная").

Необходимо иметь в виду, что выбор желаемой потоковой модели в диалоговом окне мастера не гарантирует, что COM-объект будет защищен необходимым образом в соответствии с выбранной потоковой моделью. Для поддержки определенной потоковой модели необходимо написать код, гарантирующий, что COM-сервер будет работать с выбранной моделью надлежащим образом. Практически всегда при этом необходимо использовать объекты синхронизации потоков для защиты доступа к глобальным данным или данным экземпляров в COM-объектах. Подробные сведения о синхронизации потоков приводятся в главе 11, "Создание многопоточных приложений" (том I).

Стандарт COM+

Компания Microsoft реализовала свое самое значительное за последнее время обновление технологии COM в виде новой версии — составной части версии Windows 2000, — которую назвала *COM+*. Цель выпуска стандарта COM+ состоит в упрощении процесса разработки COM на основе интеграции нескольких технологий-сателлитов, из которых самыми значительными являются Microsoft Transaction Server, или MTS (подробнее о ней — ниже в этой главе), и Microsoft Message Queue (MSMQ). Интеграция этих технологий со стандартными динамическими средствами COM+ означает, что все разработчики COM+ смогут воспользоваться преимуществами таких средств, как управление транзакциями, поддержка безопасности, удаленное администрирование, обслуживание очереди компонентов, а также сервис, связанный с публикацией и подпиской на события. Поскольку модель COM+ состоит в основном из имеющихся в наличии частей, это означает полную обратную совместимость, в результате которой все существующие COM- и MTS-приложения автоматически становятся COM+-приложениями.

Object Pascal и COM

Кратко представив базовые концепции и терминологию технологий COM, ActiveX и OLE, можно переходить к рассмотрению способов реализации этих концепций в Delphi. В этом разделе более детально рассматривается как сама технология COM, так и ее согласование с языком Object Pascal и библиотекой VCL.

Интерфейсы

COM определяет стандарты для расположения в памяти функций объектов. Функции располагаются в *виртуальных таблицах* (virtual tables — vtables), т.е. таблицах адресов функций, аналогичных таблицам виртуальных методов (virtual method table — VMT) классов в Delphi. Описание каждой виртуальной таблицы в языке программирования называется *интерфейсом* (interface).

Рассмотрим интерфейс с точки зрения отдельного класса. Каждый его аспект можно представить как специфический набор функций или процедур, предназначенных для манипулирования классом. Например, COM-объект, представляющий растровое изображение, может поддерживать два интерфейса: один, содержащий методы, которые позволяют воспроизводить рисунок на экране монитора и распечатывать его, а другой, управляющий записью и считыванием рисунка в файл или из файла на диске.

Реально любой интерфейс представляется двумя частями. Первая часть — это определение интерфейса, которое включает коллекцию, состоящую из одного или нескольких объявлений функций, расположенных в определенном порядке. Определение интерфейса используется как самим объектом, так и пользователем объекта. Вторая часть — реализация интерфейса, представляющая собой воплощение на практике функций, описанных в его определе-

нии. Определение интерфейса подобно контракту, заключенному между COM-объектом и использующим его клиентом — оно гарантирует клиенту, что данный объект реализует определенные методы, выстроенные в определенном порядке.

Введенное в Delphi 3 ключевое слово `interface` языка Object Pascal позволяет достаточно просто определять COM-интерфейсы. Объявление интерфейсов семантически подобно объявлению классов, за исключением того, что интерфейсы могут содержать только свойства и методы, но не данные. Поскольку интерфейсы не могут содержать данные, их свойства должны записываться и считываться только с помощью методов. Однако самое важное то, что интерфейсы не имеют секции реализации, поскольку они лишь определяют контракт.

Интерфейс `IUnknown`

Подобно тому, как все классы Object Pascal являются неявными потомками класса `TObject`, все COM-интерфейсы (а следовательно, и все интерфейсы Object Pascal) неявно выведены из интерфейса `IUnknown`. Интерфейс `IUnknown` определяется в модуле `System` следующим образом:

```
type
  IUnknown = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```

Как видно из приведенного фрагмента кода, помимо ключевого слова `interface`, между объявлениями интерфейса и класса существует еще одно явное различие, заключающееся в присутствии *глобально-уникального идентификатора* (Globally Unique Identifier — GUID), используемого в технологии COM.

Глобально-уникальные идентификаторы (GUID)

GUID представляет собой 128-разрядное целое число, используемое в технологии COM для уникальной идентификации интерфейсов, компонентных классов и других объектов. GUID практически гарантирует настоящую глобальную уникальность благодаря использованию чисел достаточно большой размерности и превосходного алгоритма их генерации. GUID генерируется с помощью API-функции `CoCreateGUID()`, а алгоритм генерации нового GUID основан на комбинации из следующей информации: текущая дата и время, частота процессора, номер сетевой карты, остаток на банковском счете Билла Гейтса (ну, хорошо, пусть, со счетом мы несколько погорячились). Если на компьютере установлена сетевая карта, сгенерированный на этом компьютере GUID будет действительно уникальным, поскольку уникальность каждой сетевой карты гарантируется встроенным в нее глобальным идентификатором (ID). Если же на компьютере нет сетевой карты, ее номер можно заменить другим, синтезировав его с помощью параметров другого установленного в компьютере оборудования.

Поскольку не существует языкового типа, позволяющего хранить 128 двоичных разрядов, для представления GUID используется запись с типом `TGUID`, которая определяется в модуле `System` следующим образом:

```
PGUID = ^TGUID;
TGUID = record;
    D1: LongWord;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
end;
```

Поскольку присваивать переменным и константам значения GUID в этом формате записи достаточно сложно, Object Pascal также позволяет определить запись `TGUID` как строку следующего формата:

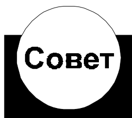
```
'{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}'
```

Благодаря этому следующие объявления эквивалентны:

```
MyGuid: TGUID = (
  D1:$12345678;D2:$1234;D3:$1234;D4:($01,$02,$03,$04,$05,$06,$07,$08));

MyGuid: TGUID = '{12345678-1234-1234-12345678}';
```

В COM каждый интерфейс или класс имеет сопровождающий GUID, который является уникальным определителем интерфейса. В этом случае два интерфейса или класса, имеющих одинаковые имена и созданных двумя независимыми разработчиками, никогда не будут конфликтовать, поскольку соответствующие им GUID всегда будут различны. При определении интерфейса GUID обычно называют *идентификатором интерфейса* (interface ID — IID), а при определении класса GUID обычно называют *идентификатором класса* (class ID — CLSID).



Для генерации нового GUID в среде Delphi необходимо нажать комбинацию клавиш <Ctrl+Shift+G> в окне редактора кода.

Помимо своего идентификатора IID, в интерфейсе `IUnknown` объявлены три метода: `QueryInterface()`, `_AddRef()` и `_Release()`. Поскольку интерфейс `IUnknown` является базовым интерфейсом COM, все остальные интерфейсы должны реализовать интерфейс `IUnknown` и его методы. Метод `_AddRef()` следует вызывать в том случае, если клиент получает и желает использовать указатель на данный интерфейс. Вызов этого метода должен сопровождаться вызовом метода `_Release()`, когда клиент заканчивает работу с интерфейсом. В этом случае объект, реализующий данный интерфейс, сможет поддерживать счетчик клиентов, хранящих ссылку на этот объект, или вести *подсчет ссылок* (reference count). Когда число ссылок станет равным нулю, объект должен будет выгрузить самого себя из памяти. Функция `QueryInterface()` используется для выполнения запроса о том, поддерживает ли данный объект некоторый интерфейс. Если требуемый интерфейс поддерживается, то клиенту возвращается указатель на него. Предположим, что объект `O` поддерживает интерфейсы `I1` и `I2`, и у клиента уже имеется указатель на интерфейс `I1` объекта `O`. Для получения от объекта `O` указателя на его интерфейс `I2` необходимо вызвать метод `I1.QueryInterface()`.

Опытный COM-разработчик может заметить, что символ подчеркивания перед методами `_AddRef()` и `_Release()` не используется в других языках программирования или даже в документации Microsoft по COM. Поскольку Object Pascal "знает" интерфейс `IUnknown`, эти методы нельзя вызывать напрямую (об этом чуть позднее), так что символ подчеркивания существует главным образом для того, чтобы привлечь внимание разработчика и заставить его задуматься, прежде чем выполнять вызов этих методов.

Поскольку каждый интерфейс в Delphi косвенно происходит от интерфейса `IUnknown`, каждый класс Delphi, реализующий интерфейсы, должен также поддерживать этих три метода интерфейса `IUnknown`. Эту "грязную работу" можно выполнить вручную, а можно поручить ее подпрограммам библиотеки VCL, сделав свой класс потомком класса `TInterfacedObject`, в котором интерфейс `IUnknown` уже реализован.

Использование интерфейсов

В главе 2, “Язык программирования Object Pascal” (том I) и в документации Delphi описывается семантика использования интерфейсов, поэтому приводить ее здесь не имеет смысла. Вместо этого рассмотрим, как интерфейс IUnknown неявно интегрируется с языком Object Pascal.

Когда переменной интерфейса присваивается значение, компилятор автоматически генерирует вызов метода интерфейса _AddRef(), чтобы увеличить содержимое счетчика ссылок. Когда переменная интерфейса выходит за пределы области видимости или принимает значение nil, компилятор автоматически генерирует вызов метода интерфейса _Release(). Рассмотрим следующий фрагмент кода:

```
var
  I: ISomeInterface;       // Некоторый интерфейс
begin
  I := FunctionThatReturnsAnInterface;  // Эта функция возвращает интерфейс
  I.SomeMethod;            // Вызов некоторого метода интерфейса
end;
```

А сейчас обратите внимание на следующий код, который вводит разработчик (выделен полужирным шрифтом), и Pascal-код, генерируемый компилятором (обычный шрифт) при реализации приведенного ранее фрагмента:

```
var
  I: ISomeInterface;
begin
  // Интерфейс автоматически инициализируется равным nil
  I := nil;
  try
    // Ваш код должен быть здесь
    I := FunctionThatReturnsAnInterface;
    // Метод _AddRef() вызывается неявно при присваивании I значения
    I._AddRef;
    I.SomeMethod;
  finally
    // Неявный блок завершения гарантирует, что ссылки
    // на интерфейс будут удалены
    if I <> nil I._Release;
  end;
end;
```

Компилятор Delphi также достаточно интеллектуален и “знает”, когда вызывать методы _AddRef() и _Release(). Это делается при переназначении интерфейсов экземплярам другого интерфейса или при присваивании интерфейсам значения nil. Рассмотрим, к примеру, такой программный блок:

```
var
  I: ISomeInteface;       // Некоторый интерфейс
begin
  // Назначение I
  I := FunctionThatReturnsAnInterface;  // Получение некоторого интерфейса
  I.SomeMethod;           // Вызов некоторого метода этого интерфейса
```

```
  // Переназначение I
  I := OtherFunctionThatReturnsAnInterface;  // Другой интерфейс
  I.OtherMethod;          // Вызов некоторого метода другого интерфейса
  // Установка I в nil
  I := nil;
end;
```

И снова оцените комбинацию, состоящую из кода, написанного пользователем (полужирный шрифт), и кода, сгенерированного компилятором (обычный шрифт):

```
var
  I: ISomeInterface;
begin
  // Интерфейс автоматически инициализируется в nil
  I := nil;
  try
    // Ваш код должен быть здесь
    // Назначение I
    I := FunctionThatReturnsAnInterface;
    // Метод _AddRef() вызывается неявно при присваивании I
    I._AddRef;
    I.SomeMethod;
    // Переназначение I
    I._Release
    I := OtherFunctionThatReturnsAnInterface;
    I._AddRef;
    I.OtherMethod;
    // Установка I в nil
    I._Release
    I := nil;
  finally
    // Неявный блок завершения гарантирует, что ссылка
    // на интерфейс будет удалена
    if I <> nil I._Release;
  end;
end;
```

Приведенный пример помогает понять, почему в Delphi используется символ подчеркивания в методах `_AddRef()` и `_Release()`. Об увеличении или уменьшении числа ссылок интерфейса часто забывали, и это было классической ошибкой при программировании с применением технологии COM в те времена, когда не было ключевого слова `interface`. Интерфейсы Delphi как раз и призваны помочь разработчикам избавиться от такого рода проблем — путем неявного вызова этих методов.

Поскольку компилятор "знает", как и когда генерировать вызовы методов `_AddRef()` и `_Release()`, резонно предположить, что он "знает" и о третьем методе интерфейса `IUnknown`, `QueryInterface()`. Конечно же, это так. Получив указатель на интерфейс от некоторого объекта, можно использовать оператор `as` для "приведения" его к типу другого интерфейса, который поддерживается COM-объектом. Понятие "приведение типа" наилучшим образом подходит для описания этого процесса, хотя такое применение оператора `as` является на самом деле не приведением типа в точном смысле этого выражения, а внутренним обращением к методу `QueryInterface()`. Вот как выглядит демонстрация описанного процесса:

```
var
  I1: ISomeInterface;          // Некоторый интерфейс
  I2: ISomeOtherInterface;     // Некоторый другой интерфейс
begin
  // Назначение I1
  I1 := FunctionThatReturnsAnInterface;  // Функция возвращает интерфейс
  // Метод QueryInterface I1 для интерфейса I2
  I2 := I1 as ISomeOtherInterface;  // Приведение к типу другого интерфейса
end;
```

В этом примере оператором as исключительная ситуация порождается в случае, если объект, на который ссылается интерфейс I1, не поддерживает интерфейс ISomeOtherInterface.

Одно дополнительное языковое правило, касающееся интерфейсов, заключается в следующем: переменные интерфейса совместимы по присвоению с классом Object Pascal, который реализует этот интерфейс. Например, рассмотрим следующие объявления интерфейса и класса:

```
type
  IFoo = interface
    // Определение IFoo
  end;

  IBar = interface(IFoo)
    // Определение IBar
  end;

  TBarClass = class(TObject, IBar)
    // Определение TBarClass
  end;
```

При таких объявлениях следующий код является вполне корректным:

```
var
  IB: IBar;
  TB: TBarClass;
begin
  TB := TBarClass.Create;
  try
    // Получение указателя на интерфейс IBar объекта TB:
    IB := TB;
    // Использование объекта TB и интерфейса IB
  finally
    IB := nil;    // Явное освобождение интерфейса IB
    TB.Free;
  end;
end;
```

Несмотря на то что такой подход, казалось бы, нарушает традиционные правила языка Pascal, связанные с совместимостью при выполнении операций присваивания, он делает интерфейсы более естественными и простыми в работе.

Существует важное, но не вполне очевидное следствие из этого правила — интерфейсы совместимы по присвоению только с классами, которые явно поддерживают этот интерфейс. Например, класс TBarClass, определенный выше, объявляет явную поддержку интерфейса IBar. Поскольку интерфейс IBar происходит от интерфейса IFoo, резонно было бы предположить, что интерфейс TBarClass также поддерживает интерфейс IFoo. Однако это вовсе не так, как иллюстрируется в приведенном ниже фрагменте.

```
var
  IF: IFoo;
  TB: TBarClass;
begin
  TB := TBarClass.Create;
  try
    // В следующей строке обнаружена ошибка компиляции,
    // поскольку класс TBarClass не поддерживает интерфейс IFoo явно
    IF := TB;
    // Использование объекта TB и интерфейса IF
  finally
    IF := nil;    // Явное освобождение интерфейса IF
    TB.Free;
  end;
end;
```

Интерфейсы и идентификаторы интерфейсов

Поскольку идентификатор интерфейса описывается как часть объявления интерфейса, компилятор Object Pascal знает о том, как получить его. Следовательно, можно передать тип интерфейса процедуре или функции, которой необходимы параметры типа TIID или TGUID. Предположим, существует функция, подобная следующей:

```
procedure TakesIID(const IID: TIID);
```

В этом случае приведенная ниже строка кода синтаксически правильна:

```
TakesIID(IUnknown);
```

Такая возможность предотвращает необходимость использования констант IID_*InterfaceType*, определенных для каждого типа интерфейса. Эти константы хорошо знакомы всем, кому приходилось иметь дело с разработкой или использованием COM-объектов на языке C++.

Псевдоним метода

Иногда при реализации нескольких интерфейсов в одном классе появляется еще одна проблема, заключающаяся в коллизии имен методов в одном или нескольких интерфейсах. Рассмотрим следующий фрагмент:

```
type
  IIntf1 = interface
    procedure AProc;
  end;
```

```
  IIntf2 = interface
    procedure AProc;
  end;
```

Каждый из описываемых интерфейсов содержит метод AProc(). Как в этом случае объявить класс с реализацией обоих интерфейсов? Оказывается, для этого можно использовать *псевдоним метода* (method aliasing). С помощью псевдонима метод интерфейса в классе можно поставить в соответствие методу с другим именем. Рассмотрим фрагмент, демонстрирующий объявление класса, в котором реализованы интерфейсы IIntf1 и IIntf2:

```
type
  TNewClass = class(TInterfacedObject, IIntf1, IIntf2)
  protected
    procedure IIntf2.AProc = AProc2;
    procedure AProc;    // Связывает с IIntf1.AProc
    procedure AProc2;   // Связывает с IIntf2.AProc
  end;
```

В этом объявлении метод AProc() интерфейса IIntf2 отображается на метод с именем AProc2(). Создание псевдонима в этом случае позволяет реализовать любой интерфейс в любом классе без конфликта имен методов.

Тип возвращаемого значения HResult

Вероятно, вы обратили внимание на то, что метод QueryInterface() интерфейса IUnknown возвращает результат типа HResult. Это наиболее популярный тип значений, возвращаемых многими методами различных интерфейсов ActiveX и OLE, а также функций COM API. Тип HResult определяется в модуле System как тип LongWord. Возможные значения типа HResult перечисляются в модуле Windows (если у вас есть исходный код библиотеки VCL, описания значений можно найти под заголовком { HRESULT value definitions }). Значение S_OK или NOERROR (0) типа HResult говорит об успешном выполнении. Если же старший бит значения типа Hresult установлен равным единице, значит, при выполнении произошла ошибка. В модуле Windows есть две функции — Succeeded() и Failed(), — которые принимают значение типа HResult в качестве параметра и возвращают значение типа BOOL, означающее успешное выполнение или наличие сбоя. Синтаксис вызова этих методов имеет следующий вид:

```
if Succeeded(FunctionThatReturnsHResult) then
  // Продолжение, если все нормально

if Failed(FunctionThatReturnsHResult) then
  // Код обработки ошибки
```

Естественно, проверка значения, возвращаемого при каждом отдельном вызове функции, — скучное занятие. Кроме того, работа над ошибками, возвращаемыми функциями, относится к "компетенции" методов обработки исключительных ситуаций Delphi. А потому в модуле ComObj определяется процедура OleCheck(), с помощью которой ошибки типа HResult преобразуются в исключительные ситуации. Синтаксис вызова этого метода имеет вид:

```
OleCheck(FunctionThatReturnsHResult);
```

Эта процедура удобна в использовании, благодаря чему значительно упрощается чтение кода программы.

COM-объекты и фабрики классов

Кроме поддержки одного или нескольких интерфейсов, которые происходят от интерфейса IUnknown, и реализации подсчета ссылок для отслеживания своего срока существования, COM-объекты имеют еще одну специфическую особенность: они создаются специальными объектами, называемыми *фабриками классов* (class factories). Каждый COM-класс имеет соответствующую фабрику класса, которая отвечает за создание экземпляров объектов этого COM-класса. Фабрика класса — это специальный COM-объект, который поддерживает интерфейс IClassFactory. Этот интерфейс определяется в модуле ActiveX следующим образом:

```
type
  IClassFactory = interface(IUnknown)
    ['{00000001-0000-0000-C000-000000000046}']
    function CreateInstance(const unkOuter: IUnknown; const iid: TIID;
      out obj): HResult; stdcall;
    function LockServer(fLock: BOOL): HResult; stdcall;
  end;
```

Метод CreateInstance() вызывается для создания экземпляра COM-объекта, связанного с данной фабрикой класса. Параметр unkOuter этого метода указывает на управляющий интерфейс IUnknown, если объект создается в качестве некоего агрегата (понятие агрегирования, или сборки, будет разъяснено ниже в этой главе). Параметр iid содержит идентификатор интерфейса (IID), с помощью которого можно управлять объектом. Параметр obj должен содержать указатель на интерфейс, определенный параметром iid.

Метод LockServer() вызывается для хранения COM-сервера в памяти даже в том случае, если на сервер не ссылается ни один клиент. Если параметр fLock равен True, счетчик блокировок сервера увеличивается на единицу. Если же параметр fLock равен False, счетчик блокировок сервера уменьшается на единицу. Если в результате счетчик ссылок сервера окажется равен 0 и при этом нет использующих его клиентов, данный COM-сервер выгружается из памяти.

Классы TComObject и TComObjectFactory

В Delphi существует два класса, инкапсулирующие COM-объекты и фабрики классов, — они называются соответственно TComObject и TComObjectFactory. Класс TComObject содержит необходимую инфраструктуру для поддержки интерфейса IUnknown и создания объектов с помощью класса TComObjectFactory. Подобным же образом класс TComObjectFactory поддерживает интерфейс IClassFactory и “умеет” создавать объекты класса TComObject. Проще всего генерировать COM-объект с помощью мастера COM Object Wizard, пиктограмма которого имеется во вкладке ActiveX диалогового окна New Items. В листинге 23.1 приведен псевдокод модуля, сгенерированный этим мастером. Этот псевдокод иллюстрирует отношения между упомянутыми классами.

Листинг 23.1. Псевдокод модуля COM-сервера

```
unit ComDemo;

interface

uses ComObj;
```

```
type
  TSomeComObject = class(TComObject, поддерживаемые интерфейсы)
    Здесь объявляются методы класса и интерфейса
  end;

implementation

uses ComServ;

Здесь создается реализация объекта TSomeComObject

initialization
  TComObjectFactory.Create(ComServer, TSomeComObject,
    CLSID_TSomeComObject, 'ИмяКласса', 'Описание');
end;
```

Класс, производный от класса TComServer, объявляется и реализуется подобно большинству других VCL-классов. Параметры, переданные конструктору Create() класса TComObjectFactory, связывают этот потомок класса TComServer с соответствующим объектом TComObjectFactory. Первый параметр конструктора — объект TComServer. В качестве этого параметра практически всегда передается глобальный объект ComServer, объявленный в модуле ComServ. Второй параметр — это класс TComObject, который требуется связать с фабрикой класса, а третий параметр — это идентификатор CLSID COM-класса TComObject. Четвертый и пятый параметры — это строки имени и описания класса, используемые для описания COM-класса в системном реестре.

Экземпляр класса TComObjectFactory создается в модуле инициализации; в этом случае фабрика класса обязательно будет доступной для создания экземпляров COM-объекта сразу после загрузки COM-сервера. Выполнение загрузки COM-сервера зависит от его типа, т.е. от того, *внутренний* это сервер (DLL) или *внешний* (приложение).

Внутренние COM-серверы

Внутренний COM-сервер представляет собой библиотеку DLL, которая может создавать COM-объекты, предназначенные для использования в главном приложении. Этот тип COM-сервера называется внутренним, потому что он, как и библиотека DLL, относится к тому же процессу, что и основное приложение. Внутренний сервер должен экспортировать четыре стандартные функции:

```
function DllRegisterServer: HResult; stdcall;
function DllUnregisterServer: HResult; stdcall;
function DllGetClassObject (const CLSID, IID: TGUID; var Obj): HResult; stdcall;
function DllCanUnloadNow: HResult; stdcall;
```

Каждая из этих функций уже реализована в модуле ComServ, поэтому вам остается лишь убедиться в том, что эти функции добавлены в описания exports проекта.

На заметку

Подробный пример применения внутренних COM-серверов можно найти в главе 24, “Расширение оболочки Windows”. Этот пример демонстрирует создание расширений оболочки.

Функция DllRegisterServer()

Функция DllRegisterServer() вызывается для регистрации библиотеки DLL COM-сервера в системном реестре. Если просто экспортировать данный метод из приложения Delphi, как описывалось выше, подпрограммы библиотеки VCL последовательно опросят все COM-объекты этого приложения и зарегистрирует их в системном реестре. Для каждого COM-класса при регистрации COM-сервера в системном реестре создается раздел в следующей ветви:

```
HKEY_CLASSES_ROOT\CLSID\{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx}
```

Здесь серия *x...x* представляет собой идентификатор CLSID этого COM-класса. Для внутренних серверов создается также подраздел InProcServer32. Параметром по умолчанию в этом подразделе является полный путь к библиотеке DLL внутреннего сервера. На рис. 23.1 показан пример COM-сервера, зарегистрированного в системном реестре.

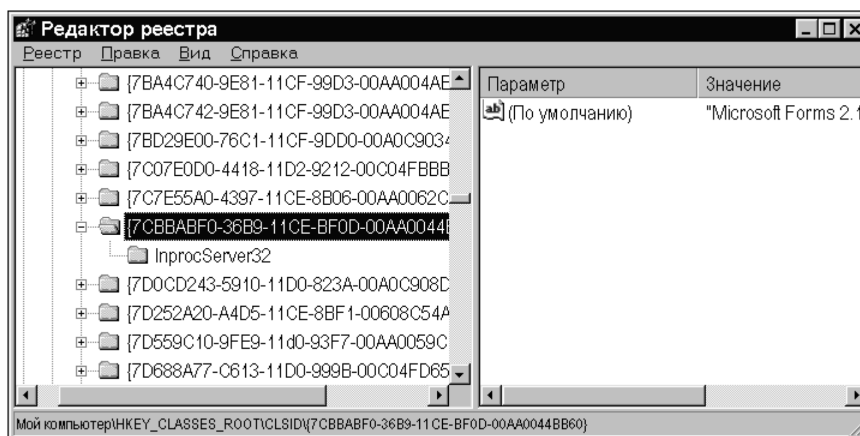


Рис. 23.1. COM-сервер в окне редактора системного реестра

Функция DllUnregisterServer()

Эта функция предназначена для отмены действий, выполненных функцией DllRegisterServer(). При вызове эта функция удаляет все разделы, подразделы и параметры, созданные функцией DllRegisterServer() в системном реестре.

Функция DllGetClassObject()

Функция DllGetClassObject() вызывается механизмом COM для получения фабрики класса конкретного COM-класса. Параметр CLSID этого метода — не что иное, как идентификатор CLSID типа создаваемого COM-класса. Параметр IID содержит значение идентификатора интерфейса (IID) указателя на экземпляр, который необходимо получить для объекта фабрики классов (обычно здесь передается идентификатор интерфейса IClassFactory). При корректном завершении функции параметр Obj содержит указатель на интерфейс фабрики классов, обозначенный параметром IID и способный создавать COM-объекты с типом класса, определенным параметром CLSID.

Функция DllCanUnloadNow()

Эта функция вызывается механизмом COM с целью определения, можно ли выгрузить из памяти библиотеку DLL данного COM-сервера. Если внутри этой библиотеки DLL существуют указатели на какой-либо COM-объект, функция возвращает значение S_FALSE, показывающее, что библиотеку выгрузить нельзя. Если же в библиотеке DLL не используется ни один из COM-объектов, функция возвращает значение S_TRUE.

Совет

Даже если все ссылки на COM-объекты внутреннего сервера уже освобождены, иногда не требуется вызывать функцию DllCanUnloadNow() для инициирования процесса выгрузки из памяти библиотеки DLL внутреннего сервера. Если желательно гарантированно выгрузить все неиспользуемые библиотеки DLL COM-сервера из памяти, вызовите функцию COM API CoFreeUnusedLibraries(), которая определяется в модуле ActiveX следующим образом:

```
procedure CoFreeUnusedLibraries; stdcall;
```

Создание экземпляра внутреннего COM-сервера

Для создания экземпляра COM-сервера в среде Delphi необходимо использовать функцию CreateComObject(), определяемую в модуле ComObj следующим образом:

```
function CreateComObject(const ClassID: TGUID): IUnknown;
```

Параметр ClassID содержит идентификатор CLSID, который определяет тип создаваемого COM-объекта. Значение, возвращаемое этой функцией, — интерфейс IUnknown требуемого COM-объекта. Если COM-объект не может быть создан, генерируется исключительная ситуация.

Функция CreateComObject() инкапсулирует функцию COM API CoCreateInstance(). Внутри функции CoCreateInstance() с целью получения интерфейса IClassFactory заданного COM-объекта вызывается функция COM API CoGetClassObject(). Функция CoCreateInstance() выполняет этот вызов путем поиска в системном реестре параметра InProcServer32 заданного COM-класса с целью определения пути к библиотеке DLL внутреннего сервера, вызова функции LoadLibrary() для загрузки этой библиотеки DLL и, наконец, вызова принадлежащей данной библиотеке DLL функции DllGetClassObject(). После получения указателя на интерфейс IClassFactory, для создания экземпляра объекта заданного COM-класса функция CoCreateInstance() вызывает функцию IClassFactory.CreateInstance().

Совет

Для создания с помощью фабрики классов нескольких объектов одного типа использовать функцию CreateComObject() неэффективно. Это объясняется тем, что после создания заданного COM-объекта эта функция освобождает указатель на интерфейс IClassFactory, полученный от функции CoGetClassObject(). Если необходимо создать несколько экземпляров одного и того же COM-объекта, вызовите непосредственно функцию CoGetClassObject(), а затем многократно обращайтесь к функции IClassFactory.CreateInstance() для создания необходимого количества объектов.

На заметку

Перед использованием любой функции COM или OLE API необходимо инициализировать библиотеку COM, вызвав функцию CoInitialize() с единственным параметром nil. Для правильного отключения библиотеки COM необходимо вызвать функцию CoUninitialize() (это же справедливо и в отношении библиотеки OLE). Эти вызовы подсчитываются системой, поэтому каждый вызов функции CoInitialize() в приложении должен сопровождаться вызовом функции CoUninitialize().

На заметку

В приложениях Delphi функция `CoInitialize()` вызывается автоматически из метода `Application.Initialize()`, а функция `CoUninitialize()` — при завершении работы модуля `ComObj`.

Нет необходимости вызывать эти функций из библиотек внутренних серверов, поскольку выполнение процедур инициализации и завершения возлагается на приложения-клиенты.

Внешние COM-серверы

Внешние COM-серверы представляют собой выполняемые файлы, которые могут создавать COM-объекты для использования в других приложениях. Серверы этого типа называются внешними, поскольку они выполняются вне процесса клиента, в контексте их собственного процесса.

Регистрация

Внешние серверы, как и их внутренние "родственники", должны быть зарегистрированы в системном реестре. Для каждого внешнего сервера должен быть создан раздел следующего вида:

```
HKEY_CLASSES_ROOT\CLSID\{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx}
```

Полный путь к выполняемому файлу внешнего сервера должен быть описан в параметре `LocalServer32` этого раздела.

COM-серверы приложений Delphi регистрируются в методе `Application.Initialize()`, который обычно вызывается в первой строке кода файла проекта приложения. Если при запуске приложения ему передается параметр командной строки `/regserver`, метод `Application.Initialize()` зарегистрирует COM-классы в системном реестре и немедленно прекратит выполнение приложения. Точно так же, если при запуске приложения указывается параметр командной строки `/unregserver`, метод `Application.Initialize()` удалит описание COM-классов из системного реестра и немедленно прекратит выполнение приложения. Если в командной строке не использован ни один из упомянутых ключей, метод `Application.Initialize()` зарегистрирует COM-классы в системном реестре и продолжит выполнение приложения в обычном режиме.

Создание экземпляра внешнего COM-сервера

В первом приближении процесс создания экземпляров COM-объектов из внешнего сервера напоминает эту процедуру для внутреннего сервера — достаточно просто вызвать функцию `CreateComObject()`, определенную в модуле `ComObj`. Однако, по сути, эти процессы абсолютно не похожи. В данном случае функция `CoGetClassObject()` просматривает системный реестр в поисках параметра `LocalServer32` и вызывает соответствующее приложение, используя функцию Win32 API `CreateProcess()`. При запуске внешнего приложения-сервера, последнее должно зарегистрировать свои фабрики классов, используя функцию COM API `CoRegisterClassObject()`. Эта функция добавляет указатель на интерфейс `IClassFactory` во внутреннюю таблицу зарегистрированных активных объектов COM-классов. В результате функция `CoGetClassObject()` получит возможность выбрать из этой таблицы необходимый указатель `IClassFactory` для создания требуемого экземпляра COM-объекта.

Агрегирование

Теперь мы знаем, что основными строительными блоками модели COM являются интерфейсы, а также то, что для интерфейсов возможно наследование. Однако интерфейсы являются сущностями, не имеющими реализации. Что же произойдет, если необходимо организовать реализацию одного COM-объекта внутри другого? Для ответа на этот вопрос в COM существует концепция *агрегирования* (aggregation). Под агрегированием подразумевается, что содержащий (внешний) объект создает содержащийся (внутренний) объект как часть процесса своего создания, при этом интерфейсы внутреннего объекта предоставляются клиентам внешним объектом. Объект должен позволить работать с собой как с неким агрегатом, предоставив средства перенаправления всех вызовов его методов `IUnknown` содержащему его внешнему объекту. Примером агрегирования COM-объектов в контексте библиотеки VCL может служить класс `TAggregatedObject` в модуле `AxCtrls`.

Распределенная модель COM

Предложенная в Windows NT 4 модель *распределенной COM* (Distributed COM — DCOM) предоставляет средства доступа к COM-объектам, расположенным на других компьютерах в сети. Кроме создания удаленных объектов, модель DCOM также предлагает простые средства обеспечения безопасности, позволяющие серверам определять, какие клиенты имеют право создавать экземпляры серверов и какие операции они могут выполнять. Операционные системы Windows NT 4 и Windows 98 содержат встроенную поддержку модели DCOM, а для Windows 95 необходимо загрузить с Web-узла фирмы Microsoft (`http://www.microsoft.com`) специальный модуль-приложение, который будет выполнять функции клиента DCOM.

Для создания удаленного COM-объекта необходимо использовать функцию `CreateRemoteComObject()`, объявленную в модуле `ComObj` следующим образом:

```
function CreateRemoteComObject(const MachineName: WideString;
  const ClassID: TGUID): IUnknown;
```

Первый параметр этой функции, `MachineName`, является строкой, определяющей сетевое имя компьютера, который содержит требуемый COM-класс. Параметр `ClassID` задает идентификатор CLSID создаваемого COM-объекта. Значение, возвращаемое этой функцией, — указатель на интерфейс `IUnknown` COM-объекта, определенного параметром `ClassID`. Если объект невозможно создать, генерируется исключительная ситуация.

Функция `CreateRemoteComObject()` инкапсулирует функцию COM API `CoCreateInstanceEx()`, которая является расширенной версией функции `CoCreateInstance()` и предназначена для создания удаленных объектов.

Автоматизация

Автоматизация (automation), ранее известная как *OLE-автоматизация* (OLE automation), позволяет приложениям или библиотекам DLL предоставлять свои программируемые объекты с целью их использования в других приложениях. Приложения или библиотеки DLL, которые предоставляют свои программируемые объекты, называются *серверами автоматизации* (automation server). Приложения, которые получают доступ к управлению программируемыми объектами, содержащимися в серверах автоматизации, называются *контроллерами автоматизации* (automation controller). Контроллеры автоматизации способны программировать сервер автоматизации посредством некоторого макроязыка, предлагаемого сервером.

Одно из основных преимуществ использования автоматизации в приложениях — независимость от языка. Контроллеры автоматизации могут управлять сервером независимо от языка, использовавшегося при разработке любого компонента. Кроме того, поскольку автоматизация поддерживается на уровне операционной системы, теоретически, в будущем можно будет легко использовать все вновь появившиеся возможности этой технологии, начав применение автоматизации уже сегодня. Если вам интересна эта тема, давайте продолжим изучение автоматизации. Ниже в этой главе рассматриваются вопросы создания серверов и контроллеров автоматизации в Delphi.

Если у вас есть проект с использованием автоматизации, созданный в Delphi 2, который требуется перенести в текущую версию Delphi, не забывайте, что применяемые для создания автоматизации методы коренным образом изменились, начиная с Delphi 3. Запомните, что нельзя смешивать подпрограммы модуля автоматизации `OleAuto` Delphi 2 с подпрограммами модулей `ComObj` и `ComServ`. Если необходимо скомпилировать проект автоматизации Delphi 2 в среде Delphi 5, модуль `OleAuto` следует поместить в папку `\Delphi5\lib\Delphi2` с целью организации поддержки обратной совместимости.

Интерфейс `IDispatch`

Объекты автоматизации — это, в сущности, просто COM-объекты, в которых реализован интерфейс `IDispatch`. Интерфейс `IDispatch` определяется в модуле `System` следующим образом:

```
type
  IDispatch = interface(IUnknown)
    ['{00020400-0000-0000-C000-000000000046}']
    function GetTypeInfoCount(out Count: Integer): Integer; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo):
      Integer; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): Integer; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer;
      Flags: Word; var Params; VarResult, ExcepInfo, ArgErr: Pointer): Integer;
  end;
```

Прежде всего следует отметить, что, для того чтобы воспользоваться преимуществами автоматизации в Delphi, вам вовсе не обязательно вникать во все детали интерфейса `IDispatch`, поэтому не стоит "сушить мозги", пытаясь разобраться во всех его сложностях. Вообще говоря, вам и не придется работать с этим интерфейсом напрямую, поскольку Delphi предлагает весьма элегантное решение — инкапсуляцию автоматизации. Описание интерфейса `IDispatch` в этом разделе предлагается только в качестве хорошей основы для понимания сути автоматизации.

Основной функцией интерфейса `IDispatch` является метод `Invoke()`. Когда клиент получает указатель `IDispatch` на сервер автоматизации, он может вызвать метод `Invoke()` для выполнения определенных методов на сервере. Параметр `DispID` этого метода содержит число, называемое *диспетчерским идентификатором* (dispatch ID), который показывает, какой метод должен быть вызван в сервере. Параметр `IID` не используется. Параметр `LocaleID` содержит информацию о языке. Параметр `Flags` определяет, какого вида метод будет вызван: обычный или метод доступа к свойствам. Свойство `Params` содержит указатель на массив `TDispParams`, который содержит параметры, передаваемые этому методу. Параметр `VarResult` — это указатель на

переменную типа OleVariant, в которую будет записано возвращаемое значение вызываемого метода. Параметр ExcepInfo является указателем на запись типа TExcepInfo, которая будет содержать информацию об ошибке, если метод Invoke() возвратит значение DISP_E_EXCEPTION. И, наконец, если метод Invoke() возвращает значение DISP_E_TYPEMISMATCH или DISP_E_PARAMNOTFOUND, то параметр ArgError — указатель на целое число — будет содержать индекс некорректного параметра в массиве Params.

Метод GetIDsOfNames() интерфейса IDispatch вызывается для получения диспетчерского идентификатора одного или нескольких имен методов, заданных в виде строк. Параметр IID этого метода не используется, параметр Names указывает на массив имен методов типа PWideChar. Параметр NameCount содержит число строк в массиве Names. Параметр LocaleID содержит информацию о языке. Последний параметр, DispIDs, является указателем на массив целых чисел NameCount, который метод GetIDsOfNames() заполнит диспетчерскими идентификаторами для методов, перечисленных в параметре Names.

Метод GetTypeInfo() возвращает информацию о типе объектов автоматизации (описываемую ниже в этой главе). Параметр Index определяет тип интересующей информации и обычно равен 0. Параметр LCID содержит информацию о языке. При успешном выполнении параметр TypeInfo будет содержать указатель ITypeInfo на информацию о типе объекта автоматизации.

Метод GetTypeInfoCount() в параметре Count возвращает число интерфейсов информации о типе, поддерживаемых объектом автоматизации. В настоящее время параметр Count может содержать только одно из двух возможных значений: 0, если объект автоматизации не поддерживает информацию о типе, и 1, если он ее поддерживает.

Информация о типе

Затратив много времени и усилий на создание сервера автоматизации, было бы большим разочарованием обнаружить, что потенциальный пользователь не сможет полностью использовать все возможности созданного сервера из-за недостаточного описания его свойств и методов в прилагаемой документации. К счастью, автоматизация предлагает средство для решения этих проблем — позволяет разработчикам ассоциировать с объектом автоматизации информацию о его типе. Информация о типе хранится в так называемых *библиотеках типов* (type library). Библиотека типов сервера автоматизации может быть добавлена к приложению-серверу или к его библиотеке DLL как ресурс либо храниться во внешнем файле. Библиотеки типов содержат информацию о классах, интерфейсах, типах данных и других компонентах сервера. Эта информация предлагается клиентам сервера автоматизации наряду с другой информацией, необходимой для создания экземпляров каждого из классов и корректного вызова методов каждого интерфейса.

Delphi самостоятельно генерирует библиотеки типов при добавлении объектов автоматизации в создаваемое приложение или библиотеку. Кроме того, Delphi знает, каким образом преобразовать информацию из библиотеки типов в данные Object Pascal так, чтобы можно было легко управлять сервером автоматизации из приложения Delphi.

Позднее и раннее связывание

Элементы автоматизации, которые рассматривались до сих пор в этой главе, работают на основе подхода, который носит название *позднего связывания* (late binding). В этом случае необходимый метод вызывается с помощью метода Invoke() интерфейса IDispatch. Под поздним связыванием подразумевается, что вызов метода невозможен до момента выполнения программы, поскольку требуемый адрес просто неизвестен. Во время компиляции вызов

метода автоматизации имеет вид вызова метода IDispatch.Invoke() с соответствующими параметрами, и лишь во время выполнения программы метод Invoke() вызовет указанный метод автоматизации. При вызове метода автоматизации с помощью типа Delphi Variant или OleVariant также используется позднее связывание, поскольку транслятор Delphi сначала должен будет организовать вызов метод IDispatch.GetIDsOfNames() для преобразования имени заданного метода в его параметр DispID, а затем уже реализовать вызов указанного метода посредством вызова метода IDispatch.Invoke() с полученным параметром DispID.

Оптимизация за счет раннего связывания состоит в разрешении параметров DispID для вызываемых методов еще во время компиляции, что позволит избежать во время выполнения приложения обращений к методу GetIDsOfNames(), прежде чем можно будет вызвать тот или иной требуемый метод. Такая оптимизация часто называется ID-связыванием (связыванием идентификаторов) и обычно осуществляется при вызове методов с использованием типа dispinterface Delphi.

Раннее связывание (early binding) происходит, когда объект автоматизации предоставляет свои методы посредством пользовательского интерфейса, потомка интерфейса IDispatch. В этом случае контроллеры автоматизации могут вызывать объекты автоматизации непосредственно с помощью виртуальных таблиц (vtable), т.е. без использования метода IDispatch.Invoke(). Поскольку вызов метода осуществляется напрямую, то доступ к методу происходит быстрее, чем при позднем связывании. Раннее связывание используется при вызове метода с помощью типа interface Delphi.

Об объекте автоматизации, который позволяет вызывать методы с помощью как метода Invoke(), так и потомков интерфейса IDispatch, говорят, что он поддерживает *двойной интерфейс* (dual interface). Объекты автоматизации, сгенерированные в среде Delphi, всегда поддерживают двойной интерфейс, а контроллеры автоматизации Delphi позволяют вызывать методы как с помощью метода Invoke(), так и напрямую — через интерфейс.

Регистрация

Разумеется, объекты автоматизации должны создать такие же записи в системном реестре, как и обычные COM-объекты:

```
HKEY_CLASSES_ROOT\CLSID\{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx}
```

Однако все серверы автоматизации должны создавать и дополнительный параметр ProgID, представляющий собой строковый идентификатор класса автоматизации. Кроме того, создается параметр в ветви HKEY_CLASSES_ROOT*(ProgID string)*, который содержит идентификатор CLSID класса автоматизации, позволяющий с помощью перекрестной ссылки вернуться к первой записи реестра в разделе CLSID.

Создание сервера автоматизации

Delphi существенно упрощает работу по созданию серверов автоматизации обоих типов — как внешнего, так и внутреннего. Процедура создания сервера автоматизации сводится к выполнению четырех действий.

1. Создайте приложение или библиотеку DLL, которые будут субъектом автоматизации. В качестве отправной точки можно даже использовать одно из уже существующих приложений, к которому будут добавлены средства поддержки автоматизации. Это единственное действие, где проявляется реальное различие между созданием внутреннего и внешнего серверов.

2. Создайте объект автоматизации и добавьте его к проекту. В Delphi существует эксперт по объектам автоматизации, который позволит пройти этот этап без особых затруднений.
3. Добавьте в объект автоматизации свойства и методы, пользуясь библиотекой типов. Именно эти свойства и методы будут предоставляться контроллерам автоматизации.
4. Реализуйте методы, сгенерированные Delphi по описанию в библиотеке типов.

Создание внешнего сервера автоматизации

В этом разделе речь пойдет о создании простого внешнего сервера автоматизации. Откройте новый проект и поместите в главную форму компоненты `TShape` и `TEdit`, как показано на рис. 23.2. Сохраните данный проект под именем `Srv.dpr`.

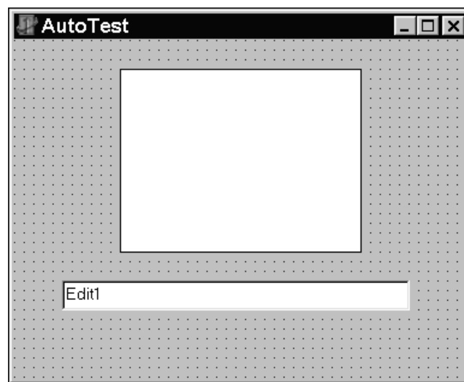


Рис. 23.2. Главная форма проекта `Srv`

Теперь добавим к проекту объект автоматизации. Для этого воспользуемся командой File⇨New. В раскрывшемся диалоговом окне New Items перейдите во вкладку ActiveX и дважды щелкните на пиктограмме Automation Object (рис. 23.3). Откроется диалоговое окно мастера Automation Object Wizard, показанное на рис. 23.4.

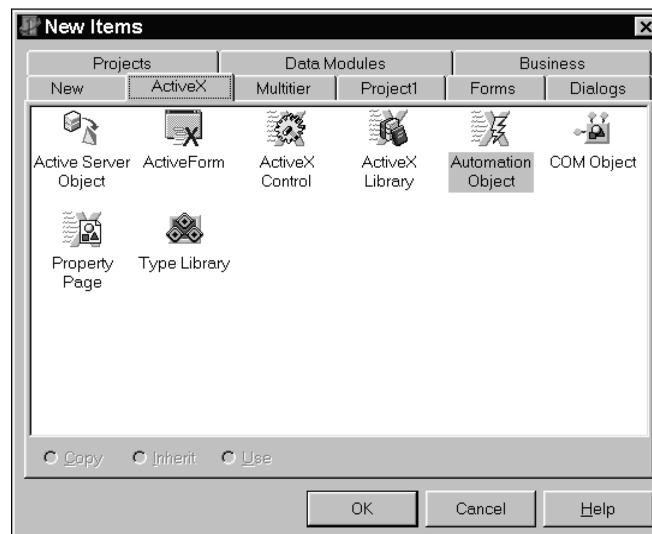


Рис. 23.3. Добавление нового объекта автоматизации

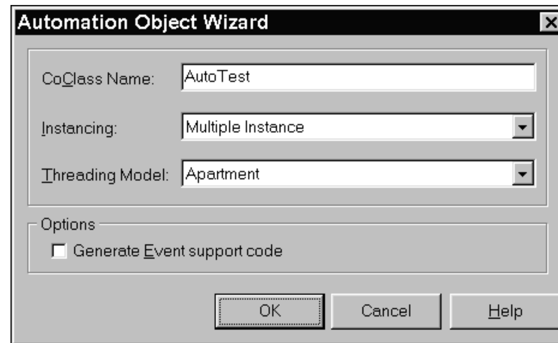


Рис. 23.4. Диалоговое окно мастера Automation Object Wizard

В поле Class Name диалогового окна Automation Object Wizard необходимо ввести имя COM-класса для данного объекта автоматизации. Мастер автоматически добавит префикс `T` к имени класса, если создается класс Object Pascal для объекта автоматизации, и префикс `I`, если создается основной интерфейс для объекта автоматизации. В раскрывающемся списке Instancing содержится три значения, которые представленны в табл. 23.1.

Таблица 23.1. Допустимые варианты использования объекта автоматизации

Значение	Описание
Internal	Этот OLE-объект может использоваться только внутри приложения и его не нужно регистрировать в системном реестре. Внешние процессы не смогут иметь доступ к внутреннему экземпляру сервера автоматизации
Single Instance	Каждый экземпляр сервера может экспортировать только один экземпляр OLE-объекта. Если приложение-контроллер запрашивает другой экземпляр OLE-объекта, Windows запускает новый экземпляр приложения-сервера
Multiple Instance	Каждый экземпляр сервера может создавать и экспортировать множество экземпляров OLE-объекта. Внутренние серверы всегда “многоэкземплярны”

После установки всех параметров в диалоговом окне Automation Object Wizard Delphi создаст новую библиотеку типов для проекта (если она еще не создана) и добавит в нее описание интерфейса и COM-класса. Мастер также сгенерирует новый модуль в проекте, который содержит реализацию интерфейса автоматизации, добавленного к библиотеке типа. На рис. 23.5 показано окно редактора библиотеки типов сразу же после закрытия диалогового окна мастера, а в листинге 23.2 приведен код модуля реализации объекта автоматизации.

Листинг 23.2. Модуль реализации объекта автоматизации

```
unit TestImpl;

interface

uses
```

```
  ComObj, ActiveX, Srv_TLB;

type
  TAutoTest = class(TAutoObject, IAutoTest)
  protected
    { Защищенные объявления }
  end;

implementation

uses ComServ;

initialization
  TAutoObjectFactory.Create(ComServer, TAutoTest, Class_AutoTest,
    ciMultiInstance, tmApartment);
end.
```

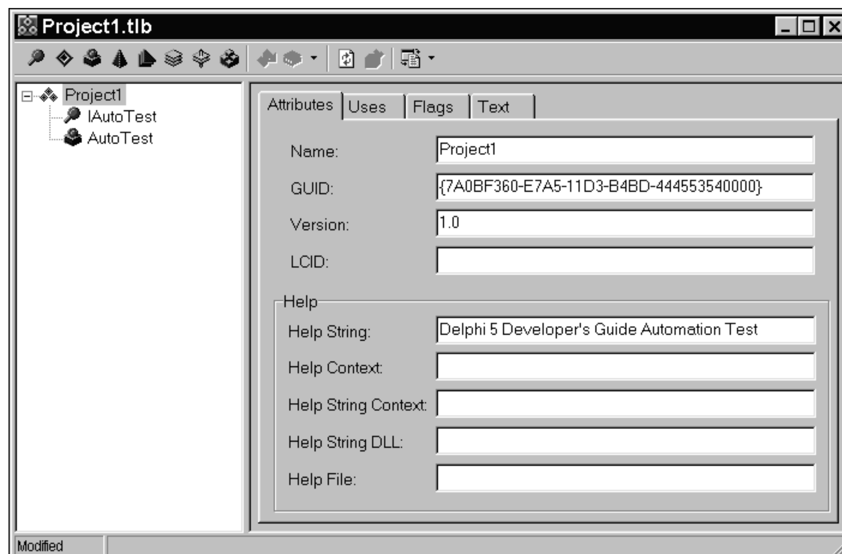


Рис. 23.5. Новый проект автоматизации в редакторе библиотеки типов

Объект автоматизации `TAutoTest` является классом, производным от класса `TAutoObject`. Класс `TAutoObject` — это базовый класс для всех серверов автоматизации. При добавлении методов к интерфейсу с использованием редактора библиотеки типов в том же модуле будут сгенерированы и каркасы новых методов, которые образуют фундамент объекта автоматизации.

И снова напомним, что не нужно путать класс `TAutoObject` Delphi 2 (модуль `OleAuto`) с классом `TAutoObject` Delphi 5 (модуль `ComObj`) — они не совместимы. Введенный в Delphi 2 спецификатор видимости `automated` устарел и в Delphi 5 практически не используется.

После добавления к проекту объекта автоматизации необходимо добавить к исходному интерфейсу одно или несколько свойств и методов, используя редактор библиотеки типов. В рассматриваемом проекте библиотека типов будет содержать свойства для чтения и установки формы, цвета и типа, а также текста в элементах управления редактированием. Для визуального представления стоит также добавить метод, который отображает текущее состояние этих свойств в диалоговом окне. На рис. 23.6 показана законченная библиотека типов проекта Srv. Обратите внимание на добавленное в библиотеку типов перечисление (его члены отображены на левой панели), предназначенное для поддержки свойства ShapeType.

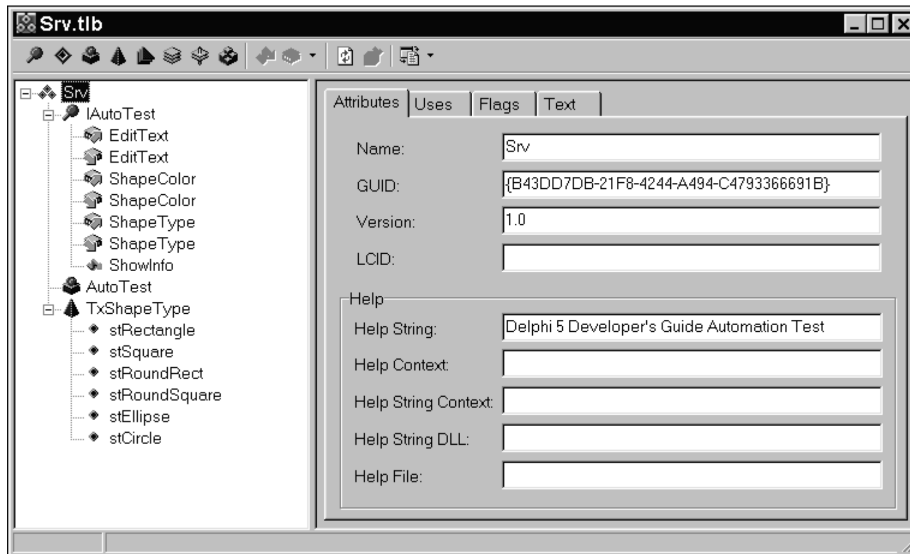


Рис. 23.6. Законченная библиотека типов

На заметку

Запомните, что при добавлении в библиотеке типов свойств и методов к объектам автоматизации используемые для этих свойств и методов параметры или возвращаемые значения должны быть совместимого с автоматизацией типа. Существуют следующие типы, совместимые с автоматизацией: Byte, SmallInt, Integer, Single, Double, Currency, TDateTime, WideString, WordBool, PSafeArray, TDecimal, OleVariant, IUnknown и IDispatch.

Завершая работу с библиотекой типов, не забудьте в каждом методе заменить текстом реализации те “заглушки”, которые были созданы редактором библиотеки типов. Текст модуля с реализацией всех методов приведен в листинге 23.3.

Листинг 23.3. Законченный модуль реализации

```
unit TestImpl;

interface

uses
  ComObj, ActiveX, Srv_TLB;
```

```
type
  TAutoTest = class(TAutoObject, IAutoTest)
  protected
    function Get_EditText: WideString; safecall;
    function Get_ShapeColor: OLE_COLOR; safecall;
    procedure Set_EditText(const Value: WideString); safecall;
    procedure Set_ShapeColor(Value: OLE_COLOR); safecall;
    function Get_ShapeType: TxShapeType; safecall;
    procedure Set_ShapeType(Value: TxShapeType); safecall;
    procedure ShowInfo; safecall;
  end;

implementation

uses ComServ, SrvMain, TypInfo, ExtCtrls, Dialogs, SysUtils, Graphics;

function TAutoTest.Get_EditText: WideString;
begin
  Result := FrmAutoTest.Edit.Text;
end;

function TAutoTest.Get_ShapeColor: OLE_COLOR;
begin
  Result := ColorToRGB(FrmAutoTest.Shape.Brush.Color);
end;

procedure TAutoTest.Set_EditText(const Value: WideString);
begin
  FrmAutoTest.Edit.Text := Value;
end;

procedure TAutoTest.Set_ShapeColor(Value: OLE_COLOR);
begin
  FrmAutoTest.Shape.Brush.Color := Value;
end;

function TAutoTest.Get_ShapeType: TxShapeType;
begin
  Result := TxShapeType(FrmAutoTest.Shape.Shape);
end;

procedure TAutoTest.Set_ShapeType(Value: TxShapeType);
begin
  FrmAutoTest.Shape.Shape := TShapeType(Value);
end;

procedure TAutoTest.ShowInfo;
const
  SInfoStr = 'The Shape's color is %s, and it's shape is %s.'#13#10 +
    'The Edit's text is "%s."';
```

```
begin
  with FrmAutoTest do
    ShowMessage(Format(SInfoStr, [ColorToString(Shape.Brush.Color),
      GetEnumName(TypeInfo(TShapeType), Ord(Shape.Shape)), Edit.Text]));
end;

initialization
  TAutoObjectFactory.Create(ComServer, TAutoTest, Class_AutoTest,
    ciMultiInstance, tmApartment);
end.
```

Раздел uses приведенного выше модуля включает имя модуля Srv_TLB. Содержимое модуля Srv_TLB представляет библиотеку типов данного проекта в виде трансляции Object Pascal и приведено в листинге 23.4.

Листинг 23.4. Модуль Srv_TLB — файл библиотеки типов сервера автоматизации

```
unit Srv_TLB;

// ************************************************************************ //
// ВНИМАНИЕ
// --------
// Типы, объявленные в этом файле, были сгенерированы из данных библиотеки
// типов. Если эта библиотека типа явно или косвенно (через другую библиотеку
// типов, ссылающуюся на эту библиотеку типа) повторно импортирована или
// с помощью команды 'Refresh' в окне Type Library Editor активизирована
// во время редактирования библиотеки типа, содержимое этого файла будет
// регенерировано, а все внесенные вручную изменения будут утеряны.
// ************************************************************************ //

// PASTLWTR : $Revision: 1.88  $
// Файл сгенерирован 10/28/99 1:55:17 PM из библиотеки типов, описанной ниже.
// ************************************************************************ //
// ПРИМЕЧАНИЕ:
// Элементы, защищенные директивой $IFDEF_LIVE_SERVER_AT_DESIGN_TIME,
// используются свойствами, возвращаемыми объектами, которые придется явно
// создать с помощью вызова функции до любого доступа посредством
// свойства. Эти элементы запрещены ради предотвращения
// случайного их использования из окна инспектора
// объектов. Вы можете разрешить их, определив константу
// LIVE_SERVER_AT_DESIGN_TIME или избирательно удалив их из
// блоков $IFDEF. Однако такие элементы должны по-прежнему создаваться
// программным путем с помощью метода соответствующего компонентного
// класса перед их использованием
// ************************************************************************ //
// Type Lib:   C:\work\d5dg\code\Ch23\Automate\Srv.tlb (1)
// IID\ LCID: {B43DD7DB-21F8-4244-A494-C4793366691B}\0
// Helpfile:
// DepndLst:
//   (1) v2.0 stdole, (C:\WINDOWS\SYSTEM\STDOLE2.TLB)
```

```
//   (2) v4.0 StdVCL, (C:\WINDOWS\SYSTEM\STDVCL40.DLL)
// ************************************************************************ //
{$TYPEDADDRESS OFF} // Модуль должен быть скомпилирован без указателей,
                    // проверяемых на тип

interface

uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;

// **********************************************************************//
// GUID объявлены в TypeLibrary. Используются следующие префиксы:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// **********************************************************************//
const
// Главная и вспомогательные версии библиотеки типов
  SrvMajorVersion = 1;
  SrvMinorVersion = 0;

  LIBID_Srv: TGUID = '{B43DD7DB-21F8-4244-A494-C4793366691B}';

  IID_IAutoTest: TGUID = '{C16B6A4C-842C-417F-8BF2-2F306F6C6B59}';
  CLASS_AutoTest: TGUID = '{64C576F0-C9A7-420A-9EAB-0BE98264BC9D}';
// **********************************************************************//
// Объявление перечислений, определенных в библиотеке типов
// **********************************************************************//
// Константы для перечисления TxShapeType
type
    TxShapeType = TOleEnum;
const
    stRectangle = $00000000;
    stSquare = $00000001;
    stRoundRect = $00000002;
    stRoundSquare = $00000003;
    stEllipse = $00000004;
    stCircle = $00000005;

type

// **********************************************************************//
// Предварительное объявление интерфейсов, определенных в библиотеке типов
// **********************************************************************//
    IAutoTest = interface;
    IAutoTestDisp = dispinterface;

// **********************************************************************//
// Объявление компонентных классов, определенных в библиотеке типа
// (ПРИМЕЧАНИЕ. Здесь описывается каждый компонентный класс со своим
```

```
// стандартным интерфейсом)
// *********************************************************************//
    AutoTest = IAutoTest;

// *********************************************************************//
// Interface: IAutoTest
// Flags:     (4432) Hidden Dual OleAutomation Dispatchable
// GUID:      {C16B6A4C-842C-417F-8BF2-2F306F6C6B59}
// *********************************************************************//
    IAutoTest = interface(IDispatch)
        ['{C16B6A4C-842C-417F-8BF2-2F306F6C6B59}']
        function Get_EditText: WideString; safecall;
        procedure Set_EditText(const Value: WideString); safecall;
        function Get_ShapeColor: OLE_COLOR; safecall;
        procedure Set_ShapeColor(ShapeColor: OLE_COLOR); safecall;
    function  Get_ShapeType: TxShapeType; safecall;
    procedure Set_ShapeType(Value: TxShapeType); safecall;
    procedure ShowInfo; safecall;
    property EditText: WideString read Get_EditText write Set_EditText;
    property ShapeColor: OLE_COLOR read Get_ShapeColor write Set_ShapeColor;
    property ShapeType: TxShapeType read Get_ShapeType write Set_ShapeType;
  end;
//*********************************************************************//
// DispIntf:  IAutoTestDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {C16B6A4C-842C-417F-8BF2-2F306F6C6B59}
// *********************************************************************//
    IAutoTestDisp = dispinterface
['{C16B6A4C-842C-417F-8BF2-2F306F6C6B59}']
    property EditText: WideString dispid 1;
    property ShapeColor: OLE_COLOR dispid 2;
    property ShapeType: TxShapeType dispid 3;
    procedure ShowInfo; dispid 4;
  end;
// *********************************************************************//
// Класс CoAutoTest предоставляет методы Create и CreateRemote
// для создания экземпляров стандартного интерфейса IAutoTest, открываемого
// компонентным классом AutoTest. Эти функции предназначены для использования
// клиентами, желающими автоматизировать CoClass-объекты, открытые
// сервером этой библиотеки типов
// *********************************************************************//
  CoAutoTest = class
    class function Create: IAutoTest;
    class function CreateRemote(const MachineName: string): IAutoTest;
  end;
// *********************************************************************//
// OLE Server Proxy class declaration
// Server Object    : TAutoTest
// Help String      : AutoTest Object
// Default Interface: IAutoTest
```

```
// Def. Intf. DISP? : No
// Event   Interface:
// TypeFlags       : (2) CanCreate
// *********************************************************************//
{$IFDEF LIVE_SERVER_AT_DESIGN_TIME}
  TAutoTestProperties= class;
{$ENDIF}
  TAutoTest = class(TOleServer)
  private
    FIntf:        IAutoTest;
{$IFDEF LIVE_SERVER_AT_DESIGN_TIME}
    FProps:       TAutoTestProperties;
    function      GetServerProperties: TAutoTestProperties;
{$ENDIF}
    function      GetDefaultInterface: IAutoTest;
  protected
    procedure InitServerData; override;
    function  Get_EditText: WideString;
    procedure Set_EditText(const Value: WideString);
    function  Get_ShapeColor: OLE_COLOR;
    procedure Set_ShapeColor(Value: OLE_COLOR);
    function  Get_ShapeType: TxShapeType;
    procedure Set_ShapeType(Value: TxShapeType);
  public
    constructor Create(AOwner: TComponent); override;
    destructor  Destroy; override;
    procedure Connect; override;
    procedure ConnectTo(svrIntf: IAutoTest);
    procedure Disconnect; override;
    procedure ShowInfo;
    property  DefaultInterface: IAutoTest read GetDefaultInterface;
    property EditText: WideString read Get_EditText write Set_EditText;
    property ShapeColor: OLE_COLOR read Get_ShapeColor write
      Set_ShapeColor;
    property ShapeType: TxShapeType read Get_ShapeType write
      Set_ShapeType;
  published
{$IFDEF LIVE_SERVER_AT_DESIGN_TIME}
    property Server: TAutoTestProperties read GetServerProperties;
{$ENDIF}
  end;

{$IFDEF LIVE_SERVER_AT_DESIGN_TIME}
// *********************************************************************//
// OLE Server Properties Proxy Class
// Server Object    : TAutoTest
// (Этот объект используется инспектором свойств IDE для разрешения
// редактирования свойств данного сервера)
// *********************************************************************//
 TAutoTestProperties = class(TPersistent)
```

```
  private
    FServer:    TAutoTest;
    function    GetDefaultInterface: IAutoTest;
    constructor Create(AServer: TAutoTest);
  protected
    function  Get_EditText: WideString;
    procedure Set_EditText(const Value: WideString);
    function  Get_ShapeColor: OLE_COLOR;
    procedure Set_ShapeColor(Value: OLE_COLOR);
    function  Get_ShapeType: TxShapeType;
    procedure Set_ShapeType(Value: TxShapeType);
  public
    property DefaultInterface: IAutoTest read GetDefaultInterface;
  published
    property EditText: WideString read Get_EditText write Set_EditText;
    property ShapeColor: OLE_COLOR read Get_ShapeColor write
      Set_ShapeColor;
    property ShapeType: TxShapeType read Get_ShapeType write
      Set_ShapeType;
  end;
{$ENDIF}

procedure Register;

implementation

uses ComObj;

class function CoAutoTest.Create: IAutoTest;
begin
  Result := CreateComObject(CLASS_AutoTest) as IAutoTest;
end;

class function CoAutoTest.CreateRemote(const MachineName: string):
  IAutoTest;
begin
Result := CreateRemoteComObject(MachineName, CLASS_AutoTest) as IAutoTest;
end;

procedure TAutoTest.InitServerData;
const
  CServerData: TServerData = (
    ClassID:   '{64C576F0-C9A7-420A-9EAB-0BE98264BC9D}';
    IntfIID:   '{C16B6A4C-842C-417F-8BF2-2F306F6C6B59}';
    EventIID:  '';
    LicenseKey: nil;
    Version: 500);
begin
  ServerData := @CServerData;
end;
```

```
procedure TAutoTest.Connect;
var
  punk: IUnknown;
begin
  if FIntf = nil then
  begin
    punk := GetServer;
    Fintf:= punk as IAutoTest;
  end;
end;

procedure TAutoTest.ConnectTo(svrIntf: IAutoTest);
begin
  Disconnect;
  FIntf := svrIntf;
end;

procedure TAutoTest.DisConnect;
begin
if Fintf <> nil then
  begin
    FIntf := nil;
  end;
end;

function TAutoTest.GetDefaultInterface: IAutoTest;
const
  ErrStr = 'DefaultInterface is NULL. Component is not connected to ' +
    'Server. You must call ''Connect'' or ''ConnectTo'' before this ' +
    'operation';
begin
  if FIntf = nil then
    Connect;
  Assert(FIntf <> nil, ErrStr);
  Result := FIntf;
end;

constructor TAutoTest.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
{$IFDEF LIVE_SERVER_AT_DESIGN_TIME}
  FProps := TAutoTestProperties.Create(Self);
{$ENDIF}
end;

destructor TAutoTest.Destroy;
begin
{$IFDEF LIVE_SERVER_AT_DESIGN_TIME}
  FProps.Free;
```

```
{$ENDIF}
  inherited Destroy;
end;

{$IFDEF LIVE_SERVER_AT_DESIGN_TIME}
function TAutoTest.GetServerProperties: TAutoTestProperties;
begin
  Result := FProps;
end;
{$ENDIF}

function  TAutoTest.Get_EditText: WideString;
begin
  Result := DefaultInterface.Get_EditText;
end;

procedure TAutoTest.Set_EditText(const Value: WideString);
begin
  DefaultInterface.Set_EditText(Value);
end;

function  TAutoTest.Get_ShapeColor: OLE_COLOR;
begin
  Result := DefaultInterface.Get_ShapeColor;
end;

procedure TAutoTest.Set_ShapeColor(Value: OLE_COLOR);
begin
  DefaultInterface.Set_ShapeColor(Value);
end;

function  TAutoTest.Get_ShapeType: TxShapeType;
begin
  Result := DefaultInterface.Get_ShapeType;
end;

procedure TAutoTest.Set_ShapeType(Value: TxShapeType);
begin
  DefaultInterface.Set_ShapeType(Value);
end;

procedure TAutoTest.ShowInfo;
begin
  DefaultInterface.ShowInfo;
end;

{$IFDEF LIVE_SERVER_AT_DESIGN_TIME}
constructor TAutoTestProperties.Create(AServer: TAutoTest);
begin
  inherited Create;
```

```
  FServer := AServer;
end;

function TAutoTestProperties.GetDefaultInterface: IAutoTest;
begin
  Result := FServer.DefaultInterface;
end;

function  TAutoTestProperties.Get_EditText: WideString;
begin
  Result := DefaultInterface.Get_EditText;
end;

procedure TAutoTestProperties.Set_EditText(const Value: WideString);
begin
  DefaultInterface.Set_EditText(Value);
end;

function  TAutoTestProperties.Get_ShapeColor: OLE_COLOR;
begin
  Result := DefaultInterface.Get_ShapeColor;
end;

procedure TAutoTestProperties.Set_ShapeColor(Value: OLE_COLOR);
begin
  DefaultInterface.Set_ShapeColor(Value);
end;

function  TAutoTestProperties.Get_ShapeType: TxShapeType;
begin
  Result := DefaultInterface.Get_ShapeType;
end;

procedure TAutoTestProperties.Set_ShapeType(Value: TxShapeType);
begin
  DefaultInterface.Set_ShapeType(Value);
end;

{$ENDIF}

procedure Register;
begin
  RegisterComponents('Servers',[TAutoTest]);
end;

end.
```

Рассмотрим подробнее этот модуль. Вначале объявляется версия библиотеки типов, а затем — уникальный идентификатор GUID для этой библиотеки типов — LIBID_Srv. Это значение идентификатора GUID будет использовано при регистрации библиотеки типов в сис-

темном реестре. Затем приводятся значения для перечисления TxShapeType. Самое интересное в этом перечислении то, что его значения объявляются как константы, а не с использованием перечислимого типа Object Pascal. Эти перечисления подобны перечислениям в C/C++, которые, в отличие от перечислений в Object Pascal, не обязаны начинаться с нулевого значения и представлять ряд последовательных значений.

Затем в модуле Srv_TLB объявляется интерфейс IAutoTest. В этом объявлении присутствуют свойства и методы, созданные в редакторе библиотеки типов. Кроме того, нетрудно заметить, что методы Get_*XXX* и Set_*XXX* генерируются как методы чтения и записи для каждого свойства.

Соглашение о вызовах safecall

Это — соглашение о вызовах по умолчанию для методов, введенных в редакторе библиотеки типа (см. приведенное выше объявление интерфейса IAutoTest). Однако ключевое слово safecall несет в себе несколько большую нагрузку, чем просто соглашение по вызову. Поясним, что имеется в виду. Во-первых, метод должен вызываться с использованием соглашения по вызову safecall. Во-вторых, метод будет инкапсулирован так, чтобы возвращать значение типа HResult. Например, предположим, что существует метод, который в Object Pascal выглядит следующим образом:

```
function Foo(W: WideString): Integer; safecall;
```

На самом деле этот метод компилируется в следующую программную строку:

```
function Foo(W: WideString; outRetVal: Integer): HResult; stdcall;
```

Преимущество соглашения safecall состоит в том, что все исключительные ситуации перехватываются до того, как они будут возвращены в вызывающий метод. При возникновении в safecall-методе необработанного исключения, оно обрабатывается неявным инкапсулятором-оболочкой с преобразованием в значение типа HResult, который и возвращается вызывающему методу.

В модуле Srv_TLB есть объявление dispinterface объекта автоматизации IAutoTestDisp. Ключевое слово dispinterface сигнализирует вызывающему методу, что методы автоматизации могут быть выполнены с помощью метода Invoke(), но при этом не имеется в виду пользовательский интерфейс, посредством которого могут выполняться методы. Несмотря на то что интерфейс IAutoTest может быть использован как инструмент разработки, который поддерживает автоматизацию с применением раннего связывания, благодаря объявлению dispinterface объекта IAutoTestDisp можно использовать средства, поддерживающие позднее связывание.

Затем в модуле Srv_TLB объявляется класс CoAutoTest, который весьма облегчает процесс создания объекта автоматизации, в результате чего для создания экземпляра этого объекта достаточно вызвать метод CoAutoTest.Create().

Наконец, в модуле Srv_TLB создается класс TAutoTest, превращающий сервер в компонент, который можно поместить в палитру компонентов. Эта возможность, появившаяся лишь в Delphi 5, относится скорее к импортируемым, чем к вновь создаваемым серверам автоматизации.

Как уже упоминалось выше в этой главе, созданное приложение первый раз запускается для выполнения регистрации в системном реестре. Далее в этой главе речь пойдет о том, как использовать приложение-контроллер для управления сервером.

Создание внутреннего сервера автоматизации

Подобно тому как создание внешних серверов начинается с построениям отдельных приложений, создание внутренних серверов начинается с построения библиотек DLL. Можно воспользоваться уже существующей библиотекой DLL или создать совершенно новую, дважды щелкнув на пиктограмме DLL в диалоговом окне New Items (команда File⇨New).

На заметку

Если вы незнакомы с методами разработки библиотек DLL, обратитесь к главе 9, “Динамически компонуемые библиотеки” (том I). В данной главе предполагается, что читатель уже имеет представление о создании подобных приложений.

Как уже упоминалось выше, чтобы успешно работать в качестве внутреннего сервера автоматизации, библиотека DLL должна экспортировать четыре функции, определения которых присутствуют в модуле `ComServ`: `DllGetClassObject()`, `DllCanUnloadNow()`, `DllRegisterServer()` и `DllUnregisterServer()`. Добавьте эти четыре функции в раздел `exports` вашего проекта так, как показано в листинге 23.5.

Листинг 23.5. Проект `IPS.dpr` — файл проекта для внутреннего сервера

```
library IPS;

uses
  ComServ;

exports
  DllRegisterServer,
  DllUnregisterServer,
  DllGetClassObject,
  DllCanUnloadNow;

begin
end.
```

Объект автоматизации добавляется к проекту DLL таким же способом, как и к выполняемому проекту, — с помощью мастера объектов автоматизации Automation Object Wizard. Для данного проекта мы добавим лишь одно свойство и один метод, как показано на рис. 23.7. Текст библиотеки типов в версии Object Pascal приведен в листинге 23.6.

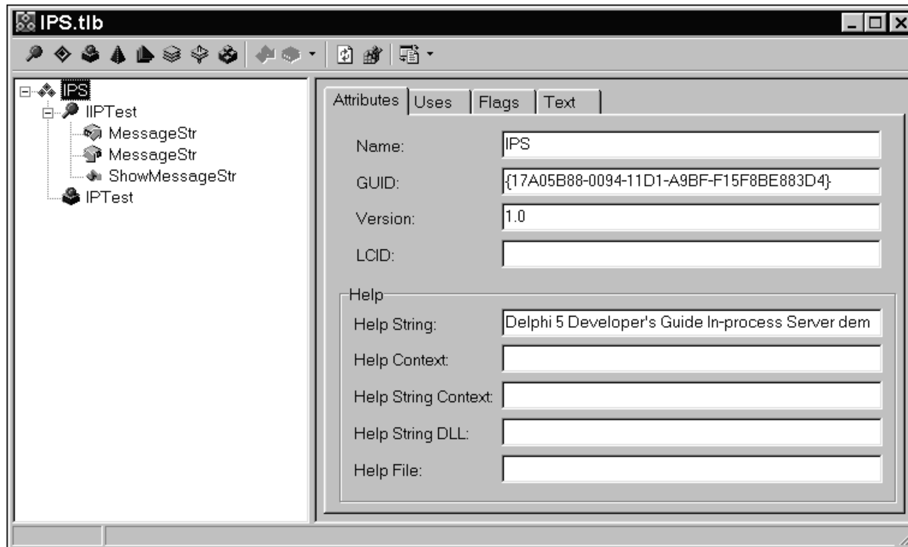


Рис. 23.7. Проект `IPS` в окне редактора библиотеки типов

Листинг 23.6. Модуль `IPS_TLB.pas` — файл импортирования библиотеки типов для проекта внутреннего сервера

```
unit IPS_TLB;

// ************************************************************************//
// ВНИМАНИЕ
// --------
// Типы, объявленные в этом файле, были сгенерированы из данных библиотеки
// типа. Если эта библиотека типов явно или косвенно (через другую библиотеку
// типов, ссылающуюся на данную) будет реимпортирована или обновлена
// с помощью команды 'Refresh' в окне Type Library Editor во время
// редактирования библиотеки типов, содержимое этого файла будет
// регенерировано, а все внесенные вручную изменения быдут утеряны
// ************************************************************************ //

// PASTLWTR : $Revision: 1.79   $
// Файл сгенерирован 8/14/99 11:37:16 PM из библиотеки типов, описанной ниже

// ************************************************************************ //
// Type Lib: C:\work\d5dg\code\Ch23\Automate\IPS.tlb
// IID\ LCID: {17A05B88-0094-11D1-A9BF-F15F8BE883D4}\0
// Helpfile:
// DepndLst:
//   (1) v1.0 stdole, (C:\WINDOWS\SYSTEM\stdole32.tlb)
//   (2) v2.0 StdType, (c:\WINDOWS\SYSTEM\OLEPRO32.DLL)
//   (3) v1.0 StdVCL, (C:\WINDOWS\SYSTEM\STDVCL32.DLL)
// ************************************************************************ //
interface

uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;

// ************************************************************************//
// Идентификаторы GUID объявлены в библиотеке типов. Используются префиксы:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// ************************************************************************//
const
// Главная и вспомогательные версии библиотеки типов
  IPSMajorVersion = 1;
  IPSMinorVersion = 0;

  LIBID_IPS: TGUID = '{17A05B88-0094-11D1-A9BF-F15F8BE883D4}';

  IID_IIPTest: TGUID = '{17A05B89-0094-11D1-A9BF-F15F8BE883D4}';
  CLASS_IPTest: TGUID = '{17A05B8A-0094-11D1-A9BF-F15F8BE883D4}';
type
```

```
// *********************************************************************//
// Предварительное объявление интерфейсов, определенных в библиотеке типов.
// *********************************************************************//
  IIPTest = interface;
  IIPTestDisp = dispinterface;

// *********************************************************************//
// Объявление компонентных классов, определенных в библиотеке типов.
// (ПРИМЕЧАНИЕ. Здесь мы отображаем каждый компонентный класс на
// его стандартный интерфейс.)
// *********************************************************************//
  IPTest = IIPTest;

// *********************************************************************//
// Interface: IIPTest
// Flags:     (4432) Hidden Dual OleAutomation Dispatchable
// GUID:      {17A05B89-0094-11D1-A9BF-F15F8BE883D4}
// *********************************************************************//
  IIPTest = interface(IDispatch)
    ['{17A05B89-0094-11D1-A9BF-F15F8BE883D4}']
    function Get_MessageStr: WideString; safecall;
    procedure Set_MessageStr(const Value: WideString); safecall;
    function ShowMessageStr: Integer; safecall;
    property MessageStr: WideString read Get_MessageStr write Set_MessageStr;
  end;

// *********************************************************************//
// DispIntf:  IIPTestDisp
// Flags:     (4432) Hidden Dual OleAutomation Dispatchable
// GUID:      {17A05B89-0094-11D1-A9BF-F15F8BE883D4}
// *********************************************************************//
  IIPTestDisp = dispinterface
    ['{17A05B89-0094-11D1-A9BF-F15F8BE883D4}']
    property MessageStr: WideString dispid 1;
    function ShowMessageStr: Integer; dispid 2;
  end;
// *********************************************************************//
// Класс CoIPTest предоставляет методы Create и CreateRemote
// для создания экземпляров стандартного интерфейса IIPTest, предоставляемого
// компонентным классом IPTest. Эти функции предназначены для использования
// клиентами, желающими средствами автоматизации использовать объекты
// компонентного класса, предоставляемые сервером этой библиотеки типов.
// *********************************************************************//
  CoIPTest = class
    class function Create: IIPTest;
    class function CreateRemote(const MachineName: string): IIPTest;
  end;


implementation
```

```
uses ComObj;

class function CoIPTest.Create: IIPTest;
begin
  Result := CreateComObject(CLASS_IPTest) as IIPTest;
end;

class function CoIPTest.CreateRemote(const MachineName: string): IIPTest;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_IPTest) as IIPTest;
end;

end.
```

Очевидно, что выше приведен простейший пример сервера автоматизации, но он предназначен всего лишь для целей иллюстрации. Свойству `MessageStr` может быть присвоено значение, которое затем можно отобразить с помощью функции `ShowMessageStr()`. Реализация интерфейса `IIPTest` содержится в модуле `IPSMain.pas`, код которого приведен в листинге 23.7.

Листинг 23.7. Модуль `IPSMain.pas` — главный модуль проекта создания внутреннего сервера автоматизации

```
unit IPSMain;

interface

uses
  ComObj, IPS_TLB;

type
  TIPTest = class(TAutoObject, IIPTest)
  private
    MessageStr: string;
  protected
    function Get_MessageStr: WideString; safecall;
    procedure Set_MessageStr(const Value: WideString); safecall;
    function ShowMessageStr: Integer; safecall;
  end;

implementation

uses Windows, ComServ;

function TIPTest.Get_MessageStr: WideString;
begin
  Result := MessageStr;
end;

function TIPTest.ShowMessageStr: Integer;
```

```
begin
  MessageBox(0, PChar(MessageStr), 'Ваша строка...', MB_OK);
  Result := Length(MessageStr);
end;

procedure TIPTest.Set_MessageStr(const Value: WideString);
begin
  MessageStr := Value;
end;

initialization
  TAutoObjectFactory.Create(ComServer, TIPTest, Class_IPTest, ciMultiInstance);
end.
```

Как уже упоминалось выше в этой главе, внутренние серверы регистрируются не так, как внешние. Для регистрации в системном реестре внутреннего сервера вызывается функция DllRegisterServer(). В интегрированной среде разработки Delphi этот процесс выполняется очень просто — с помощью команды Run⇨Register ActiveX Server.

Создание контроллеров автоматизации

В Delphi процессы управления серверами автоматизации из приложений максимально упрощены. Но, тем не менее, при этом обеспечивается высокая степень гибкости управления — благодаря возможности раннего (с помощью интерфейсов) или позднего (с использованием Delphi-типов dispinterface или OleVariant) связывания.

Управление внешним сервером

Проект Control представляет собой контроллер автоматизации, с помощью которого демонстрируются все три типа поддержки функций автоматизации (интерфейсы, диспинтерфейсы и варианты). Проект Control — это контроллер, предназначенный для работы с приложением сервера автоматизации Srv, создание которого описано ранее в этой главе. Главная форма этого проекта показана на рис. 23.8.

По щелчку на кнопке Connect приложение Control подключается к серверу, как показано ниже.

```
FIntf := CoAutoTest.Create;
FDispintf := CreateComObject(Class_AutoTest)
 as IAutoTestDisp;
FVar := CreateOleObject('Srv.AutoTest');
```

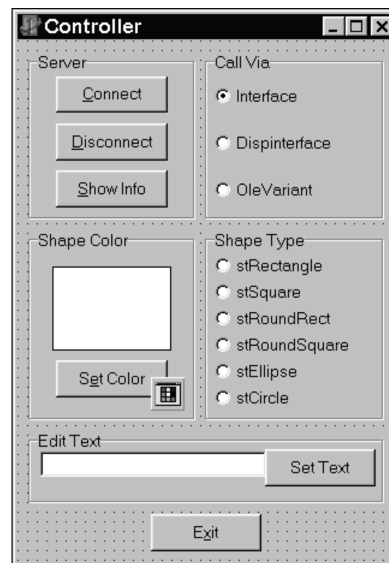


Рис. 23.8. Главная форма проекта Control

В этом фрагменте кода вы видите переменные типа `interface`, `dispinterface` и `OleVariant`, с помощью которых экземпляры серверов автоматизации создаются тремя различными способами. Интересно то, что эти способы практически абсолютно взаимозаменяемы. Например, следующий код также верен:

```
FIntf := CreateComObject(Class_AutoTest) as IAutoTest;
FDispintf := CreateOleObject('Srv.AutoTest') as IAutoTestDisp;
FVar := CoAutoTest.Create;
```

В листинге 23.8 показан текст модуля `Ctrl`, который содержит остальную часть кода контроллера автоматизации. Заметьте, что это приложение позволяет управлять сервером, используя любой тип переменной Delphi: `interface`, `dispinterface` или `OleVariant`.

Листинг 23.8. Модуль `Ctrl.pas` — главный модуль проекта контроллера, предназначенного для работы с внешним сервером

```
unit Ctrl;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ColorGrd, ExtCtrls, Srv_TLB, Buttons;

type
  TControlForm = class(TForm)
    CallViaRG: TRadioGroup;
    ShapeTypeRG: TRadioGroup;
    GroupBox1: TGroupBox;
    GroupBox2: TGroupBox;
    Edit: TEdit;
    GroupBox3: TGroupBox;
    ConBtn: TButton;
    DisBtn: TButton;
    InfoBtn: TButton;
    ColorBtn: TButton;
    ColorDialog: TColorDialog;
    ColorShape: TShape;
    ExitBtn: TButton;
    TextBtn: TButton;
    procedure ConBtnClick(Sender: TObject);
    procedure DisBtnClick(Sender: TObject);
    procedure ColorBtnClick(Sender: TObject);
    procedure ExitBtnClick(Sender: TObject);
    procedure TextBtnClick(Sender: TObject);
    procedure InfoBtnClick(Sender: TObject);
    procedure ShapeTypeRGClick(Sender: TObject);
  private
    { Закрытые объявления }
```

```
    FIntf: IAutoTest;
    FDispintf: IAutoTestDisp;
    FVar: OleVariant;
    procedure SetControls;
    procedure EnableControls(DoEnable: Boolean);
  public
    { Открытые объявления }
  end;

var
  ControlForm: TControlForm;

implementation

{$R *.DFM}

uses ComObj;

procedure TControlForm.SetControls;
// Инициализация управляющих элементов текущими значениями сервера
begin
  case CallViaRG.ItemIndex of
    0:
      begin
        ColorShape.Brush.Color := FIntf.ShapeColor;
        ShapeTypeRG.ItemIndex := FIntf.ShapeType;
        Edit.Text := FIntf.EditText;
      end;
    1:
      begin
        ColorShape.Brush.Color := FDispintf.ShapeColor;
        ShapeTypeRG.ItemIndex := FDispintf.ShapeType;
        Edit.Text := FDispintf.EditText;
      end;
    2:
      begin
        ColorShape.Brush.Color := FVar.ShapeColor;
        ShapeTypeRG.ItemIndex := FVar.ShapeType;
        Edit.Text := FVar.EditText;
      end;
  end;
end;

procedure TControlForm.EnableControls(DoEnable: Boolean);
begin
  DisBtn.Enabled := DoEnable;
  InfoBtn.Enabled := DoEnable;
  ColorBtn.Enabled := DoEnable;
```

```
  ShapeTypeRG.Enabled := DoEnable;
  Edit.Enabled := DoEnable;
  TextBtn.Enabled := DoEnable;
end;

procedure TControlForm.ConBtnClick(Sender: TObject);
begin
  FIntf := CoAutoTest.Create;
  FDispintf := CreateComObject(Class_AutoTest) as IAutoTestDisp;
  FVar := CreateOleObject('Srv.AutoTest');
  EnableControls(True);
  SetControls;
end;

procedure TControlForm.DisBtnClick(Sender: TObject);
begin
  FIntf := nil;
  FDispintf := nil;
  FVar := Unassigned;
  EnableControls(False);
end;

procedure TControlForm.ColorBtnClick(Sender: TObject);
var
  NewColor: TColor;
begin
  if ColorDialog.Execute then
  begin
    NewColor := ColorDialog.Color;
    case CallViaRG.ItemIndex of
      0: FIntf.ShapeColor := NewColor;
      1: FDispintf.ShapeColor := NewColor;
      2: FVar.ShapeColor := NewColor;
    end;
    ColorShape.Brush.Color := NewColor;
  end;
end;

procedure TControlForm.ExitBtnClick(Sender: TObject);
begin
  Close;
end;

procedure TControlForm.TextBtnClick(Sender: TObject);
begin
  case CallViaRG.ItemIndex of
    0: FIntf.EditText := Edit.Text;
    1: FDispintf.EditText := Edit.Text;
```

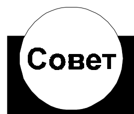
```
    2: FVar.EditText := Edit.Text;
  end;
end;

procedure TControlForm.InfoBtnClick(Sender: TObject);
begin
  case CallViaRG.ItemIndex of
    0: FIntf.ShowInfo;
    1: FDispintf.ShowInfo;
    2: FVar.ShowInfo;
  end;
end;

procedure TControlForm.ShapeTypeRGClick(Sender: TObject);
begin
  case CallViaRG.ItemIndex of
    0: FIntf.ShapeType := ShapeTypeRG.ItemIndex;
    1: FDispintf.ShapeType := ShapeTypeRG.ItemIndex;
    2: FVar.ShapeType := ShapeTypeRG.ItemIndex;
  end;
end;

end.
```

Еще одна интересная особенность, иллюстрируемая в этом листинге, — простота отключения от сервера автоматизации: переменные типа interface и dispinterface достаточно установить равными значению nil, а переменную типа Variant — равной значению Unassigned. Естественно, сервер автоматизации также освобождается, если приложение Control закрывается обычным способом.



Использование интерфейсов практически всегда более производительно, чем использование типов dispinterface или Variant, поэтому по возможности всегда для управления серверами автоматизации используйте интерфейсы.

Тип Variant по части производительности хуже прочих методов, поскольку в этом случае во время выполнения приложения перед вызовом нужного метода посредством обращения к методу Invoke() необходимо вызывать функцию GetIDsOfNames() для преобразования имени метода в его диспетчерский идентификатор (dispatch ID).

Производительность при использовании типа dispinterface можно оценить как промежуточную между использованием интерфейса и типа Variant. Но почему же существует различие в производительности при использовании типов Variant и dispinterface, если в обоих случаях применяется позднее связывание? Причина в следующем: при использовании типа dispinterface возможна оптимизация, называемая *связыванием идентификаторов* (ID binding). Имеется в виду следующее: диспетчерские идентификаторы (dispatch ID) используемых методов известны еще на стадии компиляции, поэтому компилятору не нужно генерировать динамический вызов метода GetIDsOfNames() перед вызовом метода Invoke(). Однако наиболее заметное преимущество диспинтерфейсов перед типом Variant состоит в том, что тип dispinterface поддерживает использование утилиты CodeInsight с целью упрощения кодирования программ, что невозможно при работе с типом Variant.

На рис. 23.9 показана форма приложения Control, управляющего сервером Srv.

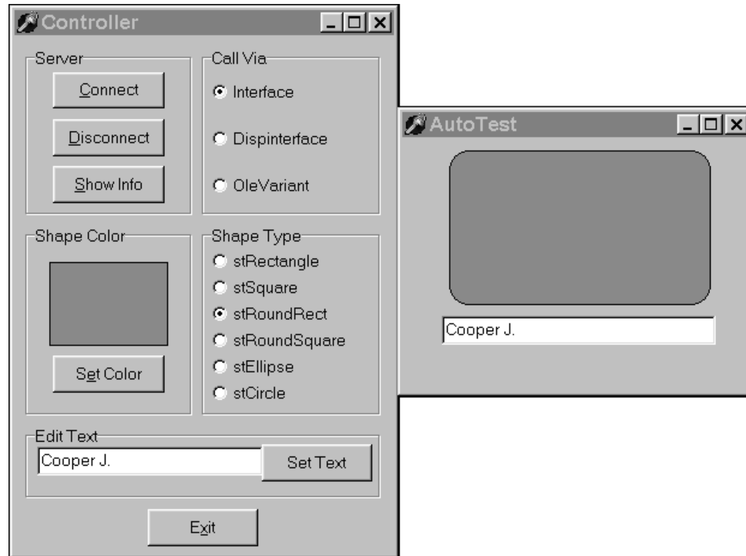


Рис. 23.9. Контроллер и сервер автоматизации

Управление внутренним сервером

Механизм управления внутренним сервером не отличается от механизма управления внешним сервером. Необходимо лишь помнить, что контроллер автоматизации в этом случае работает в пространстве процесса управляющего приложения. Это значит, что производительность будет несколько выше, чем при управлении внешним сервером, но это также означает и то, что крах сервера автоматизации может привести к полному разрушению приложения контроллера.

Рассмотрим приложение контроллера, предназначенного для управления внутренним сервером автоматизации, создание которого описано выше в этой главе. В данном случае для управления сервером используем только интерфейс. На рис. 23.10 показана главная форма проекта IPCtrl. Код главного модуля проекта IPCtrl (файл IPCMain.pas) приведен в листинге 23.9.

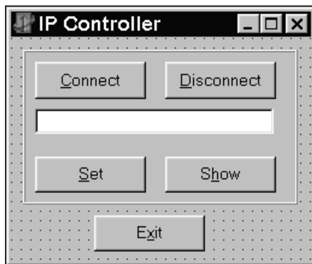


Рис. 23.10. Главная форма проекта IPCtrl

Листинг 23.9. Модуль IPCMain.pas — главный модуль проекта контроллера для управления внутренним сервером

```
unit IPCMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
```

```
  StdCtrls, ExtCtrls, IPS_TLB;

type
  TIPCForm = class(TForm)
    ExitBtn: TButton;
    Panel1: TPanel;
    ConBtn: TButton;
    DisBtn: TButton;
    Edit: TEdit;
    SetBtn: TButton;
    ShowBtn: TButton;
    procedure ConBtnClick(Sender: TObject);
    procedure DisBtnClick(Sender: TObject);
    procedure SetBtnClick(Sender: TObject);
    procedure ShowBtnClick(Sender: TObject);
    procedure ExitBtnClick(Sender: TObject);
  private
    { Закрытые объявления }
    IPTest: IIPTest;
    procedure EnableControls(DoEnable: Boolean);
  public
    { Открытые объявления }
  end;

var
  IPCForm: TIPCForm;

implementation

uses ComObj;

{$R *.DFM}

procedure TIPCForm.EnableControls(DoEnable: Boolean);
begin
  DisBtn.Enabled := DoEnable;
  Edit.Enabled := DoEnable;
  SetBtn.Enabled := DoEnable;
  ShowBtn.Enabled := DoEnable;
end;

procedure TIPCForm.ConBtnClick(Sender: TObject);
begin
  IPTest := CreateComObject(CLASS_IPTest) as IIPTest;
  EnableControls(True);
end;

procedure TIPCForm.DisBtnClick(Sender: TObject);
begin
  IPTest := nil;
```

```
  EnableControls(False);
end;

procedure TIPCForm.SetBtnClick(Sender: TObject);
begin
  IPTest.MessageStr := Edit.Text;
end;

procedure TIPCForm.ShowBtnClick(Sender: TObject);
begin
  IPTest.ShowMessageStr;
end;

procedure TIPCForm.ExitBtnClick(Sender: TObject);
begin
  Close;
end;

end.
```

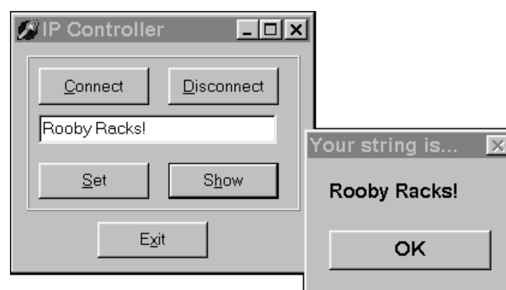


Рис. 23.11. Контроллер IPCtrl управляет сервером IPS

Не забудьте, что перед запуском проекта IPCtrl сервер должен быть зарегистрирован. Это можно сделать несколькими способами: воспользоваться командой Run⇨Register ActiveX Server, предварительно открыв проект IPS в среде разработки Delphi, прибегнуть к утилите Windows RegSrv32.exe или применить программу TRegSrv.exe, которая входит в поставку Delphi. На рис. 23.11 показан запущенный проект IPCtrl, управляющий сервером IPS.

Использование более сложных технологий автоматизации

В этом разделе мы рассмотрим более интересные возможности автоматизации, о которых часто не знают даже признанные мастера Delphi. Вы познакомитесь с событиями автоматизации, коллекциями, особенностями библиотеки типов и поддержкой языка низкого уровня для COM. Не будем тратить время на пространные рассуждения — лучше сразу приступим к делу!

События автоматизации

Программисты Delphi давно освоили работу с событиями, особенно в части их использования для создания обычных элементов управления. Берем, например, кнопку, помещаем ее в форму, дважды щелкаем на ее событии `OnClick` в окне инспектора объектов, пишем нужный программный код — и вся недолга. Даже при создании новых элементов управления с событиями особых проблем не возникает: создаем новый тип метода, добавляем в этот элемент управления поле и публикуемое свойство и остаемся вполне довольны собой. Однако у разработчиков COM в среде Delphi одно только упоминание о событиях может вызвать нервную дрожь. Многие Delphi-разработчики стараются избегать COM-событий лишь потому, что у них, видите ли, "нет времени разбираться во всех этих штучках". Если к последней группе относитесь и вы, читатель, то вам в очередной раз будет предоставлена возможность убедиться, что "не так страшен черт..." — благодаря прекрасной встроенной поддержке Delphi. И хотя все новые термины, связанные с событиями автоматизации, могут, казалось бы, еще более усложнить общую картину, в этом разделе, думаю, нам удастся добраться до самой сути этих событий, и вскоре вы не сможете сдержать недоумения: "И всего делов-то?!".

Что представляют собой события COM

Попросту говоря, события обеспечивают серверу средство организации обратного вызова в приложение-клиент для предоставления ему определенной информации. При использовании традиционной модели "клиент/сервер" клиент вызывает сервер для выполнения некоторого действия или получения некоторых данных, сервер выполняет нужное действие или получает данные, после чего управление возвращается клиенту. В большинстве случаев эта модель прекрасно работает, но она оказывается бессильной, когда событие, в котором заинтересован клиент, асинхронно по своей природе или зависит от элементов пользовательского интерфейса. Например, она не подходит в том случае, когда клиент посылает серверу требование загрузить файл, и при этом он (клиент) не хочет бесполезно тратить время на ожидание окончания загрузки, после чего он смог бы продолжить свою работу (особенно при использовании достаточно медленной модемной связи). Для клиента более предпочтительным было бы использовать такую модель, при которой сразу после отправки серверу некоторой инструкции можно было бы продолжать работу до тех пор, пока сервер не уведомит клиента о завершении загрузки файла. Аналогично, даже такой простейший способ работы с пользовательским интерфейсом, как щелчок на кнопке, служит хорошим примером ситуации, в которой серверу потребуется некоторый механизм уведомления клиента об имевшем место событий. Очевидно, что сам клиент не может вызвать в сервере метод, который будет ожидать щелчка на некоторой кнопке.

В общем случае сервер должен сам нести ответственность за перехват и генерацию событий, тогда как клиент будет отвечать за получение событий и подключение к серверу с целью их реализации. Конечно, такая вольная трактовка распределения ответственности может послужить прекрасной мишенью для придирок, тем не менее она работает. В соответствии с этим определением средствами Delphi и механизма автоматизации предлагаются два различных подхода к понятию событий. Углубленное рассмотрение деталей каждой из этих моделей поможет нам правильно охватить всю картину в перспективе.

События в Delphi

Delphi при подходе к событиям придерживается KISS-методологии (*k*eep *i*t *s*imple, *s*tupid! — чем проще, тем лучше). События реализуются как указатели на методы — эти указатели могут быть назначены для некоторого метода в приложении, — и они (события) выполняются посред-

ством вызова этого метода с помощью указателя на него. В качестве иллюстрации рассмотрим сценарий разработки приложения, в задачу которого входит обработка события в компоненте. Если на эту ситуацию посмотреть абстрактно, то "сервер" в данном случае будет компонентом, который определяет и генерирует событие. А роль "клиента" будет исполнять приложение, которое использует этот компонент, поскольку оно подключается к событию путем связывания имени некоторого заданного метода с указателем на нужный метод.

Несмотря на то что именно эта простая модель генерации событий и делает среду разработки в Delphi элегантной и простой в применении, ради этой простоты определенно приходится жертвовать некоторой мощностью. Например, не существует встроенного способа разрешить сразу нескольким клиентам "услышать" одно и то же событие — это называется многопунктовостью (`multicasting`). Кроме того, невозможно динамически получить описание типа для события, не написав некоторого RTTI-кода (который, возможно, и не пришлось бы использовать в приложении из-за привязки к конкретной версии).

События в автоматизации

Если о модели событий Delphi можно сказать, что она проста, но весьма ограниченна, то модель событий автоматизации отличается бо́льшими возможностями, за которые приходится "расплачиваться" большей сложностью. Как COM-программист, вы, вероятно, уже догадались, что в технологии автоматизации события реализуются с помощью интерфейсов. Но события определяются не на основе связки метод-событие, а только как часть интерфейса. Такой интерфейс часто называется интерфейсом событий или *исходящим интерфейсом* (outgoing interface). Это название связано с тем, что он реализуется не самим сервером (подобно другим интерфейсам), а его клиентами, причем методы интерфейса будут вызываться со стороны сервера. Как и все интерфейсы, интерфейсы событий имеют соответствующие идентификаторы интерфейсов (IID), которые определяют их уникальным образом. Кроме того, описание интерфейса событий (подобно прочим интерфейсам) содержится в библиотеке типов объекта автоматизации, связанной с компонентным классом этого объекта автоматизации.

В серверах, интерфейсы событий которых необходимо сделать доступными для клиентов, следует реализовать интерфейс `IConnectionPointContainer`. Этот интерфейс определен в модуле `ActiveX` следующим образом:

```
type
  IConnectionPointContainer = interface
    ['{B196B284-BAB4-101A-B69C-00AA00341D07}']
    function EnumConnectionPoints(out Enum: IEnumConnectionPoints):
      HResult; stdcall;
    function FindConnectionPoint(const iid: TIID;
      out cp: IConnectionPoint): HResult; stdcall;
  end;
```

В терминологии COM под *точкой подключения* (connection point) понимаются некоторые средства, предоставляющее программируемый доступ к исходящему интерфейсу. Если клиенту нужно выяснить, поддерживает ли сервер события, то для этого достаточно вызвать функцию `QueryInterface` для интерфейса `IConnectionPointContainer`. Если этот интерфейс присутствует, сервер может служить источником событий. Метод `EnumConnectionPoints()` интерфейса `IConnectionPointContainer` позволяет клиентам опросить все исходящие интерфейсы, поддерживаемые сервером. Для получения конкретного внешнего интерфейса клиенты могут использовать метод `FindConnectionPoint()`.

Нетрудно заметить, что метод FindConnectionPoint() обеспечивает клиента интерфейсом IConnectionPoint, который представляет нужный исходящий интерфейс. Интерфейс IConnectionPoint также определен в модуле ActiveX, и это объявление имеет следующий вид:

```
type
  IConnectionPoint = interface
    ['{B196B286-BAB4-101A-B69C-00AA00341D07}']
    function GetConnectionInterface(out iid: TIID): HResult; stdcall;
    function GetConnectionPointContainer(
      out cpc: IConnectionPointContainer): HResult; stdcall;
    function Advise(const unkSink: IUnknown; out dwCookie: Longint):
      HResult; stdcall;
    function Unadvise(dwCookie: Longint): HResult; stdcall;
    function EnumConnections(out Enum: IEnumConnections): HResult;
      stdcall;
  end;
```

Метод GetConnectionInterface() интерфейса IConnectionPoint предоставляет идентификатор (IID) исходящего интерфейса, поддерживаемый данной точкой подключения. А метод GetConnectionPointContainer() предоставляет интерфейс IConnectionPointContainer (он описан выше), который управляет этой точкой подключения. Весьма любопытен метод Advise. Именно этот метод воплощает в реальность магию связывания исходящих событий на сервере с интерфейсом events, реализованным клиентом. Первый параметр, передаваемый этому методу, представляет собой реализацию клиентом интерфейса events, второму же будет направлено извещение, определяющее это конкретное подключение. Функция Unadvise() предназначена для разрыва связи, установленной ранее между клиентом и сервером с помощью функции Advise(). Функция EnumConnections() позволяет клиенту опросить все активные в данный момент подключения, т.е. все подключения, выполненные функцией Advise().

Ввиду очевидной путаницы, которая может возникнуть, если описывать участников этих отношений просто как *клиент* и *сервер*, в технологии автоматизации было решено использовать некоторые другие термины, позволяющие однозначно определить, "кто есть кто". Поэтому реализацию исходящего интерфейса, содержащегося внутри клиента, договорились называть *стоком* (sink), а объект сервера, который генерирует события для клиента — *источником* (source).

Из всего выше сказанного, как мы надеемся, вполне очевидно, что события автоматизации имеют два преимущества перед событиями Delphi. А именно: они могут быть многопунктовыми, поскольку функцию IConnectionPoint.Advise() в этом случае можно вызывать более одного раза. И кроме того, события автоматизации являются самоописательными (благодаря использованию библиотеки типов и методов перечисления), поэтому ими можно управлять динамически.

События автоматизации в Delphi

Итак, все эти рассуждения звучат прекрасно, но как на самом деле заставить события автоматизации работать в среде Delphi? С удовольствием ответим. На данном этапе мы создадим приложение-сервер автоматизации, который предоставляет некоторый внешний интерфейс, и приложение-клиент, которое реализует сток для этого интерфейса. Учтите, что вовсе не требуется быть экспертом по части точек подключения, стоков, источников и прочих премудростей, чтобы добиться от Delphi желаемых результатов. Однако глубокое понимание того, что действительно происходит под покровами оболочек и инкапсулирующих функций Delphi, сослужит вам хорошую и, надеемся, полезную службу.

Сервер

Создание сервера начнем с создания нового приложения. В целях демонстрации мы создадим новое приложение, содержащее одну форму с полем текстового редактора (компонент TMemo), которое будет управляться клиентом — как показано на рис. 23.12.

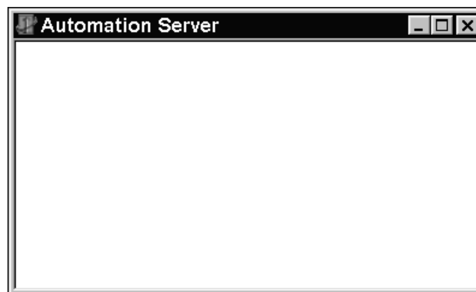


Рис. 23.12. Сервер автоматизации со своей главной формой Events

Затем к этому приложению добавим объект автоматизации. Для этого в главном меню Delphi выберем команду File⇨New⇨ActiveX⇨Automation Object, в результате чего откроется окно мастера объектов автоматизации Automation Object Wizard (см. рис. 23.4).

Обратите внимание на флажок опции Generate Event Support Code (Генерировать код поддержки событий) в окне мастера Automation Object Wizard. Данный флажок нужно установить, поскольку только в этом случае будет сгенерирован код, необходимый для активизации исходящего интерфейса в объекте автоматизации. Кроме того, при этом в библиотеку типов будет записана вся необходимая информация, связанная с исходящим интерфейсом. По щелчку на кнопке OK в диалоговом окне мастера открывается окно редактора библиотеки типов Type Library Editor. При этом в библиотеке типов уже присутствуют как интерфейс автоматизации, так и исходящий интерфейс (с именами IServerWithEvents и IServerWithEventsEvents соответственно). В интерфейс IServerWithEvents добавлены методы AddText() и Clear(), а в интерфейс IServerWithEventsEvents — методы OnTextChanged() и OnClear().

Нетрудно догадаться, что метод Clear() предназначен для очистки содержимого поля текстового редактора, а метод AddText() — для добавления в него еще одной строки текста. Событие OnTextChanged() происходит при изменении содержимого поля текстового редактора, а событие OnClear() — при очистке его поля. Заметьте также, что каждому из методов AddText() и OnTextChanged() передается один параметр типа WideString.

Первое, что нужно сделать — это реализовать методы AddText() и Clear(). Реализация этих методов выглядит следующим образом:

```
procedure TServerWithEvents.AddText(const NewText: WideString);
begin
  MainForm.Memo.Lines.Add(NewText);
end;

procedure TServerWithEvents.Clear;
begin
  MainForm.Memo.Lines.Clear;
  if FEvents <> nil then FEvents.OnClear;
end;
```

Скорее всего, вам уже знаком весь этот код, за исключением, разве что, последней строки в определении метода Clear(). В этой строке программы с помощью проверки переменной FEvents на неравенство значению nil гарантируется существование стока клиента, готового к приему данного события, которое в этом случае генерируется простым вызовом метода FEvents.OnClear().

Чтобы сформировать событие OnTextChanged(), необходимо сначала обработать событие OnChange поля текстового редактора. Сделаем это путем вставки строки кода в метод Initialized() объекта TServerWithEvents, связывающей данное событие с методом в объекте TServerWithEvents:

```
MainForm.Memo.OnChange := MemoChange;
```

Реализация метода MemoChange() имеет следующий вид:

```
procedure TServerWithEvents.MemoChange(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnTextChanged((Sender as TMemo).Text);
end;
```

При выполнении этого кода также проверяется готовность клиента к прослушиванию события. И если это событие ожидаемо (переменная FEvents не равна значению nil), оно генерируется с передачей методу FEvents.OnTextChanged в качестве параметра текста из поля текстового редактора.

Верьте или нет, но на этом с реализацией сервера покончено! Теперь займемся клиентом.

Клиент

В качестве клиента мы воспользуемся приложением с одной формой, содержащей компоненты TEdit, Tmemo и три компонента Tbutton, — как показано на рис. 23.13.

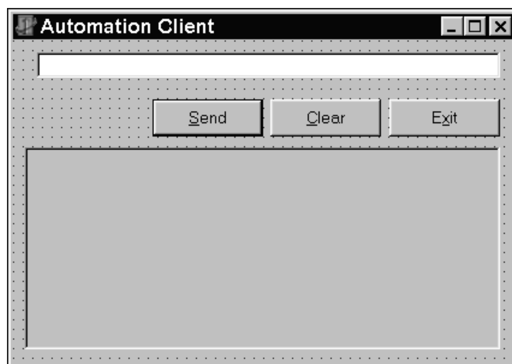


Рис. 23.13. Главная форма приложения клиента

В раздел uses модуля главной формы приложения клиента было добавлено имя модуля Server_TLB, благодаря чему у нас есть доступ к типам и методам, содержащимся внутри этого модуля. Объект главной формы приложения клиента TMainForm будет содержать поле, которое ссылается на сервер типа IServerWithEvents с именем FServer. В конструкторе формы TMainForm мы создадим экземпляр этого сервера, используя вспомогательный класс, описанный в модуле Server_TLB:

```
FServer := CoServerWithEvents.Create;
```

Следующий шаг состоит в реализации класса стока событий. Поскольку этот класс будет вызываться сервером через механизмы автоматизации, он должен реализовать интерфейс IDispatch (и, разумеется, интерфейс IUnknown). Вот как выглядит объявление типа для этого класса:

```
type
  TEventSink = class(TObject, IUnknown, IDispatch)
  private
    FController: TMainForm;
    { IUnknown }
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
    { IDispatch }
    function GetTypeInfoCount(out Count: Integer): HResult; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo):
      HResult; stdcall;
function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): HResult; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer;
      Flags: Word; var Params; VarResult, ExcepInfo, ArgErr: Pointer):
      HResult; stdcall;
public
    constructor Create(Controller: TMainForm);
  end;
```

Большинство методов интерфейсов IUnknown и IDispatch не реализованы, за исключением таких методов, как IUnknown.QueryInterface() и IDispatch.Invoke(). О них и пойдет речь ниже.

Метод QueryInterface() класса TEventSink реализован следующим образом:

```
function TEventSink.QueryInterface(const IID: TGUID; out Obj): HResult;
begin
  // Сначала ищем мою собственную реализацию интерфейса
  // (Я реализовал интерфейсы IUnknown и IDispatch).
  if GetInterface(IID, Obj) then
    Result := S_OK
  // Затем при поиске внешнего интерфейса выполняем рекурсивное обращение,
  // чтобы возвратить указатель на интерфейс IDispatch.
  else if IsEqualIID(IID, IServerWithEventsEvents) then
    Result := QueryInterface(IDispatch, Obj)
  // Во всех остальных случаях возвращаем признак ошибки.
  else
    Result := E_NOINTERFACE;
end;
```

По сути, этот метод возвращает экземпляр только в том случае, если запрашиваемым интерфейсом является IUnknown, IDispatch или IServerWithEventsEvents.

Вот как выглядит метод Invoke класса TEventSink:

```
function TEventSink.Invoke(DispID: Integer; const IID: TGUID;
  LocaleID: Integer; Flags: Word; var Params; VarResult, ExcepInfo,
  ArgErr: Pointer): HResult;
```

```
var
  V: OleVariant;
begin
  Result := S_OK;
  case DispID of
    1:
      begin
        // Первый параметр - новая строка
        V := OleVariant(TDispParams(Params).rgvarg^[0]);
        FController.OnServerMemoChanged(V);
      end;
    2: FController.OnClear;
  end;
end;
```

Метод `TEventSink.Invoke()` жестко закодирован для методов, имеющих значения идентификаторов `DispID` 1 или 2. Эти значения выбраны в приложении сервера для методов `OnTextChanged()` и `OnClear()` соответственно. Метод `OnClear()` имеет самую простую реализацию: в ответ на это событие он просто вызывает метод `OnClear()` главной формы клиента. Событие `OnTextChanged()` несколько сложнее: оно "выдергивает" параметр из массива `Params.rgvarg`, который передается этому методу в качестве параметра, и передает его методу `OnServerMemoChanged()` главной формы клиента. Заметьте, что поскольку нам известны число и тип параметров, мы можем упростить присваивания в исходном коде. В принципе, можно реализовать функцию `Invoke()` в общем виде, чтобы она вычисляла количество и типы параметров и помещала их в стек или в регистры до вызова соответствующей функции. Если вам это интересно, рассмотрите метод `TOleControl.InvokeEvent()` в модуле `OleCtrls`. Этот метод представляет логику приема событий для контейнера элементов управления ActiveX.

Методы `OnClear()` и `OnServerMemoChanged()` предназначены для выполнения манипуляций над содержимым `memo`-поля клиента. Вот как реализованы эти методы:

```
procedure TMainForm.OnServerMemoChanged(const NewText: string);
begin
  Memo.Text := NewText;
end;

procedure TMainForm.OnClear;
begin
  Memo.Clear;
end;
```

Теперь осталось лишь соединить сток событий с исходящим интерфейсом сервера-источника. Это легко осуществляется с помощью функции `InterfaceConnect()` (содержащейся в модуле `ComObj`), которая будет вызываться из конструктора главной формы:

```
InterfaceConnect(FServer, IServerWithEventsEvents, FEventSink, FCookie);
```

Первый параметр, передаваемый этой функции, представляет собой ссылку на объект-источник. Второй параметр — это идентификатор исходящего интерфейса (IID). Третий — интерфейс стока событий. Четвертый (и последний) параметр задает определитель (cookie) и является ссылкой — он будет заполнен вызывающей стороной.

"Приличия требуют", чтобы, наигравшись с событиями, клиент корректно завершил работу, вызвав функцию InterfaceDisconnect(). Эти действия реализуются в деструкторе главной формы:

```
InterfaceDisconnect(FEventSink, IServerWithEventsEvents, FCookie);
```

Запуск демонстрационного примера

Теперь, когда приложения клиента и сервера написаны, можно увидеть их в действии. Прежде чем запускать приложение клиента, не забудьте один раз запустить и закрыть приложение сервера (или запустить его с ключом /regserver), чтобы зарегистрировать этот сервер. На рис. 23.14 показаны результаты взаимодействия между клиентом, сервером, источником и стоком.

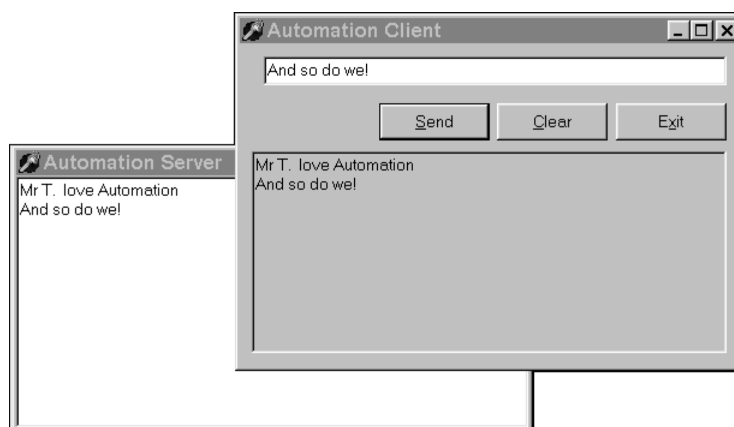


Рис. 23.14. Клиент автоматизации, манипулирующий сервером и получающий события

События с несколькими стоками

Хотя описанная выше методика прекрасно работает при генерации событий, предназначенных для одного клиента, она не так уж хороша при работе с несколькими клиентами. Однако ситуации подключения к серверу нескольких клиентов возникают довольно часто, и в этом случае необходимо генерировать события для всех клиентов. К счастью, для этого потребуется добавить лишь несколько строк программы. Чтобы сгенерировать события для нескольких клиентов, нужно написать код, опрашивающий все имеющие подключения и вызывающий соответствующий метод стока. Это можно реализовать путем внесения нескольких изменений в предыдущий пример.

Пойдем по порядку. Чтобы поддерживать несколько подключений клиентов в точке подключения, необходимо в качестве параметра Kind метода TConnectionPoints.CreateConnectionPoint() передать значение ckMulti. Этот метод вызывается из метода Initialize() объекта автоматизации:

```
FConnectionPoints.CreateConnectionPoint(AutoFactory.EventIID, ckMulti,
  EventConnect);
```

Перед построением перечисления подключений нужно получить ссылку на интерфейс IConnectionPointContainer. Из интерфейса IConnectionPointContainer можно получить интерфейс IConnectionPoint, представляющий собой исходящий интерфейс, а с помощью ме-

тода IConnectionPoint.EnumConnections() можно получить интерфейс IEnumConnections, который можно будет использовать для построения перечисления подключений. Вся эта логика "укладывается" в следующем методе:

```
function TServerWithEvents.GetConnectionEnumerator: IEnumConnections;
var
  Container: IConnectionPointContainer;
  CP: IConnectionPoint;
begin
  Result := nil;
  OleCheck(QueryInterface(IConnectionPointContainer, Container));
  OleCheck(Container.FindConnectionPoint(AutoFactory.EventIID, CP));
  CP.EnumConnections(Result);
end;
```

После получения интерфейса перечисления подключений, для каждого из клиентов осуществляется вызов его стока посредством последовательного перебора всех имеющихся подключений. Эта логика демонстрируется в следующем коде, который генерирует событие OnTextChanged():

```
procedure TServerWithEvents.MemoChange(Sender: TObject);
var
  EC: IEnumConnections;
  ConnectData: TConnectData;
  Fetched: Cardinal;
begin
  EC := GetConnectionEnumerator;
  if EC <> nil then
  begin
    while EC.Next(1, ConnectData, @Fetched) = S_OK do
      if ConnectData.pUnk <> nil then
        (ConnectData.pUnk as
        IServerWithEventsEvents).OnTextChanged((Sender as TMemo).Text);
  end;
end;
```

Наконец, чтобы позволить клиентам подключиться к единственному объекту автоматизации, необходимо вызвать функцию COM API RegisterActiveObject(). Этой функции передаются следующие параметры: интерфейс IUnknown для объекта, идентификатор класса объекта (CLSID), флажок "силы" регистрации (сильная (сервер должен иметь метод AddRef) или слабая (без метода AddRef)) и дескриптор, который возвращается по ссылке:

```
RegisterActiveObject(Self as IUnknown, Class_ServerWithEvents,
  ACTIVEOBJECT_WEAK, FObjRegHandle);
```

Все эти программные фрагменты связываются воедино в модуле ServAuto, полный исходный текст которого представлен в листинге 23.10.

Листинг 23.10. Модуль ServAuto.pas

```
unit ServAuto;

interface

uses
```

```
  ComObj, ActiveX, AxCtrls, Server_TLB;

type
  TServerWithEvents = class(TAutoObject, IConnectionPointContainer,
    IServerWithEvents)
  private
    { Закрытые объявления }
    FConnectionPoints: TConnectionPoints;
    FObjRegHandle: Integer;
    procedure MemoChange(Sender: TObject);
  protected
    { Защищенные объявления }
    procedure AddText(const NewText: WideString); safecall;
    procedure Clear; safecall;
    function GetConnectionEnumerator: IEnumConnections;
    property ConnectionPoints: TConnectionPoints read FConnectionPoints
      implements IConnectionPointContainer;
  public
    destructor Destroy; override;
    procedure Initialize; override;
  end;

implementation

uses Windows, ComServ, ServMain, SysUtils, StdCtrls;

destructor TServerWithEvents.Destroy;
begin
  inherited Destroy;
  RevokeActiveObject(FObjRegHandle, nil);  // Удаление объекта из таблицы ROT
end;

procedure TServerWithEvents.Initialize;
begin
  inherited Initialize;
  FConnectionPoints := TConnectionPoints.Create(Self);
  if AutoFactory.EventTypeInfo <> nil then
    FConnectionPoints.CreateConnectionPoint(AutoFactory.EventIID, ckMulti,
      EventConnect);
  // Направляем событие OnChange memo-поля главной формы на метод MemoChange:
  MainForm.Memo.OnChange := MemoChange;
  // Регистрируем этот объект с помощью COM-таблицы Running Object Table (ROT),
  // чтобы другие клиенты могли подключиться к этому экземпляру.
  RegisterActiveObject(Self as IUnknown, Class_ServerWithEvents,
    ACTIVEOBJECT_WEAK, FObjRegHandle);
end;

procedure TServerWithEvents.Clear;
var
  EC: IEnumConnections;
```

```
  ConnectData: TConnectData;
  Fetched: Cardinal;
begin
  MainForm.Memo.Lines.Clear;
  EC := GetConnectionEnumerator;
  if EC <> nil then
  begin
    while EC.Next(1, ConnectData, @Fetched) = S_OK do
      if ConnectData.pUnk <> nil then
        (ConnectData.pUnk as IServerWithEventsEvents).OnClear;
  end;
end;

procedure TServerWithEvents.AddText(const NewText: WideString);
begin
  MainForm.Memo.Lines.Add(NewText);
end;

procedure TServerWithEvents.MemoChange(Sender: TObject);
var
  EC: IEnumConnections;
  ConnectData: TConnectData;
  Fetched: Cardinal;
begin
  EC := GetConnectionEnumerator;
  if EC <> nil then
  begin
    while EC.Next(1, ConnectData, @Fetched) = S_OK do
      if ConnectData.pUnk <> nil then
        (ConnectData.pUnk as
          IServerWithEventsEvents).OnTextChanged((Sender as TMemo).Text);
  end;
end;

function TServerWithEvents.GetConnectionEnumerator: IEnumConnections;
var
  Container: IConnectionPointContainer;
  CP: IConnectionPoint;
begin
  Result := nil;
  OleCheck(QueryInterface(IConnectionPointContainer, Container));
  OleCheck(Container.FindConnectionPoint(AutoFactory.EventIID, CP));
  CP.EnumConnections(Result);
end;

initialization
  TAutoObjectFactory.Create(ComServer, TServerWithEvents,
    Class_ServerWithEvents, ciMultiInstance, tmApartment);
end.
```

В программное обеспечение клиентов также нужно внести небольшие изменения с той целью, чтобы позволить клиентам подключаться к активному экземпляру, если он уже работает. Эти действия реализуются с помощью функции COM API `GetActiveObject()`:

```
procedure TMainForm.FormCreate(Sender: TObject);
var
  ActiveObj: IUnknown;
begin
  // Получаем активный объект, если он доступен, в противном
  // случае создаем новый
  GetActiveObject(Class_ServerWithEvents, nil, ActiveObj);
  if ActiveObj <> nil then FServer := ActiveObj as IServerWithEvents
  else FServer := CoServerWithEvents.Create;
  FEventSink := TEventSink.Create(Self);
  InterfaceConnect(FServer, IServerWithEventsEvents, FEventSink, FCookie);
end;
```

На рис. 23.15 показаны клиенты, принимающие события от одного-единственного сервера.

Рис. 23.15. Несколько клиентов, манипулирующих одним и тем же сервером и принимающих его события

Коллекции автоматизации

Прямо скажем: нас, программистов, со всех сторон окружают программные объекты, которые используются в качестве контейнеров для других программных объектов. Задумайтесь, как велико их разнообразие — будь то массив, список (компонент `TList`), коллекция (компонент `TCollection`), класс контейнера шаблона C++ или вектор Java. Кажется, что мы постоянно только то и делаем, что подыскиваем оптимальный вариант пресловутой мышеловки для про-

граммных объектов, которая бы наилучшим образом справлялась с задачей хранения других программных объектов. Если оценить время, затраченное на создание идеального контейнерного класса, то станет ясно, что это одна из самых важных проблем, занимающих умы разработчиков. А почему бы и нет? Подобное логическое разделение контейнера и его содержимого помогает лучше организовать наши алгоритмы и создает вполне приемлемое соответствие реальному миру (в корзинке могут лежать яйца, в кармане — деньги, на стоянке можно спокойно оставлять автомобили и т.д.). При изучении нового языка или модели разработки всегда приходится знакомиться с "новым способом" управления группами некоторых элементов. Это и есть та самая мысль, к которой мы вас подводили: подобно любой другой модели разработки программных продуктов, модель COM также имеет свои способы управления собственными разновидностями групп элементов. Чтобы добиться эффективности в разработке COM-приложений, необходимо знать, как обращаться с такими объектами.

При работе с интерфейсом IDispatch модель COM определяет два основных метода, с помощью которых представляется категория контейнера: массивы и коллекции. Если вам уже приходилось иметь дело с технологией автоматизации или элементами ActiveX в Delphi, то, скорее всего, с массивами вы уже знакомы. В Delphi создание массивов автоматизации не вызывает сложностей. Это осуществляется путем добавления свойства массива в потомок интерфейса IDispatch или в диспинтерфейс, как показано в следующем примере:

```
type
  IMyDisp = interface(IDispatch)
    function GetProp(Index: Integer): Integer; safecall;
    procedure SetProp(Index, Value: Integer); safecall;
    property Prop[Index: Integer]: Integer read GetProp write SetProp;
  end;
```

Массивы полезны во многих ситуациях, но им свойственны некоторые ограничения. Например, о массивах стоит говорить, когда есть данные, к которым можно получить доступ логическим способом, использующим фиксированный индекс, например, при доступе к строкам в типе IStrings. Однако, если природа данных такова, что отдельные элементы часто удаляются, добавляются или перемещаются, массив оказывается не слишком удобным контейнером. Классический пример — группа активных окон. Поскольку окна постоянно создаются, разрушаются и изменяют свою z-упорядоченность, не существует надежного критерия для определения порядка размещения окон в массиве.

Для решения этой проблемы и были разработаны коллекции. Они позволяют манипулировать рядом элементов таким способом, который не предполагает какого бы то ни было порядка следования или номеров элементов. Необычность коллекций в том, что в действительности не существует объекта или интерфейса *коллекции*, вместо этого коллекция представляется как пользовательский интерфейс IDispatch, который соблюдает некоторый набор правил и предписаний. Итак, чтобы интерфейс IDispatch можно было бы квалифицировать как коллекцию, необходимо твердо придерживаться следующих правил.

- Коллекции должны содержать свойство _NewEnum, возвращающее интерфейс IUnknown объекту, который поддерживает интерфейс IEnumVARIANT. Интерфейс IEnumVARIANT будет использован для перечисления элементов в коллекции. Обратите внимание на то, что имя этого свойства должно начинаться с символа подчеркивания, а само свойство должно быть отмечено как зарегистрированное в библиотеке типов. Диспетчерский идентификатор DispID для свойства _NewEnum должен быть равен значению DISPID_NEWENUM (-4), и он будет определен в редакторе библиотеки типов Delphi следующим образом:

```
function _NewEnum: IUnknown [propget, dispid $FFFFFFFC, restricted];
  safecall;
```

Такие языки, как Visual Basic, которые поддерживают конструкцию For Each, будут использовать этот метод для получения интерфейса IEnumVARIANT, необходимого для перечисления элементов коллекции. (Подробности ниже.)

- Коллекции должны содержать метод Item(), который возвращает элемент из коллекции на базе индекса. Идентификатор диспетчера DispID для этого метода должен быть равен 0, и его следует отметить признаком *стандартного элемента коллекции* (default collection element). Если бы нам нужно было реализовать коллекцию указателей на интерфейсы IFoo, определение для метода в редакторе библиотеки типов выглядело бы примерно так:

```
function Item(Index: Integer): IFoo [propget, dispid $00000000,
  defaultcollelem]; safecall;
```

  Обратите также внимание на то, что параметр Index вполне может иметь тип OleVariant, чтобы рассматриваемый элемент можно было индексировать с помощью значения типа Integer, WideString или другого какого-либо типа.

- Коллекции должны содержать свойство Count, которое возвращает количество элементов в коллекции. Этот метод обычно определяется в редакторе библиотеки типов следующим образом:

```
function Count: Integer [propget, dispid $00000001]; safecall;
```

В дополнение к вышеизложенным правилам, при создании собственных объектов коллекций также полезно будет следовать приведенным ниже рекомендациям.

- Свойство или метод, возвращающие коллекцию, должны иметь имя, представляющее собой форму множественного числа, образованного от имени элементов коллекции. Например, если бы у вас было свойство, возвращающее коллекцию элементов списка, в качестве имени этого свойства подошло бы слово Items (если элемент коллекции имеет имя Item). Аналогично элемент с именем Foot (нога) будет входить в свойство-коллекцию с именем Feet (ноги). В тех редких случаях, когда форма множественного числа для нужного слова совпадает с формой единственного числа (например, для коллекции элементов fish или deer), имя свойства коллекции должно состоять из имени элемента, к которому присоединяется слово "Collection" (FishCollection или DeerCollection).
- Коллекции, которые поддерживают добавление элементов, должны осуществлять это с помощью метода Add(). Параметры для этого метода варьируются в зависимости от реализации, но вам стоит подумать о передаче параметров, которые определяют начальную позицию нового элемента внутри коллекции. Метод Add() обычно возвращает ссылку на элемент, добавленный в коллекцию.
- Коллекции, которые поддерживают удаление элементов, должны осуществлять это с помощью метода Remove(). Этому методу достаточно передать один параметр, который идентифицирует индекс удаляемого элемента, причем поведение этого индекса семантически должно совпадать с методом Item().

Реализация в Delphi

Если вам когда-либо приходилось создавать элементы управления ActiveX в Delphi, то вы могли заметить, что в раскрывающемся списке мастера ActiveX Control Wizard перечислено меньше элементов управления, чем представлено в палитре компонентов IDE. Дело в том, что компания Inprise не допускает отображения в списке мастера некоторых элементов

управления благодаря использованию функции RegisterNonActiveX(). Примером элемента управления, который доступен в палитре компонентов, но не в окне мастера, может служить элемент TListView, расположенный во вкладке Win32 палитры. Причина "сокрытия" мастером элемента управления TListView состоит в том, что мастер "не знает", что ему делать со свойством Items данного компонента, которое имеет тип TListItems. Поскольку мастер не знает, как охватить этот тип свойства элементом управления ActiveX, такой элемент просто исключается из списка мастера, чтобы запретить пользователю создавать совершенно бесполезную оболочку для элемента управления ActiveX.

Однако в случае с элементом TListView функция RegisterNonActiveX() вызывается с признаком axrComponentOnly, который означает, что потомок компонента TListView будет помещен в список мастера ActiveX Control Wizard. Предприняв некоторые вспомогательные меры по созданию, казалось бы, бездействующего потомка компонента TListView с именем TListView2 и добавив его в палитру, мы сможем затем создать элемент управления ActiveX, который инкапсулирует элемент управления TListView. Конечно, затем мы все равно столкнемся с той же самой проблемой отказа мастера от генерации оболочек для свойства Items и вынуждены будем признать бесполезность созданного элемента управления ActiveX. К счастью, при создании элемента управления ActiveX нас никто не принуждает ограничиться кодом, сгенерированным мастером, и на этом этапе вы вольны сами заняться свойством Items, чтобы сделать элемент управления полезным. Возможно, вы уже начали догадываться, что идеальным способом для инкапсуляции свойства Items элемента TListView является коллекция.

Чтобы реализовать эту коллекцию элементов списка, необходимо создать новые объекты, представляющие как отдельный элемент, так и всю коллекцию, а затем добавить новое свойство в стандартный интерфейс элемента управления ActiveX, которое будет возвращать данную коллекцию. Начнем с определения объекта, представляющего элемент списка, которому присвоим имя ListItem. Первый шаг к созданию объекта ListItem состоит в разработке нового объекта автоматизации с помощью пиктограммы, расположенной во вкладке ActiveX диалогового окна New Items. После создания этого объекта заполним его свойства и методы в редакторе библиотеки типов. В целях демонстрации добавим в элемент TListView свойства Caption, Index, Checked и SubItems. Аналогично, создадим другой новый объект автоматизации для самой коллекции и присвоим ему имя ListItems. Пополним его описанными выше методами _NewEnum, Item(), Count(), Add() и Remove(). Наконец, в стандартный интерфейс элемента управления ActiveX добавим новое свойство с именем Items, которое будет возвращать коллекцию.

После того как интерфейсы IListItem и IListItems будут полностью определены в редакторе библиотеки типов, потребуется провести небольшое ручное вмешательство в файлы реализации, сгенерированные для этих объектов. Стандартным родительским классом для нового объекта автоматизации является класс TAutoObject; однако наши новые объекты будут создаваться только внутренне (т.е. без помощи фабрики класса), поэтому мы вручную изменим тип предка и укажем в качестве родителя компонент TAutoInfObject, который больше подходит для внутренне создаваемых объектов автоматизации. Кроме того, поскольку эти объекты не создаются с помощью фабрики класса, из модулей можно удалить код инициализации, реализующий ненужные нам фабрики.

Теперь, когда вся инфраструктура приняла надлежащий вид, пора приняться за реализацию объектов ListItem и ListItems. Объект ListItem — самый простой, поскольку он представляет собой незатейливую оболочку вокруг элемента списка. Текст модуля, содержащего этот объект, представлен в листинге 23.11.

Листинг 23.11. Оболочка для элемента `Listview`

```
unit LVItem;

interface

uses
  ComObj, ActiveX, ComCtrls, LVCtrl_TLB, StdVcl, AxCtrls;

type
  TListItem = class(TAutoIntfObject, IListItem)
  private
    FListItem: ComCtrls.TListItem;
  protected
    function Get_Caption: WideString; safecall;
    function Get_Index: Integer; safecall;
    function Get_SubItems: IStrings; safecall;
    procedure Set_Caption(const Value: WideString); safecall;
    procedure Set_SubItems(const Value: IStrings); safecall;
    function Get_Checked: WordBool; safecall;
    procedure Set_Checked(Value: WordBool); safecall;
  public
    constructor Create(AOwner: ComCtrls.TListItem);
  end;

implementation

uses ComServ;

constructor TListItem.Create(AOwner: ComCtrls.TListItem);
begin
  inherited Create(ComServer.TypeLib, IListItem);
  FListItem := AOwner;
end;

function TListItem.Get_Caption: WideString;
begin
  Result := FListItem.Caption;
end;

function TListItem.Get_Index: Integer;
begin
  Result := FListItem.Index;
end;

function TListItem.Get_SubItems: IStrings;
begin
  GetOleStrings(FListItem.SubItems, Result);
end;
```

```
procedure TListItem.Set_Caption(const Value: WideString);
begin
  FListItem.Caption := Value;
end;

procedure TListItem.Set_SubItems(const Value: IStrings);
begin
  SetOleStrings(FListItem.SubItems, Value);
end;

function TListItem.Get_Checked: WordBool;
begin
  Result := FListItem.Checked;
end;

procedure TListItem.Set_Checked(Value: WordBool);
begin
  FListItem.Checked := Value;
end;

end.
```

Заметьте, что в конструктор передается объект ComCtrls.TListItem, который будет служить в качестве элемента списка, контролируемого данным объектом автоматизации.

Реализация объекта коллекции ListItems сложнее, но не намного. Во-первых, поскольку этот объект должен обеспечивать объектную поддержку типа IEnumVARIANT, чтобы реализовать свойство _NewEnum, тип IEnumVARIANT поддерживается прямо в этом объекте. Следовательно, класс TListItems поддерживает оба интерфейса: IListItems и IEnumVARIANT. Интерфейс IEnumVARIANT содержит четыре метода, которые описаны Windows табл. 23.2.

Таблица 23.2. Методы интерфейса IEnumVARIANT

Метод	Назначение
Next	Считывает следующие *n* элементов в коллекции
Skip	Пропускает *n* элементов в коллекции
Reset	Восстанавливает текущий элемент назад к первому элементу в коллекции
Clone	Создает копию интерфейса IEnumVARIANT

Исходный текст модуля, содержащего объект ListItems, представлен в листинге 23.12.

Листинг 23.12. Оболочка для элементов Listview

```
unit LVItems;

interface

uses
  ComObj, Windows, ActiveX, ComCtrls, LVCtrl_TLB;
```

```
type
  TListItems = class(TAutoIntfObject, IListItems, IEnumVARIANT)
  private
    FListItems: ComCtrls.TListItems;
    FEnumPos: Integer;
  protected
    { Методы IListItems }
    function Add: IListItem; safecall;
    function Get_Count: Integer; safecall;
    function Get_Item(Index: Integer): IListItem; safecall;
    procedure Remove(Index: Integer); safecall;
    function Get__NewEnum: IUnknown; safecall;
    { Методы IEnumVariant }
    function Next(celt: Longint; out elt;  pceltFetched: PLongint): HResult;
      stdcall;
    function Skip(celt: Longint): HResult; stdcall;
    function Reset: HResult; stdcall;
    function Clone(out Enum: IEnumVariant): HResult; stdcall;
public
    constructor Create(AOwner: ComCtrls.TListItems);
  end;

implementation

uses ComServ, LVItem;

{ TListItems }

constructor TListItems.Create(AOwner: ComCtrls.TListItems);
begin
  inherited Create(ComServer.TypeLib, IListItems);
  FListItems := AOwner;
end;

{ TListItems.IListItems }

function TListItems.Add: IListItem;
begin
  Result := LVItem.TListItem.Create(FListItems.Add);
end;

function TListItems.Get__NewEnum: IUnknown;
begin
  Result := Self;
end;

function TListItems.Get_Count: Integer;
begin
  Result := FListItems.Count;
end;
```

```
function TListItems.Get_Item(Index: Integer): IListItem;
begin
  Result := LVItem.TListItem.Create(FListItems[Index]);
end;

procedure TListItems.Remove(Index: Integer);
begin
  FListItems.Delete(Index);
end;

{ TListItems.IEnumVariant }

function TListItems.Clone(out Enum: IEnumVariant): HResult;
begin
  Enum := nil;
  Result := S_OK;
  try
    Enum := TListItems.Create(FListItems);
  except
    Result := E_OUTOFMEMORY;
  end;
end;

function TListItems.Next(celt: Integer; out elt; pceltFetched: PLongint):
  HResult;
var
  V: OleVariant;
  I: Integer;
begin
  Result := S_FALSE;
  try
    if pceltFetched <> nil then pceltFetched^ := 0;
    for I := 0 to celt - 1 do
    begin
      if FEnumPos >= FListItems.Count then Exit;
      V := Get_Item(FEnumPos);
      TVariantArgList(elt)[I] := TVariantArg(V);
      // Прием для защиты переменной типа variant от сбора мусора, так как он
      // должен оставаться "живым", поскольку является частью массива elt
      TVarData(V).VType := varEmpty;
      TVarData(V).VInteger := 0;
      Inc(FEnumPos);
      if pceltFetched <> nil then Inc(pceltFetched^);
    end;
  except
  end;
  if (pceltFetched = nil) or ((pceltFetched <> nil) and
    (pceltFetched^ = celt)) then
Result := S_OK;
end;
```

```
function TListItems.Reset: HResult;
begin
  FEnumPos := 0;
  Result := S_OK;
end;

function TListItems.Skip(celt: Integer): HResult;
begin
  Inc(FEnumPos, celt);
  Result := S_OK;
end;

end.
```

Единственным в этом модуле методом с нетривиальной реализацией является метод Next(). Параметр celt метода Next() указывает, сколько элементов должно быть считано. Параметр elt содержит массив типа TVarArgs, содержащий по крайней мере elt элементов. При возврате параметр pceltFetched (если не равен значению nil) должен хранить реальное число выбранных элементов. Этот метод возвращает значение S_OK, если число возвращаемых элементов совпадает с числом затребованных элементов; в противном случае возвращается значение S_FALSE. Логика этого метода состоит в сканировании массива elt и присвоении очередному элементу массива объекта типа TVarArg, представляющего элемент коллекции. Обратите внимание на маленький трюк, который выполняется для очистки типа OleVariant после присваивания его массиву. Тем самым гарантируется, что этот массив не попадет в "отходы". Если бы это не было сделано, содержимое массива elt могло бы потенциально "испортиться", если объекты, на которые ссылается переменная V, освободятся в процессе завершения работы типа OleVariant.

Подобно классу TListItem, конструктору класса TListItems передается в качестве параметра объект ComCtrls.TListItems, который участвует в реализации различных методов этого класса.

Наконец, для завершения реализации элемента управления ActiveX нужно добавить логику для управления свойством Items. Прежде всего к объекту следует добавить поле для хранения коллекции:

```
type
  TListViewX = class(TActiveXControl, IListViewX)
  private
    ...
    FItems: IListItems;
    ...
  end;
```

Затем в методе InitializeControl() присваиваем переменной FItems новый экземпляр объекта класса TListItems:

```
FItems := LVItems.TListItems.Create(FDelphiControl.Items);
```

Наконец, метод Get_Items() можно реализовать с помощью простого возврата значения переменной FItems:

```
function TListViewX.Get_Items: IListItems;
begin
  Result := FItems;
end;
```

Для реальной проверки работоспособности этой коллекции нужно загрузить элемент управления в среде Visual Basic 6 и попытаться использовать конструкцию `For Each` для работы с этой коллекцией. На рис. 23.16 показан результат запуска простого тестового VB-приложения.

Рис. 23.16. Приложение Visual Basic тестирует созданную коллекцию

На рис. 23.16 вы видите две кнопки. Кнопка Command1 предназначена для добавления элементов в `listview`-коллекцию, а кнопка `Command2` — для опроса всех элементов коллекции с помощью конструкции `For Each` и добавления восклицательных знаков к каждому заголовку. Вот как выглядит реализация этих методов:

```
Private Sub Command1_Click()
  ListViewX1.Items.Add.Caption = "Delphi"
End Sub

Private Sub Command2_Click()
  Dim Item As ListItem
  Set Items = ListViewX1.Items
  For Each Item In Items
   Item.Caption = Item.Caption + " Rules!!"
  Next
End Sub
```

Несмотря на чувства, которые питают некоторые Delphi-программисты к Visual Basic, мы должны помнить, что приложения Visual Basic — главный потребитель элементов управления ActiveX, и очень важно гарантировать надлежащее функционирование наших элементов управления в этой среде.

Коллекции обеспечивают мощную функциональность, которая может позволить вашим элементам управления и серверам автоматизации более естественно действовать в мире COM. Поскольку коллекции чрезвычайно трудны для реализации, стоит потратить время на их освоение — чтобы у вас вошло в привычку использовать их в любых подходящих случаях. К сожалению, вполне вероятно, что, как только вы станете с коллекциями "на ты", вскоре найдется кто-нибудь поумнее и создаст еще более новый и более мощный объект контейнера для модели COM.

Новые типы интерфейсов в библиотеке типов

Как и подобает каждому приличному разработчику Delphi, для определения новых экземпляров объектов автоматизации мы использовали редактор библиотеки типов. Однако можно легко попасть в ситуацию, когда один из методов для нового интерфейса включает параметр

типа интерфейса COM, который не поддерживается по умолчанию в редакторе библиотеки типов. А поскольку редактор библиотеки типов не позволяет работать с типами, о которых он не знает, как же тогда завершить такое определение метода?

Прежде чем перейти к разъяснениям, важно понять причины такого поведения редактора библиотеки типов. Если при создании нового метода в редакторе библиотеки типов просмотреть типы, доступные в разделе Type вкладки Parameters, вы увидите такие интерфейсы, как IDataBroker, IDispatch, IEnumVARIANT, IFont, IPicture, IProvider, IStrings и IUnknown. Почему же доступны только эти интерфейсы? Что делает их такими особенными? На самом деле в них нет ничего необычного — просто они являются типами, которые определены в библиотеках типов, используемых данной библиотекой типов. По умолчанию библиотека типов Delphi автоматически использует библиотеку типов Borland Standard VCL и библиотеку типов OLE Automation. Для определения конфигурации своей библиотеки типов выделите корневой узел в изображении дерева на левой панели редактора библиотеки типов и перейдите во вкладку Uses на правой панели. Типы, содержащиеся в библиотеках типов, используемых вашей библиотекой, станут автоматически доступными в раскрывающемся списке редактора библиотеки типов.

Зная все это, нетрудно догадаться о том, что, если интерфейс, который вы хотите использовать в качестве рассматриваемого параметра метода, определен в библиотеке типов, можно просто использовать эту библиотеку типов, — и проблема будет решена. Но что, если интерфейс не определен в библиотеке типов? Ведь существует довольно много COM-интерфейсов, которые определяются только инструментальными средствами (SDK) в заголовочных или IDL-файлах и не содержатся в библиотеках типов. В этом случае лучше всего определить параметр метода с помощью типа IUnknown. Интерфейс типа IUnknown может быть опрошен с помощью метода QueryInterface непосредственно в реализации метода, что позволит определить конкретный тип интерфейса, с которым предстоит работать. Необходимо также обязательно описать этот параметр метода как тип IUnknown, что должно обеспечить поддержку соответствующего интерфейса. Следующий фрагмент кода служит примером реализации такого метода:

```
procedure TSomeClass.SomeMethod(SomeParam: IUnknown);
var
  Intf: ISomeComInterface;
begin
  Intf := SomeParam as ISomeComInterface;
  // Остальная часть реализации метода
end;
```

Необходимо также знать, что интерфейс, к которому вы "приводите" тип IUnknown, должен быть интерфейсом, для которого модели COM известен способ передачи параметров (маршалинга). Это значит, что он должен быть либо определен в какой-нибудь библиотеке типов, либо должен быть типом, совместимым со стандартным маршалером автоматизации, либо рассматриваемый COM-сервер должен предоставить библиотеку DLL с заместителем-заглушкой (proxy/stub), способной выполнить маршалинг интерфейса.

Обмен двоичными данными

Рано или поздно, но вам обязательно потребуется выполнить обмен блоками двоичных данных между клиентом и сервером автоматизации. Поскольку модель COM не поддерживает операций обмена обычными указателями, решить задачу с помощью привычной операции передачи указателя не удастся. Поэтому воспользуемся другим, более сложным способом.

Самый простой способ обмена двоичными данными между клиентами и серверами автоматизации состоит в использовании массива байтов типа SafeArray. Delphi успешно инкапсулирует массивы этого типа в переменных типах OleVariant. Достаточно изощренный пример подробных действий показан в листингах 23.13 и 23.14, содержащих тексты модулей клиента и сервера, в которых поля текстового редактора используются для демонстрации методов передачи двоичных данных, представленных в виде массивов байт типа SafeArray.

Листинг 23.13. Модуль сервера

```
unit ServObj;

interface

uses
  ComObj, ActiveX, Server_TLB;

type
  TBinaryData = class(TAutoObject, IBinaryData)
  protected
    function Get_Data: OleVariant; safecall;
    procedure Set_Data(Value: OleVariant); safecall;
  end;

implementation

uses ComServ, ServMain;

function TBinaryData.Get_Data: OleVariant;
var
  P: Pointer;
  L: Integer;
begin
  // Переносим данные из memo-поля в массив
  L := Length(MainForm.Memo.Text);
  Result := VarArrayCreate([0, L - 1], varByte);
  P := VarArrayLock(Result);
  try
    Move(MainForm.Memo.Text[1], P^, L);
  finally
    VarArrayUnlock(Result);
  end;
end;

procedure TBinaryData.Set_Data(Value: OleVariant);
var
  P: Pointer;
  L: Integer;
  S: string;
begin
  // Переносим данные из массива в memo-поле
```

```
  L := VarArrayHighBound(Value, 1) - VarArrayLowBound(Value, 1) + 1;
  SetLength(S, L);
  P := VarArrayLock(Value);
  try
    Move(P^, S[1], L);
  finally
    VarArrayUnlock(Value);
  end;
  MainForm.Memo.Text := S;
end;

initialization
  TAutoObjectFactory.Create(ComServer, TBinaryData, Class_BinaryData,
    ciSingleInstance, tmApartment);
end.
```

Листинг 23.14. Модуль клиента

```
unit CliMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, Server_TLB;

type
  TMainForm = class(TForm)
    Memo: TMemo;
    Panel1: TPanel;
    SetButton: TButton;
    GetButton: TButton;
    OpenButton: TButton;
    OpenDialog: TOpenDialog;
    procedure OpenButtonClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure SetButtonClick(Sender: TObject);
    procedure GetButtonClick(Sender: TObject);
  private
    FServer: IBinaryData;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.FormCreate(Sender: TObject);
```

```
begin
  FServer := CoBinaryData.Create;
end;

procedure TMainForm.OpenButtonClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    Memo.Lines.LoadFromFile(OpenDialog.FileName);
end;

procedure TMainForm.SetButtonClick(Sender: TObject);
var
  P: Pointer;
  L: Integer;
  V: OleVariant;
begin
  // Отправляем memo-данные серверу
  L := Length(Memo.Text);
  V := VarArrayCreate([0, L - 1], varByte);
  P := VarArrayLock(V);
  try
    Move(Memo.Text[1], P^, L);
  finally
    VarArrayUnlock(V);
  end;
  FServer.Data := V;
end;

procedure TMainForm.GetButtonClick(Sender: TObject);
var
  P: Pointer;
  L: Integer;
  S: string;
  V: OleVariant;
begin
  // Получаем memo-данные сервера
  V := FServer.Data;
  L := VarArrayHighBound(V, 1) - VarArrayLowBound(V, 1) + 1;
SetLength(S, L);
  P := VarArrayLock(V);
  try
    Move(P^, S[1], L);
  finally
    VarArrayUnlock(V);
  end;
  Memo.Text := S;
end;

end.
```

Языковая поддержка для COM

В разговорах, касающихся разработки COM-приложений в Delphi, часто можно услышать о сильной поддержке, предоставляемой языком Object Pascal модели COM. С этим трудно спорить, если учесть, что в язык встроена поддержка таких элементов, как интерфейсы, варианты и широкие строки. И все же, что же реально означает “поддержка, встроенная в язык”? Как работают эти средства и в чем природа их зависимости от функций COM API? В данном разделе мы рассмотрим, как на внутреннем уровне все эти вещи объединяются с целью обеспечения поддержки COM в языке Object Pascal, и разберемся в некоторых деталях реализации этих языковых средств.

Как упоминалось выше, все средства поддержки COM в языке Object Pascal можно разделить на три основные категории.

- Типы данных `Variant` и `OleVariant`, которые инкапсулируют определенные в модели COM вариантные записи, массивы `SafeArray` и автоматизацию с поздним связыванием.
- Тип данных `WideString`, который инкапсулирует определенные в модели COM `BSTR`-строки.
- Типы `Interface` и `dispinterface`, которые инкапсулируют интерфейсы COM и автоматизацию с ранним связыванием, а также автоматизацию со связыванием на уровне идентификаторов (ID-bound Automation).

Разработчики OLE со стажем (со времен Delphi 2), вероятно, заметили, что зарезервированное слово `automated`, благодаря которому могли создаваться серверы автоматизации позднего связывания, практически игнорируется. Поскольку эта функция была отодвинута на второй план средствами “настоящей” поддержки автоматизации, впервые введенной в Delphi 3, она была оставлена только ради обратной совместимости, а потому здесь мы ее рассматривать не будем.

Варианты

Варианты представляют собой старейшую форму поддержки технологии COM в Delphi, впервые появившуюся в Delphi 2. Как вы уже знаете, в действительности тип `Variant` — это просто большая запись, которая используется для передачи некоторых данных одного из многочисленных допустимых типов. Если вас интересует, как выглядит эта запись, то она определена в модуле `System` как тип `TVarData`:

```
type
  PVarData = ^TVarData;
  TVarData = record
    VType: Word;
    Reserved1, Reserved2, Reserved3: Word;
    case Integer of
      varSmallint: (VSmallint: Smallint);
      varInteger:  (VInteger: Integer);
      varSingle:   (VSingle: Single);
      varDouble:   (VDouble: Double);
      varCurrency: (VCurrency: Currency);
      varDate:     (VDate: Double);
      varOleStr:   (VOleStr: PWideChar);
      varDispatch: (VDispatch: Pointer);
```

```
    varError:    (VError: LongWord);
    varBoolean:  (VBoolean: WordBool);
    varUnknown:  (VUnknown: Pointer);
    varByte:     (VByte: Byte);
    varString:   (VString: Pointer);
    varAny:      (VAny: Pointer);
    varArray:    (VArray: PVarArray);
    varByRef:    (VPointer: Pointer);
end;
```

Значение поля VType этой записи означает тип данных, содержащийся в типе Variant, и это может быть любой из кодов вариантного типа, приведенных в начале модуля System и перечисленных в разделе variant этой записи (внутри оператора case). Единственным различием между типами Variant и OleVariant является то, что тип Variant поддерживает все коды типов, в то время как тип OleVariant — только типы, совместимые с автоматизацией. Например, вполне приемлемо присвоить Pascal-тип string (varString) типу Variant, но для присвоения той же строки типу OleVariant придется преобразовать ее в совместимый с автоматизацией тип WideString (varOleStr).

При работе с типами Variant и OleVariant компилятор в действительности обрабатывает и передает экземпляры записей типа TVarData. И в самом деле, всегда можно безопасно привести тип Variant или OleVariant к типу TVarData, если по некоторым причинам необходимо манипулировать внутренними составляющими этой записи (хотя делать это не рекомендуется).

В суровом мире COM-программирования на языках C и C++ (за рамками классов) варианты представлены структурой VARIANT, определенной в файле oaidl.h. При работе с вариантами в этой среде вам придется вручную инициализировать и управлять ими с помощью функций API VariantXXX() из библиотеки oleaut32.dll. (Речь идет о функциях VariantInit(), VariantCopy(), VariantClear() и т.д.) Это делает работу с вариантами в языках C и C++ достаточно сложной задачей.

Поскольку поддержка вариантов встроена непосредственно в язык Object Pascal, компилятор генерирует необходимые обращения к процедурам поддержки вариантов автоматически, по мере использования экземпляров данных типа Variant или OleVariant. Однако за комфорт в языке нужно расплачиваться необходимостью приобретения нового багажа знаний. Если рассмотреть таблицу импортирования "ничего не делающего" EXE-файла Delphi с помощью такого инструмента, как утилита Borland TDUMP.EXE или утилита Microsoft DUMPBIN.EXE, можно заметить несколько подозрительных импортированных объектов из библиотеки oleaut32.dll: VariantChangeTypeEx(), VariantCopyInd() и VariantClear(). Это значит, что даже в приложении, в котором не используются явно типы Variant или OleVariant, EXE-файл Delphi все равно зависит от функций COM API из библиотеки oleaut32.dll.

Массивы вариантов

Массивы вариантов в Delphi разработаны для инкапсуляции массивов COM типа SafeArray, которые представляют собой тип записи, используемой для инкапсуляции массива данных в автоматизации. Они названы *безопасными* (safe) благодаря своей самоописательности. Помимо данных массива, эта запись содержит информацию, связанную числом измерений, размером элемента и числом элементов в массиве. Variant-массивы создаются и управляются в Delphi с помощью функций и процедур VarArrayXXX(), определенных в модуле System и описанных в интерактивной справочной системе. Эти функции и процедуры явля-

ются, по сути, оболочками для функций API SafeArrayXXX(). Если переменная типа Variant содержит variant-массив, то для доступа к элементам массива используется стандартный синтаксис индексации массива. И опять-таки, при сравнении этих средств с ручным кодированием safearray-массивов в языках C и C++ можно заметить, что использованная в языке Object Pascal инкапсуляция отличается ясностью, меньшей громоздкостью и меньшей "предрасположенностью" к ошибкам.

Автоматизация с поздним связыванием

Как упоминалось выше в этой главе, типы Variant и OleVariant позволяют писать приложения-клиенты, использующие автоматизацию с поздним связыванием (позднее связывание означает, что функции вызываются во время выполнения с помощью метода Invoke интерфейса IDispatch). Все это легко можно принять за чистую монету, но вот вопрос: "Где же та магическая связь между вызовом метода сервера автоматизации из переменной типа Variant в программе и самим методом IDispatch.Invoke(), каким-то образом вызванным с правильными параметрами?" Ответ можно найти на более низком уровне, чем можно было бы ожидать.

После вызова метода объектами типа Variant или OleVariant, содержащими интерфейс IDispatch, компилятор просто генерирует обращение к вспомогательной функции _DispInvoke (объявленной в модуле System), которая передает управление по значению указателя на функцию, имя которого VarDispProc. По умолчанию указатель VarDispProc указывает на метод, который при вызове просто возвращает ошибку. Однако, если в директиву uses включить модуль ComObj, то в разделе initialization модуля ComObj указатель VarDispProc будет перенаправлен на другой метод с помощью следующего оператора:

```
VarDispProc := @VarDispInvoke;
```

VarDispInvoke — это процедура в модуле ComObj, имеющая следующее объявление:

```
procedure VarDispInvoke(Result: PVariant; const Instance: Variant;
  CallDesc: PCallDesc; Params: Pointer); cdecl;
```

Реализация этой процедуры справляется со сложностью вызова метода IDispatch.GetIDsOfNames() с целью получения диспетчерского идентификатора (DispID) на основе имени метода, корректно устанавливает требуемые параметры и выполняет обращение к методу IDispatch.Invoke(). Самое интересное здесь то, что компилятор в данном случае не обладает никакими внутренними "знаниями" об интерфейсе IDispatch или о том, как делается вызов метода Invoke(). Он просто передает управление в соответствии со значением указателя на функцию. Также интересно то, что благодаря подобной архитектуре, вполне возможно перенаправить этот указатель функции на свою собственную процедуру, если необходимо самостоятельно обрабатывать все вызовы автоматизации через типы Variant и OleVariant. При этом потребовалось бы позаботиться лишь о том, чтобы объявление вашей функции совпадало с объявлением процедуры VarDispInvoke. Безусловно, это — задача не для новичков, но полезно знать, что при необходимости вполне можно воспользоваться таким гибким подходом.

Тип данных WideString

Тип данных WideString был введен в Delphi 3 с двойной целью: для поддержки двухбайтовых символов кодировки Unicode и для поддержки символьных строк, совместимых BSTR-строками COM. Тип WideString отличается от близкого типа AnsiString по нескольким ключевым параметрам.

- Все символы, входящие в строку типа WideString, имеют размер, равный двум байтам.
- Для типов WideString память всегда выделяется с помощью функции SysAllocStringLen(), и поэтому они полностью совместимы с BSTR-строками.
- Для типов WideString никогда не ведется подсчет ссылок, и поэтому эти типы всегда копируются при присваивании.

Подобно вариантам, работа с BSTR-строками при использовании стандартных функций API отличается громоздкостью, поэтому встроенная в язык Object Pascal поддержка типа WideString вносит заметное упрощение. Однако поскольку эти строки требуют двойного объема памяти и не учитывают ссылок, они менее эффективны по сравнению со строками типа AnsiString. С учетом вышесказанного, прежде чем их использовать, хорошо взвесьте все за и против.

Подобно Pascal-типу Variant, наличие типа WideString приводит к автоматическому импортированию некоторых функций из библиотеки oleaut32.dll даже в том случае, если программист сам не использует этот тип. При изучении таблицы импортирования "ничего не делающего" приложения Delphi оказывается, что функции SysStringLen(), SysFreeString(), SysReAllocStringLen() и SysAllocStringLen() извлечены библиотекой Delphi RTL для обеспечения поддержки типа WideString.

Интерфейсы

Возможно, самым важным элементом реализации модели COM в языке Object Pascal является встроенная поддержка интерфейсов. Ирония "судьбы" состоит в том, что если менее масштабные средства (имеются в виду типы Variant и WideString) реализуются с прямым использованием функций COM API, то при реализации в языке Object Pascal интерфейсов функции COM API вообще не нужны. То есть, в языке Object Pascal имеется абсолютно самодостаточная реализация интерфейсов, которая полностью соответствует спецификации COM, но при этом не использует ни одной функции COM API.

В плане соответствия спецификации COM все интерфейсы в Delphi неявно выведены из интерфейса IUnknown. Как вам, должно быть, известно, интерфейс IUnknown обеспечивает средства определения типа и поддержку учета ссылок, что является основой основ модели COM. Это значит, что знание особенностей интерфейса IUnknown встроено непосредственно в компилятор, а сам интерфейс IUnknown определен в модуле System. Сделав интерфейс IUnknown полноправным членом языка программирования, среда Delphi приобрела способность организовывать автоматический подсчет ссылок, обязав компилятор генерировать обращения к функциям IUnknown.AddRef() и IUnknown.Release() в соответствующие моменты времени. Кроме того, оператор as может быть использован в качестве ускоренного варианта определения типа интерфейса, обычно реализуемого с помощью метода QueryInterface(). Однако встроенная поддержка интерфейса IUnknown оказывается просто незначительным фрагментом, если рассмотреть весь объем низкоуровневой поддержки, обеспечиваемой языком и компилятором для интерфейсов в целом.

На рис. 23.17 показана упрощенная схема внутренней поддержки интерфейсов со стороны классов. В действительности объект Delphi — это ссылка, которая указывает на физический экземпляр. Первых четыре байта экземпляра объекта представляют собой указатель на таблицу виртуальных методов объекта (virtual method table — VMT). При положительном смещении от значения VMT находятся все виртуальные методы объекта. С отрицательным смещением размещаются те указатели на методы и данные, которые важны для внутреннего функционирования объекта. В частности, на уровне смещения -72 от значения VMT содер-

жится указатель на таблицу интерфейсов объекта. Таблица интерфейсов представляет собой список записей типа `PInterfaceEntry` (определенных в модуле `System`), которые, по сути, содержат идентификаторы интерфейса IID и информацию о том, где найти `vtable`-указатель для данного идентификатора интерфейса IID.

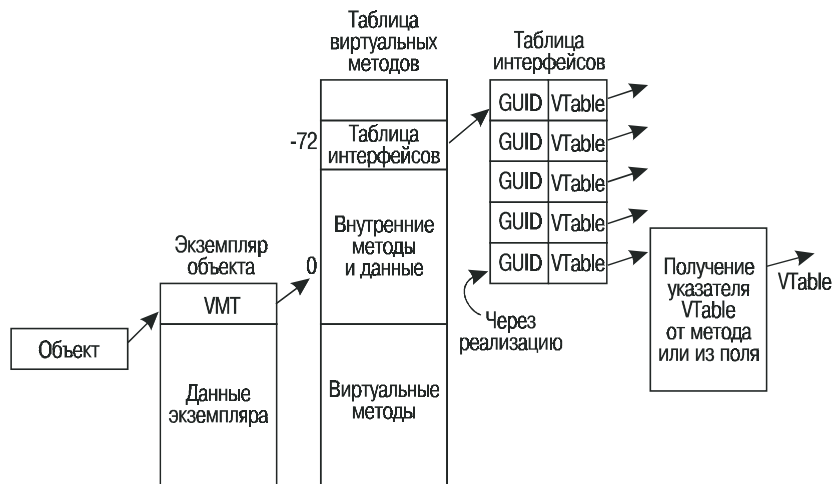


Рис 23.17. Поддержка интерфейсов внутренними средствами языка Object Pascal

Рассмотрев показанную на рис. 23.17 схему, вы сможете понять, как отдельные детали увязываются друг с другом, и затем сделать вывод, что реализация интерфейсов в Delphi направлена на попадание в определенную цель. Например, метод `QueryInterface()` обычно реализуется в объектах Object Pascal путем вызова метода `TObject.GetInterface()`. Метод `GetInterface()` просматривает таблицу интерфейсов в надежде найти нужный идентификатор интерфейса IID и возвращает указатель виртуальной таблицы (`vtable`-указатель) для этого интерфейса. Теперь понятно, почему новые типы интерфейсов должны быть определены с помощью уникального идентификатора GUID — ведь в противном случае метод `GetInterface()` не сможет находить их при просмотре таблицы интерфейсов, и, следовательно, получение интерфейса с помощью метода `QueryInterface()` будет невозможным. Приведение типов интерфейсов с помощью оператора `as` просто генерирует обращение к методу `QueryInterface()`, поэтому и здесь применяются те же самые правила.

Последняя запись в таблице интерфейсов (см. рис. 23.17) служит иллюстрацией внутренней реализации интерфейса на основе использования директивы `implements`. Вместо предоставления прямого указателя для виртуальной таблицы (`vtable`-указателя), запись таблицы интерфейсов обеспечивает адрес небольшой генерируемой компилятором функции, которая получает виртуальную таблицу интерфейса из свойства, для которого была использована директива `implements`.

Диспинтерфейсы

Диспинтерфейс обеспечивает инкапсуляцию недвойственного интерфейса `Idispatch`, т.е. интерфейса `IDispatch`, в котором методы могут быть вызваны только через метод `Invoke()`, но не через виртуальную таблицу. В этом отношении диспинтерфейс аналогичен автоматизации с вариантами. Однако диспинтерфейсы чуть более эффективны, чем варианты, поскольку объявления `dispinterface` содержат диспетчерский идентификатор DispID для каждого под-

держиваемого свойства или метода. Это означает, что метод IDispatch.Invoke() можно вызвать напрямую, без первоначального вызова метода IDispatch.GetIDsOfNames(), как это требуется в случае с вариантами. В остальном же механизм работы диспинтерфейсов аналогичен механизму работы вариантов: при вызове метода через диспинтерфейс компилятор генерирует обращение к функции _IntfDispCall из модуля System. Этот метод передает управление указателю DispCallByIDProc, который по умолчанию возвращает только ошибку. Однако при включении в список uses модуля ComObj указатель DispCallByIDProc инициализируется адресом процедуры DispCallByID(), которая объявлена в модуле ComObj следующим образом:

```
procedure DispCallByID(Result: Pointer; const Dispatch: IDispatch;
  DispDesc: PDispDesc; Params: Pointer); cdecl;
```

Сервер транзакций Microsoft (MTS)

Сообщество разработчиков COM бурно отреагировало в свое время на появление такой технологии, как Microsoft Transaction Server (MTS), и небезосновательно. MTS представляет новую парадигму для разработчиков COM. Разработчики COM долгое время наслаждались преимуществами независимых от языка интерфейсов, прозрачностью расположения объектов и автоматической их активизацией и дезактивацией. Однако благодаря технологии MTS разработчики COM могут теперь воспользоваться преимуществами мощного сервиса времени выполнения, связанного с управлением временем жизни объектов, защитой, буферизацией ресурсов и управлением транзакциями. Хотя технология MTS и несет с собой много полезных средств, она также требует внесения определенных изменений в системное проектирование, в некоторых случаях противоречащее идеям, с которыми модель COM вошла в наше сознание за годы своего существования. В этом разделе мы рассмотрим технологию MTS, а в следующем — поговорим более предметно об MTS и Delphi, реализации MTS в Delphi и поддержке этой технологии в IDE, а также рассмотрим некоторые примеры компонентов и приложений MTS.

Прежде чем погрузиться с головой в технические подробности, полезно будет узнать, что обработка транзакций — это только малая часть общей картины MTS, и то, что слово *транзакция* (transaction) попало в название этой технологии, собственно, еще ни о чем не говорит и даже где-то “путает карты”. Такое неудачное название можно сравнить с тем, как если бы мы назвали новейший телевизор инструментом для просмотра мыльных опер. Безусловно, он является таковым, но ведь это не единственная его функция. Создатели этой технологии признают свою неудачу с выбором названия, которое, к счастью, не должно уже больше раздражать нас, поскольку предполагается, что технология MTS войдет в операционную систему как часть вводимых в COM усовершенствований, известных уже под общим названием COM+.

Зачем нам MTS

Магическим словом системного проектирования в наши дни стало слово *масштабируемость* (scalability). В связи с гиперростом сетей Internet и intranet (или корпоративных сетей), консолидацией корпоративных данных в централизованных хранилищах и необходимостью для всех и каждого получать доступ к этим данным, абсолютно критичным для системы является ее способность к расширению с целью удовлетворения потребностей постоянно возрастающего числа параллельно работающих пользователей. Это определенно является пробле-

мой, особенно, если учесть такие совершенно непростительные ограничения, с которыми мы должны иметь дело, как конечное количество соединений с базой данных, пропускная способность сети, нагрузка на сервер и т.д. В старые добрые времена начала 90-х годов системы, построенные по принципу клиент/сервер, считались последним криком моды и рассматривались как способ написания масштабируемых приложений. Однако по мере того, как базы данных обрастали триггерами и хранимыми процедурами, а приложения-клиенты без конца усложнялись, накапливая там и сям всевозможные программные "примочки", предназначенные для реализации бизнес-правил, очень скоро стало ясно, что такие системы имеют ограниченные возможности расширения с целью подключения все большего числа пользователей. Популярным решением этой проблемы стала многоярусная (многоуровневая) архитектура. Разместив логику приложения и совместно используемые соединения с базами данных в среднем ярусе, оказалось возможным упростить логику работы баз данных и клиентов, а также оптимизировать использование ресурсов, что позволило создать системы с более широкой пропускной способностью.

Нужно отметить, что добавленная инфраструктура, введенная в многоярусную среду, имеет тенденцию к увеличению времени ожидания одновременно с ростом пропускной способности. Другими словами, ради достижения масштабируемости часто приходится жертвовать производительностью системы.

Компания Microsoft обеспечила COM-разработчиков средствами построения приложений, распределенных на нескольких компьютерах, — благодаря разработанной несколько лет назад распределенной модели COM (DCOM). Модель DCOM можно назвать первым шагом, сделанным в правильном направлении. Она предоставила средства, с помощью которых отдельные COM-среды могли общаться друг с другом на расстоянии, но при этом не было сделано никаких существенных движений в направлении решения реальных проблем, с которыми то и дело сталкивались разработчики распределенных приложений. Такие вопросы, как оптимизация времени жизни объектов, управление потоками, гибкие средства защиты и поддержка транзакций, по-прежнему лежали тяжким грузом на плечах отдельных разработчиков. И вот появилась технология MTS.

Что такое MTS

MTS — это COM-ориентированная модель программирования и коллекция динамических служб, предназначенных для разработки масштабируемых или ориентированных на транзакции приложений на платформе COM. Часть модели MTS, относящаяся к программированию, не слишком отличается от уже известной COM-разработчикам. Есть несколько новых деталей, о которых вы скоро узнаете, но в большинстве случаев любой внутренний (in-process) COM-объект (библиотека DLL), имеющий библиотеку типов, вполне может использоваться как объект MTS. Однако не рекомендуется в среде MTS запускать в работу COM-компоненты, не учитывающие специфику MTS. Динамические службы MTS выполняют роль опекунов над COM-компонентами. MTS может создавать их, управлять временем их жизни, обеспечивать их безопасность и пр. Это значит, что вместо работы в контексте вашего приложения объекты MTS COM работают в контексте динамической среды выполнения модели MTS. Все это добавляет целый букет новых возможностей и позволяет достичь определенных преимуществ при внесении минимальных изменений (или вообще без оных) в программный код приложения-клиента или объекта COM.

Интересно отметить, что поскольку объекты MTS не работают непосредственно внутри контекста клиента, как другие объекты COM, клиенты в действительности никогда не получают указатели интерфейсов на собственно экземпляр объекта. Вместо этого MTS вставляет своего представителя — *заместителя* (proxy) — между клиентом и объектом MTS, причем таким образом, что с точки зрения клиента этот заместитель идентичен самому объекту. Но

поскольку MTS имеет полный контроль над своим представителем, он может управлять доступом к методам интерфейса объекта, осуществляя при этом управление временем жизни объекта или его безопасностью (об этом речь впереди).

Что лучше: знать или не знать о своем состоянии?

Главной темой разговоров среди тех, кто как-то имеет дело с технологией MTS, является сравнение объектов, которые поддерживают и не поддерживают информацию о своем состоянии. Хотя модель COM сама по себе ничего не прибавляет к состоянию объекта, тем не менее на практике большинство традиционных COM-объектов "помнит" о себе. Иначе говоря, они постоянно поддерживают информацию о своем состоянии начиная с момента их создания, в процессе всего периода их использования и вплоть до момента их разрушения. Проблема работы с такими объектами заключается в том, что они определенно снижают возможности масштабирования, поскольку информацию о состоянии приходится поддерживать для *каждого* объекта, к которому получают доступ *каждый* клиент. *Объект без состояния* (stateless object) — это объект, который обычно не поддерживает информацию о своем состоянии между вызовами методов. Предпочтительнее работать именно с такими объектами, поскольку они позволяют MTS выполнять некоторые виды оптимизации. Если объект не поддерживает своего состояния между вызовами методов, MTS теоретически может заставить объект исчезнуть между вызовами, не причинив ему при этом никакого вреда. Более того, поскольку клиент поддерживает указатели только на внутреннего заместителя MTS для данного объекта, MTS мог бы при этом не интересоваться мнением клиента. И это не просто теория — именно так MTS и работает. MTS будет разрушать экземпляры объектов между вызовами методов, чтобы освободить ресурсы, связанные с данным объектом. Когда клиент сделает еще один вызов к тому же объекту, представитель MTS перехватит его и автоматически создаст новый экземпляр объекта. Это способствует масштабированию системы в направлении увеличения числа ее пользователей, поскольку в каждый момент времени, вероятно, будет существовать сравнительно немного активных экземпляров каждого класса.

Написание интерфейсов, поддерживающих объекты без запоминания состоянии, вероятно, потребует отказа от обычного способа мышления при проектировании интерфейсов. Рассмотрим, например, следующий классический интерфейс в стиле COM:

```
ICheckbook = interface
['{2CCF0409-EE29-11D2-AF31-0000861EF0BB}']
  procedure SetAccount(AccountNum: WideString); safecall;
  procedure AddActivity(Amount: Integer); safecall;
end;
```

Нетрудно предположить, что этот интерфейс можно использовать следующим образом:

```
var
  CB: ICheckbook;
begin
  CB := SomehowGetInstance;    // Каким-то образом создаем экземпляр
  CB.SetAccount('12345ABCDE'); // Открываем чековый счет
  CB.AddActivity(-100);        // Фиксируем расход в размере $100
  ...
end;
```

Проблема, связанная с этим стилем, состоит в том, что данный объект хранит информацию о своем состоянии между вызовами его методов, поскольку информация, связанная с номером счета, должна поддерживаться и после открытия чекового счета. Но при использо-

вании этого интерфейса в MTS можно предложить другой (улучшенный) подход, заключающийся в передаче всей необходимой информации методу AddActivity(), чтобы этот объект не нуждался в сохранении информации о своем текущем состоянии:

```
procedure AddActivity(AccountNum: WideString; Amount: Integer); safecall;
```

Конкретное состояние некоторого активного объекта принято называть *контекстом*. MTS поддерживает контекст для каждого активного объекта, в котором отслеживает такую информацию, как данные о безопасности и транзакциях. Каждый объект может в любое время вызвать метод GetObjectContext(), чтобы получить указатель интерфейса IObjectContext для получения своего контекста. Интерфейс IObjectContext определен в модуле Mtx следующим образом:

```
IObjectContext = interface(IUnknown)
    ['{51372AE0-CAE7-11CF-BE81-00AA00A2FA25}']
    function CreateInstance(const cid, rid: TGUID; out pv): HResult; stdcall;
    procedure SetComplete; safecall;
    procedure SetAbort; safecall;
    procedure EnableCommit; safecall;
    procedure DisableCommit; safecall;
    function IsInTransaction: Bool; stdcall;
    function IsSecurityEnabled: Bool; stdcall;
    function IsCallerInRole(const bstrRole: WideString): Bool; safecall;
  end;
```

Двумя самыми важными методами в этом интерфейсе являются методы SetComplete() и SetAbort(). Когда вызывается любой из этих методов, объект сообщает MTS, что больше не нужно поддерживать никакого состояния. Поэтому MTS разрушит объект (без ведома клиента, конечно), освободив таким образом ресурсы для других экземпляров. Если объект участвует в транзакции, то при выполнении метода SetComplete() либо SetAbort() также достигается результат либо фиксации, либо отката транзакции соответственно.

Управление временем жизни объектов

С первых шагов на пути освоения COM-технологии нам внушалось, что указатели интерфейсов следует хранить столько, сколько это необходимо, и освобождать их сразу после того, как они станут ненужными. В традиционной модели COM это имеет большое значение, поскольку не стоит систему занимать поддержкой неиспользуемых ресурсов. Но, так как MTS будет автоматически освобождать не поддерживающие своего состояния объекты после вызова ими метода SetComplete() или SetAbort(), нельзя говорить и о расходах, связанных с неограниченным хранением ссылки на такой объект. Более того, поскольку клиент никогда не узнает, что экземпляр объекта уже удален (неявно), клиентскую программу даже не придется переписывать, чтобы воспользоваться этими преимуществами.

Пакеты

Термин *пакет* (package) уже достаточно “перегружен”: пакеты Delphi, пакеты C++Builder и пакеты Oracle — это все примеры избыточного использования этого слова. Модель MTS также не обошлась без понятия пакета, значение которого отличается от значения его предшественников. Пакет MTS несет в себе больше логическую, чем физическую нагрузку, поскольку он представляет определенную программистом коллекцию MTS-объектов с такими

атрибутами, как активизация, безопасность и транзакция. С физической точки зрения пакет — это файл, который содержит ссылки на библиотеки DLL COM-сервера и объекты MTS в пределах тех серверов, которые составляют пакет. Файл пакета также содержит информацию об атрибутах внутренних объектов MTS.

MTS будет выполнять все компоненты внутри пакета в одном и том же процессе. Это позволяет настроить проверенные и работающие без сбоев пакеты так, чтобы их функционирование было изолировано от потенциальных проблем, которые могут быть вызваны ошибками в других пакетах. Интересно также отметить, что физическое расположение компонентов не имеет никакого отношения к праву на включение в пакет: один COM-сервер может содержать несколько COM-объектов, причем каждый из них может находиться в отдельном пакете.

Создание и управление пакетами осуществляется либо с помощью команды Run⇨Install MTS Objects в среде разработки Delphi, либо с помощью приложения Transaction Server Explorer, которое устанавливается вместе с поддержкой MTS (рис. 23.18).

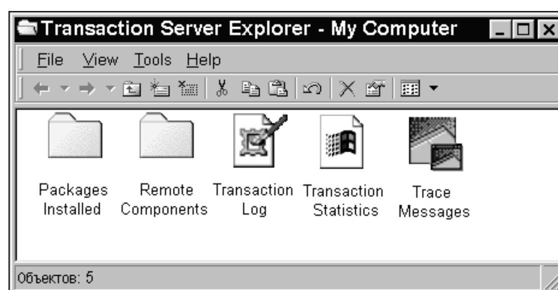


Рис. 23.18. Приложение Windows 98 `Transaction Server Explorer`

Безопасность

В технологии MTS предусмотрена система безопасности "ролевого" типа, которая гораздо гибче, чем стандартная система безопасности Windows NT, обычно используемая в распределенной модели DCOM. *Роль* представляет собой категорию пользователя (например, в банковской системе типичными ролями могут быть кассир, контролер и директор). Модель MTS позволяет задавать уровень, в пределах которого любая конкретная роль имеет право определять действия объекта, причем для каждого интерфейса в отдельности. Например, можно задать, что роль директора имеет доступ к интерфейсу `ICreateHomeLoan`, а роль кассира — нет. Если нужно получить бо́льшую степень детальности, чем доступ к целым интерфейсам, можно определить роль пользователя в текущем контексте путем вызова метода `IsCallerInRole()` интерфейса `IObjectContext`. С помощью таких мер, к примеру, можно было бы привести в исполнение бизнес-правило, согласно которому кассиры могут санкционировать закрытие только "нормальных" счетов, а контролеры — счетов, на которых более $100 000. Настройка ролей безопасности проводится в среде приложения Transaction Server Explorer.

Ах, да, еще и транзакции!

Конечно же, в технологии MTS также уделено внимание и транзакциям. Вы могли бы подумать: "Большое дело, мой сервер баз данных уже поддерживает транзакции. Для чего нужно, чтобы и создаваемые мной компоненты тоже их поддерживали?" Вполне законный вопрос, на который, к счастью, есть хороший ответ. Благодаря поддержке транзакций в MTS можно выполнять транзакции, охватывающие несколько баз данных или даже совершать одно элементарное действие, состоящее из набора некоторых операций, не имеющих отноше-

ния к базам данных. Для поддержки транзакций в объектах MTS необходимо установить корректный признак транзакций в библиотеке типов компонентного класса создаваемого объекта либо во время разработки (что и делает мастер Delphi MTS Wizard), либо после установки с помощью приложения Transaction Server Explorer.

Когда следует использовать транзакции в объектах? Ответ простой: когда у вас есть процесс, состоящий из нескольких шагов, которые необходимо объединить в одну неделимую транзакцию. В этом случае целый процесс может быть либо выполнен до конца, либо отменен с откатом состояния к началу, но логика или данные никогда не будут брошены в некорректном или неопределенном состоянии, возникшем из-за незавершенности процесса. Например, если при написании программ для банка вы хотите предусмотреть случай, когда клиент оплачивает чек, то обработка такой ситуации, вероятно, будет включать несколько шагов: снятие со счета суммы, указанной в чеке, снятие со счета платы за услуги по оплате чека и отправка письма клиенту.

Чтобы корректным образом осуществить оплату чека, необходимо выполнить каждое из перечисленных действий. Следовательно, объединив их в одну транзакцию, можно гарантировать, что все они будут выполнены (если не возникнет никакой ошибки), а при возникновении любой ошибки все они будут отменены, а система будет приведена к состоянию, какое она имела до начала транзакции.

Ресурсы

Имея дело с постоянно создаваемыми и разрушаемыми объектами и повсеместно происходящими транзакциями, модель MTS должна обладать средствами распределения между несколькими объектами определенных конечных или дорогих ресурсов (например, соединений баз данных). В MTS эта задача решается с помощью диспетчеров и распределителей ресурсов. *Диспетчер ресурсов* (resource manager) — это служба, управляющая долговременными данными, такими как остатки на счетах или инвентарные ведомости. Диспетчер ресурсов Microsoft создан в среде MS SQL Server. *Распределитель ресурсов* (resource dispenser) управляет недолговременными ресурсами, такими как соединения баз данных. Компания Microsoft предоставляет распределитель ресурсов для соединений баз данных ODBC, а компания Borland — для соединений баз данных BDE.

Когда транзакция использует некоторый тип ресурсов, эти ресурсы становятся ее частью. Это позволяет всем изменениям, вносимым в ресурсы во время транзакции, принимать участие в операциях фиксации или отката.

Технология MTS в Delphi

Выяснив все “что” и “зачем”, пора получить ответ на вопрос “как”. В частности, остановимся на поддержке MTS средствами Delphi и на возможных способах построения решений MTS в Delphi. При этом вы должны учитывать, что поддержка MTS встроена только в версию Delphi Enterprise Edition. Хотя технически возможно создать компоненты MTS, используя средства, доступные в версиях Standard Edition и Professional Edition, вряд ли стоит это делать (если вы цените свое время). Поэтому в этом разделе мы рассмотрим особенности версии Delphi Enterprise Edition.

Мастера MTS

В Delphi предусмотрено два мастера для построения компонентов MTS, которые расположены во вкладке Multitier диалогового окна New Items: мастер модулей удаленных данных MTS (MTS Remote Data Module Wizard) и мастер объектов MTS (MTS Object Wizard). Мастер MTS Remote Data Module Wizard позволяет строить серверы MIDAS, которые дей-

ствуют в среде MTS. Мастер MTS Object Wizard служит в качестве стартовой точки при создании объектов MTS, и именно на нем мы сосредоточим ваше внимание. После вызова мастера откроется диалоговое окно, показанное на рис. 23.19.

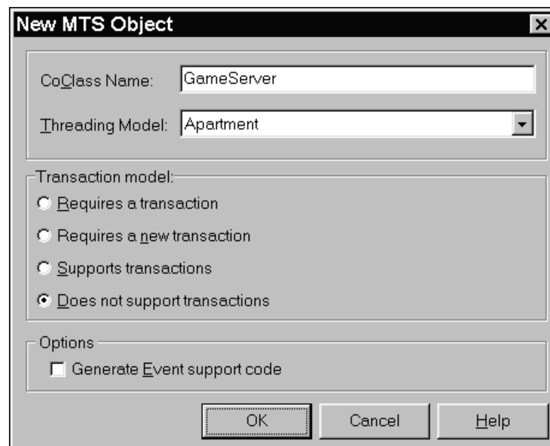


Рис. 23.19. Диалоговое окно мастера New MTS Object

Диалоговое окно, показанное на рис. 23.19, подобно диалоговому окну мастера объектов автоматизации (Automation Object Wizard), рассмотренному выше в этой главе. Основное отличие мастера объектов MTS — в предоставлении переключателя для выбора модели транзакции, поддерживаемой вашим компонентом MTS. Ниже перечислены модели транзакций:

- *Requires a transaction* (Требует транзакцию). Компонент будет создаваться в контексте транзакции. Он будет наследовать транзакцию своего создателя, если таковая существует; в противном случае он создаст новую транзакцию.
- *Requires a new transaction* (Требует новую транзакцию). Для компонента будет всегда создаваться новая транзакция.
- *Supports transactions* (Поддерживает транзакции). Компонент наследует транзакцию своего создателя, если таковая существует; в противном случае он будет выполняться без транзакции.
- *Does not support transactions* (Не поддерживает транзакции). Компонент никогда не будет создан внутри транзакции.

Информация о модели транзакции хранится в виде атрибута вместе компонентным классом в библиотеке типов.

По щелчку на кнопке OK, закрывающем это диалоговое окно, мастер генерирует пустое определение для класса, который происходит от класса `TMtsAutoObject`, и переносит вас в окно редактора библиотеки типов (Type Library Editor), чтобы можно было определить компоненты MTS путем добавления свойств, методов, интерфейсов и пр. Этот вид работы должен быть вам знаком — ввиду идентичности (на данном этапе) разработке объектов автоматизации в Delphi. Интересно отметить, что, хотя объекты MTS, созданные мастером Delphi, являются объектами автоматизации (т.е. объектами COM, которые реализуют интерфейс `IDispatch`), MTS формально не требует этого. Однако поскольку COM "знает", как выполняется маршалинг для интерфейсов типа `IDispatch`, имеющих библиотеки типов, использование объекта этого типа в MTS позволяет программистам больше уделять внимание функциональности своих компонентов и меньше — тому, как они интегрируются с MTS. Следует

также знать, что компоненты MTS всегда должны размещаться во внутренних серверах COM (библиотеках DLL). MTS не поддерживает компонентов, выполняемых во внешних серверах (EXE-файлах).

Структура MTS

Упомянутый выше класс TMtsAutoObject, который является базовым классом для всех объектов MTS, созданных мастером Delphi, определен в модуле MtsObj. Относительное простое определение класса TMtsAutoObject имеет следующий вид:

```
type
  TMtsAutoObject = class(TAutoObject, IObjectControl)
  private
    FObjectContext: IObjectContext;
  protected
    { IObjectControl }
    procedure Activate; safecall;
    procedure Deactivate; stdcall;
    function CanBePooled: Bool; stdcall;

    procedure OnActivate; virtual;
    procedure OnDeactivate; virtual;
    property ObjectContext: IObjectContext read FObjectContext;
  public
    procedure SetComplete;
    procedure SetAbort;
    procedure EnableCommit;
    procedure DisableCommit;
    function IsInTransaction: Bool;
    function IsSecurityEnabled: Bool;
    function IsCallerInRole(const Role: WideString): Bool;
  end;
```

Класс TMtsAutoObject — это, по сути, класс TAutoObject, расширенный в двух важных направлениях.

- Класс TMtsAutoObject реализует интерфейс IObjectControl, который управляет инициализацией и очисткой компонентов MTS. Методы этого интерфейса приведены в табл. 23.3.

Таблица 23.3. Методы интерфейса IObjectControl

Имя метода	Описание
Activate	Позволяет объекту при активизации выполнять контекстно-зависимую инициализацию. Этот метод будет вызван средствами MTS до вызова любого пользовательского метода в компоненте MTS
Deactivate	Позволяет при дезактивации выполнять контекстно-зависимую очистку
CanBePooled	Этот метод пока не используется, поскольку в технологии MTS еще не предусмотрена поддержка буферизации объектов

Класс `TMtsAutoObject` предоставляет виртуальные методы `OnActivate()` и `OnDeactivate()`, которые вызываются из закрытых методов `Activate()` и `Deactivate()`. Для создания специальной логики контекстно-зависимой активизации или деактивизации достаточно лишь переопределить эти методы.

- Класс `TMtsAutoObject` также поддерживает указатель на интерфейс MTS `IObjectContext` в форме свойства `ObjectContext`. Как указывалось выше, интерфейс `IObjectContext` (предоставленный средствами MTS) обеспечивает компонент возможностью управлять своим текущим контекстом. В качестве подспорья для пользователей этого класса, класс `TMtsAutoObject` также "выводит на поверхность" все методы интерфейса `IObjectContext`, которые реализуются с помощью простого обращения к объекту `ObjectContext`. Например, реализация метода `TMtsAutoObject.SetComplete()` состоит в простой проверке переменной `FObjectContext` на неравенство значению `nil` с последующим вызовом метода `FObjectContext.SetComplete()`. Полный список методов интерфейса `IObjectContext` с кратким их описанием приведен в табл. 23.4.

Таблица 23.4. Методы интерфейса IObjectContext

Имя метода	Описание
`CreateInstance`	Создает экземпляр еще одного объекта MTS. Этот метод можно представлять себе как выполняющий для объектов MTS ту же задачу, которую для обычных COM-объектов выполняет метод `IClassFactory.CreateInstance`
`SetComplete`	Уведомляет MTS о том, что данный компонент завершил свою работу и больше не должен поддерживать свое внутреннее состояние. Если компонент имеет отношение к транзакциям, то сообщается, что текущие транзакции могут быть зафиксированы. После возврата из метода, вызывающего эту функцию, средства MTS могут деактивизировать объект, освободив ресурсы с целью обеспечения масштабируемости
`SetAbort`	Аналогично методу `SetComplete()`, этот метод уведомляет средства MTS о том, что компонент завершил работу и больше не должен поддерживать свое состояние. Однако вызов этого метода также означает, что данный компонент находится в ошибочном или неопределенном состоянии и любые незаконченные транзакции должны быть прерваны
`EnableCommit`	Означает, что компонент находится в "рабочем" состоянии, при котором транзакции могут быть успешно зафиксированы, когда компонент вызовет метод `SetComplete`. Это стандартное состояние компонента
`DisableCommit`	Означает, что компонент утратил целостное состояние, и, прежде чем объект будет готов к завершению транзакций, ему понадобятся вызовы других методов
`IsInTransaction`	Позволяет компоненту определить, выполняется ли он в контексте транзакции
`IsSecurityEnabled`	Позволяет компоненту определить, активизирована ли поддержка безопасности средствами MTS. Этот метод всегда возвращает значение `True`, если только компонент не выполняется в пространстве процесса клиента
`IsCallerInRole`	Предоставляет средства, с помощью которых компонент может определить, обладает ли пользователь, рассматриваемый компонентом как клиент, конкретной ролью в MTS. Этот метод — основа основ простой в применении системы безопасности MTS, основанной на распределении ролей. (Подробнее о ролях — ниже в этой главе)

Модуль `Mtx` содержит ядро средств поддержки технологии MTS. Он представляет собой Pascal-трасляцию заголовочного файла `mtx.h` и содержит типы (такие как `IObjectControl` и `IObjectContext`) и функции, составляющие программный интерфейс MTS API.

Игра в крестики-нолики: пример приложения

Достаточно теории. Пора написать несколько программных строк и понять, наконец, как вся эта махина MTS работает на практике. MTS поставляется с приложением-примером Tic-tac-toe, которое нам кажется несколько недоработанным, что и вдохновило авторов на реализацию этой классической игры в Delphi. Для начала воспользуемся мастером MTS Object Wizard и создадим новый объект с именем `GameServer`. Используя редактор библиотеки типов, добавим к стандартному интерфейсу для этого объекта (`IGameServer`) три метода: `NewGame()`, `ComputerMove()` и `PlayerMove()`. Кроме того, добавим два новых перечисления `SkillLevels` и `GameResults`, которые будут использоваться этими методами. На рис. 23.20 показаны все эти элементы в окне редактора библиотеки типов.

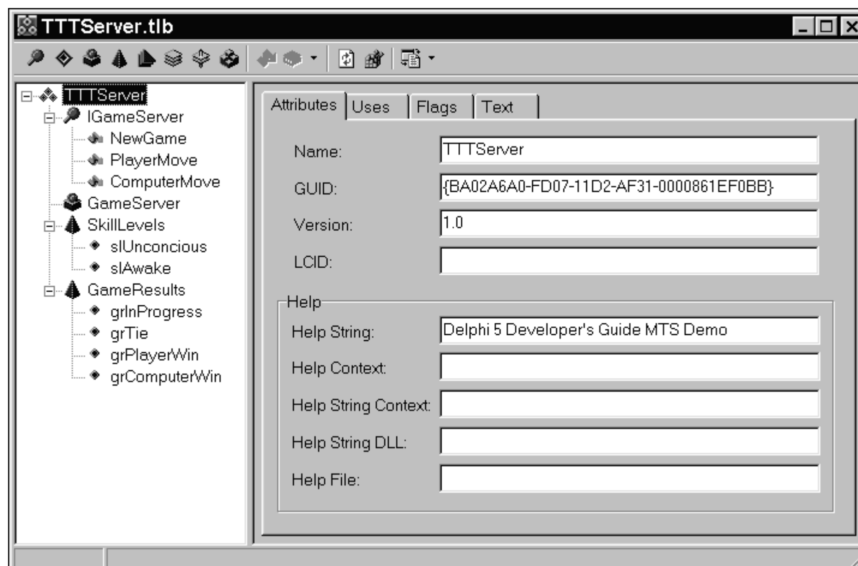


Рис. 23.20. Сервер `tic-tac-toe` в окне редактора библиотеки типов

Логика всех трех методов упомянутого интерфейса достаточно проста, и эти методы реализуют все требования для поддержки игры человека с компьютерным вариантом крестиков-ноликов (tic-tac-toe). Метод `NewGame()` инициализирует новую игру. Метод `ComputerMove()` анализирует возможные ходы и делает ход за компьютер. Метод `PlayerMove()` позволяет клиенту сообщить компьютеру о выбранном ходе. Выше в этой главе мы упоминали, что разработка компонентов MTS требует другого подхода, по сравнению с разработкой стандартных COM-компонентов. Данный компонент предоставляет прекрасную возможность проиллюстрировать этот факт.

Если бы это был обычный будничный COM-компонент, то к проектированию такого объекта можно было бы подойти со стороны инициализации в методе `NewGame()` некоторой структуры данных, предназначенной для поддержки состояния игры. Эта структура данных была бы, вероятно, некоторым полем экземпляра объекта, к которому другие методы могли бы получать доступ и выполнять некоторые манипуляции в течение жизни этого объекта.

Чем же такой подход плох в случае компонента MTS? Достаточно одного слова: состояние. Как упоминалось выше, чтобы воплотить в жизнь все выгоды технологии MTS, объекты не должны запоминать свое состояние. Однако выбор архитектуры компонента, предполагающей хранение данных экземпляра в промежутках между вызовами его методов, не отвечает этим требованиям. Более удачным проектом для среды MTS было бы вернуться к "дескриптору", идентифицирующему игру из метода `NewGame()`, и, используя этот дескриптор, поддерживать структуры данных для каждой отдельной игры в разделяемом ресурсе подходящего типа. Средства управления этими разделяемыми ресурсами следовало бы поддерживать вне контекста конкретного экземпляра объекта, поскольку MTS может активизировать и деактивизировать экземпляры объектов с каждым вызовом метода. Все остальные методы компонента могли бы принимать упомянутый дескриптор как параметр, что позволяло бы им получать данные о конкретной игре от совместно используемого средства управления ресурсами. При таком построении проекта объекты уже не фиксируют информацию о своем состоянии. Более того, от них уже не требуется оставаться активными между вызовами их методов, так как каждый метод представляет собой самодостаточную операцию, при выполнении которой вся необходимая информация извлекается с помощью средств управления разделяемыми данными на основе полученных параметров.

Это средство управления общими данными, о котором мы говорим пока абстрактно, в MTS называется *распределителем ресурсов* (resource dispenser). А если конкретнее, то распределителем ресурсов, используемым в MTS для поддержки разделяемых данных, является диспетчер общих свойств (Shared Property Manager). Этот диспетчер представлен интерфейсом `ISharedPropertyGroupManager` и находится на верхнем уровне иерархической системы хранения данных. Он поддерживает любое количество совместно используемых групп свойств, которые представлены интерфейсом `ISharedPropertyGroup`. В свою очередь, каждая группа разделяемых свойств может содержать любое число разделяемых свойств, представляемых интерфейсом `ISharedProperty`. Совместно используемые свойства удобны тем, что они существуют внутри MTS, вне контекста любого конкретного экземпляра объекта, и доступ к ним контролируется системой блокировок и семафоров, управляемых диспетчером Shared Property Manager.

С учетом всего вышесказанного реализация метода `NewGame()` будет выглядеть следующим образом:

```
procedure TGameServer.NewGame(out GameID: Integer);
var
  SPG: ISharedPropertyGroup;
  SProp: ISharedProperty;
  Exists: WordBool;
  GameData: OleVariant;
begin
  // Используем роль источника вызова (caller) для проверки безопасности
  CheckCallerSecurity;
  // Получаем группу разделяемых свойств для этого объекта
  SPG := GetSharedPropertyGroup;
  // Создаем или считываем разделяемое свойство NextGameID
  SProp := SPG.CreateProperty('NextGameID', Exists);
  if Exists then GameID := SProp.Value
  else GameID := 0;
  // Инкрементируем и запоминаем разделяемое свойство NextGameID
  SProp.Value := GameID + 1;
  // Создаем массив данных игры
```

```
  GameData := VarArrayCreate([1, 3, 1, 3], varByte);
  SProp := SPG.CreateProperty(Format(GameDataStr, [GameID]), Exists);
  SProp.Value := GameData;
  SetComplete;
end;
```

Сначала выполняется проверка того, что источник вызова действует в надлежащей роли, чтобы иметь право на вызов этого метода (подробности ниже). Затем используется разделяемое свойство для получения номера ID очередной игры. После этого создается вариантный массив для хранения данных игры, и эти данные сохраняются как разделяемое свойство. Наконец, выполняется вызов метода SetComplete(), чтобы уведомить MTS о возможности деактивизации этого экземпляра после возврата метода.

Тем самым мы приходим к правилу номер один разработки MTS: вызывайте метод SetComplete() или метод SetAbort() по возможности чаще. Было бы идеально вызывать метод SetComplete() или метод SetAbort() в каждом методе, чтобы средства MTS после возврата из данного метода могли восстановить ресурсы, ранее выделенные для экземпляра компонента. Из этого правила следует, что активизация и деактивизация объекта не должны быть "дорогим удовольствием", поскольку этот код, по всей вероятности, будет вызываться достаточно часто.

На примере реализации метода CheckCallerSecurity() иллюстрируется, насколько просто можно воспользоваться преимуществами ролевой системы безопасности в технологии MTS:

```
procedure TGameServer.CheckCallerSecurity;
begin
  // Просто ради шутки: позволяем играть в эту игру только тем,
  // кому отведена роль "TTT"
  if IsSecurityEnabled and not IsCallerInRole('TTT') then
    raise Exception.Create('Только в роли TTT можно играть в игру tic-tac-toe');
end;
```

Из этого кода напрашивается очевидный вопрос: "Как назначается роль TTT и каким образом можно установить, каким пользователям присвоена эта роль?" Хотя роли можно определять и программным путем, проще всего добавлять и настраивать использование ролей с помощью приложения Windows NT Transaction Server Explorer. После установки компонента (об этом чуть ниже) можно определить роли с помощью узла Roles, расположенного под каждым узлом пакета в окне этой программы. Важно отметить, что ролевая система безопасности поддерживается только компонентами, работающими в среде Windows NT. Для компонентов, работающих в среде Windows 9x, метод IsCallerInRole() всегда возвращает значение True.

Вот как выглядит реализация методов ComputerMove() и PlayerMove():

```
procedure TGameServer.ComputerMove(GameID: Integer;
  SkillLevel: SkillLevels; out X, Y: Integer; out GameRez: GameResults);
var
  Exists: WordBool;
  PropVal: OleVariant;
  GameData: PGameData;
  SProp: ISharedProperty;
begin
```

```
  // Получаем данные игры с помощью разделяемого свойства
  SProp := GetSharedPropertyGroup.CreateProperty(Format(GameDataStr,
    [GameID]), Exists);
  // Получаем массив данных игры и блокируем его с целью
  // повышения эффективности доступа
  PropVal := SProp.Value;
  GameData := PGameData(VarArrayLock(PropVal));
  try
    // Если игра не окончена, позволяем компьютеру сделать ход
    GameRez := CalcGameStatus(GameData);
    if GameRez = grInProgress then
    begin
      CalcComputerMove(GameData, SkillLevel, X, Y);
      // Сохраняем новый массив данных игры
      SProp.Value := PropVal;
      // Проверяем, не конец ли игры
      GameRez := CalcGameStatus(GameData);
    end;
  finally
    VarArrayUnlock(PropVal);
  end;
  SetComplete;
end;

procedure TGameServer.PlayerMove(GameID, X, Y: Integer;
  out GameRez: GameResults);
var
  Exists: WordBool;
  PropVal: OleVariant;
  GameData: PGameData;
  SProp: ISharedProperty;
begin
  // Получаем данные игры с помощью разделяемого свойства
  SProp := GetSharedPropertyGroup.CreateProperty(Format(GameDataStr,
    [GameID]), Exists);
  // Получаем массив данных игры и блокируем его с целью
  // повышения эффективности доступа
  PropVal := SProp.Value;
  GameData := PGameData(VarArrayLock(PropVal));
  try
    // Убеждаемся в том, что игра еще не окончена
    GameRez := CalcGameStatus(GameData);
    if GameRez = grInProgress then
    begin
      // Если клеточка не пуста, генерируем исключение
      if GameData[X, Y] <> EmptySpot then
        raise Exception.Create('Клетка занята!');
      // Разрешаем сделать ход
      GameData[X, Y] := PlayerSpot;
      // Сохраняем массив данных о новой игре
```

```
      SProp.Value := PropVal;
      // Проверяем, не конец ли игры
      GameRez := CalcGameStatus(GameData);
    end;
  finally
    VarArrayUnlock(PropVal);
  end;
  SetComplete;
end;
```

Эти методы схожи в том, что оба они получают данные об игре от общего свойства на основе параметра `GameID`, манипулируют этими данными для отражения текущего хода, снова сохраняют данные об игре и проверяют, не окончилась ли игра. Метод `ComputerMove()` также вызывает метод `CalcComputerMove()`, чтобы проанализировать игру и сделать ход. Если вы хотите разобраться в этом процессе, а также в другой логике работы компонента MTS, обратитесь к листингу 23.15, содержащему полный исходный текст модуля `ServMain`.

Листинг 23.15. Модуль `Main.pas`: класс `TGameServer`

```
unit ServMain;

interface

uses
  ActiveX, MtsObj, Mtx, ComObj, TTTServer_TLB;

type
  PGameData = ^TGameData;
  TGameData = array[1..3, 1..3] of Byte;

  TGameServer = class(TMtsAutoObject, IGameServer)
  private
    procedure CalcComputerMove(GameData: PGameData; Skill: SkillLevels;
      var X, Y: Integer);
    function CalcGameStatus(GameData: PGameData): GameResults;
    function GetSharedPropertyGroup: ISharedPropertyGroup;
    procedure CheckCallerSecurity;
  protected
    procedure NewGame(out GameID: Integer); safecall;
    procedure ComputerMove(GameID: Integer; SkillLevel: SkillLevels; out X,
      Y: Integer; out GameRez: GameResults); safecall;
    procedure PlayerMove(GameID, X, Y: Integer; out GameRez: GameResults);
      safecall;
  end;

implementation

uses ComServ, Windows, SysUtils;

const
  GameDataStr = 'TTTGameData%d';
```

```
  EmptySpot = 0;
  PlayerSpot = $1;
  ComputerSpot = $2;

function TGameServer.GetSharedPropertyGroup: ISharedPropertyGroup;
var
  SPGMgr: ISharedPropertyGroupManager;
  LockMode, RelMode: Integer;
  Exists: WordBool;
begin
  if ObjectContext = nil then
    raise Exception.Create('Не удается получить контекст объекта');
// Создаем группу совместно используемых свойств для этого объекта
OleCheck(ObjectContext.CreateInstance(CLASS_SharedPropertyGroupManager,
    ISharedPropertyGroupManager, SPGMgr));
  LockMode := LockSetGet;
  RelMode := Process;
  Result := SPGMgr.CreatePropertyGroup('DelphiTTT', LockMode, RelMode, Exists);
  if Result = nil then
    raise Exception.Create('Не удается получить группу свойств');
end;

procedure TGameServer.NewGame(out GameID: Integer);
var
  SPG: ISharedPropertyGroup;
  SProp: ISharedProperty;
  Exists: WordBool;
  GameData: OleVariant;
begin
  // Используем роль источника вызова для проверки безопасности
  CheckCallerSecurity;
  // Получаем группу совместно используемых свойств для этого объекта
  SPG := GetSharedPropertyGroup;
  // Создаем или считываем совместно используемое свойство NextGameID
  SProp := SPG.CreateProperty('NextGameID', Exists);
  if Exists then GameID := SProp.Value
  else GameID := 0;
  // Инкрементируем и сохраняем совместно используемое свойство NextGameID
  SProp.Value := GameID + 1;
  // Создаем массив данных об игре
  GameData := VarArrayCreate([1, 3, 1, 3], varByte);
  SProp := SPG.CreateProperty(Format(GameDataStr, [GameID]), Exists);
  SProp.Value := GameData;
  SetComplete;
end;

procedure TGameServer.ComputerMove(GameID: Integer;
  SkillLevel: SkillLevels; out X, Y: Integer; out GameRez: GameResults);
var
  Exists: WordBool;
  PropVal: OleVariant;
```

```
  GameData: PGameData;
  SProp: ISharedProperty;
begin
  // Получаем данные об игре с помощью совместно используемого свойства
  SProp := GetSharedPropertyGroup.CreateProperty(Format(GameDataStr,
    [GameID]), Exists);
  // Получаем массив данных об игре и блокируем его с целью
  // повышения эффективности доступа
  PropVal := SProp.Value;
  GameData := PGameData(VarArrayLock(PropVal));
  try
    // Если игра не окончена, позволяем компьютеру сделать ход
    GameRez := CalcGameStatus(GameData);
    if GameRez = grInProgress then
    begin
      CalcComputerMove(GameData, SkillLevel, X, Y);
      // Сохраняем массив данных о новой игре
      SProp.Value := PropVal;
      // Проверяем, окончена ли игра
      GameRez := CalcGameStatus(GameData);
    end;
  finally
    VarArrayUnlock(PropVal);
  end;
  SetComplete;
end;

procedure TGameServer.PlayerMove(GameID, X, Y: Integer;
  out GameRez: GameResults);
var
  Exists: WordBool;
  PropVal: OleVariant;
  GameData: PGameData;
  SProp: ISharedProperty;
begin
  // Получаем данные об игре с помощью совместно используемого свойства
  SProp := GetSharedPropertyGroup.CreateProperty(Format(GameDataStr,
    [GameID]), Exists);
  // Получаем массив данных об игре и блокируем его с целью
  // повышения эффективности доступа
  PropVal := SProp.Value;
  GameData := PGameData(VarArrayLock(PropVal));
  try
    // Убеждаемся, что игра не окончена
    GameRez := CalcGameStatus(GameData);
    if GameRez = grInProgress then
    begin
      // Если клеточка не пуста, генерируем исключение
      if GameData[X, Y] <> EmptySpot then
        raise Exception.Create('Клетка занята!');
      // Позволяем сделать ход
```

```
      GameData[X, Y] := PlayerSpot;
      // Сохраняем массив данных о новой игре
      SProp.Value := PropVal;
      // Проверяем, не окончена ли игра
      GameRez := CalcGameStatus(GameData);
    end;
  finally
    VarArrayUnlock(PropVal);
  end;
  SetComplete;
end;

function TGameServer.CalcGameStatus(GameData: PGameData): GameResults;
var
  I, J: Integer;
begin
  // Сначала проверяем на наличие победителя
  if GameData[1, 1] <> EmptySpot then
  begin
    // Проверяем верхнюю строку, левый столбец и диагональ с
    // левого верха до правого низа на предмет выигрыша
    if ((GameData[1, 1] = GameData[1, 2]) and (GameData[1, 1] =
      GameData[1, 3])) or((GameData[1, 1] = GameData[2, 1]) and
      (GameData[1, 1] = GameData[3, 1])) or ((GameData[1, 1] =
      GameData[2, 2]) and (GameData[1, 1] = GameData[3, 3])) then
    begin
      Result := GameData[1, 1] + 1; // Результат игры равен ID клетки + 1
      Exit;
    end;
  end;
  if GameData[3, 3] <> EmptySpot then
  begin
    // Проверяем нижнюю строку и правый столбец на предмет выигрыша
    if ((GameData[3, 3] = GameData[3, 2]) and (GameData[3, 3] =
      GameData[3, 1])) or ((GameData[3, 3] = GameData[2, 3]) and
      (GameData[3, 3] = GameData[1, 3])) then
    begin
      Result := GameData[3, 3] + 1; // Результат игры равен ID клетки + 1
      Exit;
    end;
  end;
  if GameData[2, 2] <> EmptySpot then
  begin
    // Проверяем среднюю строку, средний столбец и диагональ
    // с левого низа до правого верха на предмет выигрыша
    if ((GameData[2, 2] = GameData[2, 1]) and (GameData[2, 2] =
      GameData[2, 3])) or ((GameData[2, 2] = GameData[1, 2]) and
      (GameData[2, 2] = GameData[3, 2])) or ((GameData[2, 2] =
      GameData[3, 1]) and (GameData[2, 2] = GameData[1, 3])) then
    begin
      Result := GameData[2, 2] + 1; // Результат игры равен ID клетки ID + 1
```

```
      Exit;
    end;
  end;
  // Наконец, проверяем, продолжается ли еще игра
  for I := 1 to 3 do
    for J := 1 to 3 do
      if GameData[I, J] = 0 then
      begin
        Result := grInProgress;
        Exit;
      end;
  // Если мы дошли сюда, значит, ничья
  Result := grTie;
end;

procedure TGameServer.CalcComputerMove(GameData: PGameData;
  Skill: SkillLevels; var X, Y: Integer);
type
  // Используется для сканирования возможных ходов по строке,
  // по столбцу или по диагонали
  TCalcType = (ctRow, ctColumn, ctDiagonal);
  // mtWin = один ход до выигрыша, mtBlock = оппоненту один ход до выигрыша,
  // mtOne = я занимаю еще одну клетку в этой строке, mtNew = я не занимаю
  // ни одной клетки в этой строке
  TMoveType = (mtWin, mtBlock, mtOne, mtNew);
var
  CurrentMoveType: TMoveType;

  function DoCalcMove(CalcType: TCalcType; Position: Integer): Boolean;
  var
    RowData, I, J, CheckTotal: Integer;
    PosVal, Mask: Byte;
  begin
    Result := False;
    RowData := 0;
    X := 0;
    Y := 0;
    if CalcType = ctRow then
    begin
      I := Position;
      J := 1;
end
    else if CalcType = ctColumn then
    begin
      I := 1;
      J := Position;
    end
    else begin
      I := 1;
      case Position of
        1: J := 1; // Сканирование от левого верха к правому низу
```

```
    2: J := 3; // Сканирование от правого верха к левому низу
  else
    Exit;   // Сканируется только 2-я диагональ
  end;
end;
// Переменная Mask отключает Игрока или Компьютер, в зависимости
// от того, за кого мы играем – за нападение или защиту.
// Переменная Checktotal определяет, та ли это строка,
// в которую нам нужно сделать ход.
case CurrentMoveType of
  mtWin:
    begin
      Mask := PlayerSpot;
      CheckTotal := 4;
    end;
  mtNew:
    begin
      Mask := PlayerSpot;
      CheckTotal := 0;
    end;
  mtBlock:
    begin
      Mask := ComputerSpot;
      CheckTotal := 2;
    end;
else
  begin
    Mask := 0;
    CheckTotal := 2;
  end;
end;
// Проходим в цикле все линии в текущем варианте направления CalcType
repeat
  // Получаем статус текущей клетки: (X, O или пусто)
  PosVal := GameData[I, J];
  // Запоминаем последнюю пустую клетку в случае, если решено ходить сюда
  if PosVal = 0 then
  begin
    X := I;
    Y := J;
  end
  else
    // Если клетка не пуста, добавляем маскированное значение
    // к значению RowData
    Inc(RowData, (PosVal and not Mask));
  if (CalcType = ctDiagonal) and (Position = 2) then
  begin
    Inc(I);
    Dec(J);
  end
  else begin
```

```
      if CalcType in [ctRow, ctDiagonal] then Inc(J);
      if CalcType in [ctColumn, ctDiagonal] then Inc(I);
    end;
  until (I > 3) or (J > 3);
  // Если значение RowData равно CheckTotal, мы должны выигрывать или
  // блокировать выигрыш в зависимости от того, нападаем мы или защищаемся.
  Result := (X <> 0) and (RowData = CheckTotal);
  if Result then
  begin
    GameData[X, Y] := ComputerSpot;
    Exit;
  end;
end;

var
  A, B, C: Integer;
begin
  if Skill = slAwake then
  begin
    // Сначала рассматриваем возможность выигрыша,
    // а затем - блокирования выигрыша
    for A := Ord(mtWin) to Ord(mtBlock) do
    begin
      CurrentMoveType := TMoveType(A);
      for B := Ord(ctRow) to Ord(ctDiagonal) do
        for C := 1 to 3 do
          if DoCalcMove(TCalcType(B), C) then Exit;
    end;
    // Затем рассматриваем возможность занять центральную клетку поля
    if GameData[2, 2] = 0 then
    begin
      GameData[2, 2] := ComputerSpot;
      X := 2;
      Y := 2;
      Exit;
    end;
    // Затем ищем самую выгодную позицию в линии
    for A := Ord(mtOne) to Ord(mtNew) do
    begin
      CurrentMoveType := TMoveType(A);
      for B := Ord(ctRow) to Ord(ctDiagonal) do
        for C := 1 to 3 do
          if DoCalcMove(TCalcType(B), C) then Exit;
    end;
  end;
  // Наконец (или в случае бессознательного уровня мастерства),
  // просто находим первую попавшуюся свободную клетку
  for A := 1 to 3 do
    for B := 1 to 3 do
      if GameData[A, B] = 0 then
      begin
```

```
        GameData[A, B] := ComputerSpot;
        X := A;
        Y := B;
        Exit;
      end;
end;

procedure TGameServer.CheckCallerSecurity;
begin
  // Только ради шутки: позволяем играть в эту игру только тем,
  // кому назначена роль "TTT".
  if IsSecurityEnabled and not IsCallerInRole('TTT') then
    raise Exception.Create('Только в роли TTT можно играть в игру tic-tac-toe');
end;

initialization
  TAutoObjectFactory.Create(ComServer, TGameServer, Class_GameServer,
    ciMultiInstance, tmApartment);
end.
```

Инсталляция сервера

Если вы уже написали сервер и готовы инсталлировать его в среде MTS, Delphi максимально упростит эту процедуру. Просто выберите в меню команду Run⇨Install MTS Objects, в результате чего откроется диалоговое окно Install MTS Objects. В этом диалоговом окне можно установить объект (объекты) в новый или существующий пакет, как показано на рис. 23.21.

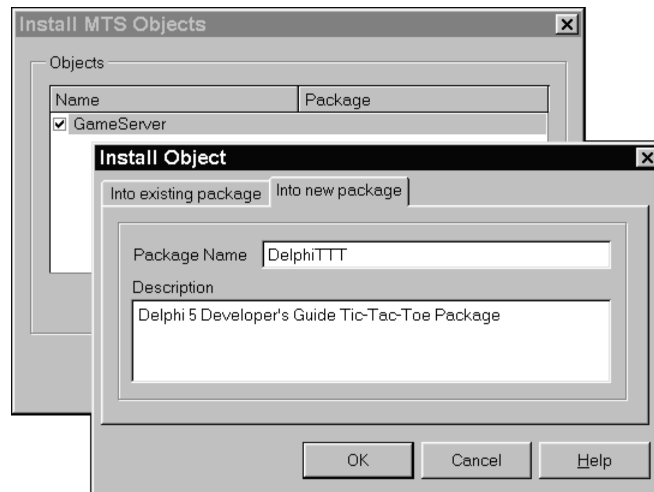


Рис. 23.21. Установка объекта MTS с помощью IDE Delphi

Выберите подлежащий установке компонент (компоненты), укажите использование нового пакета или уже существующего, щелкните на кнопке OK, и дело с концом — компонент будет установлен. В качестве альтернативного варианта можно также инсталлировать компоненты MTS, используя приложение Transaction Server Explorer. Заметьте: эта процедура ин-

сталляции явно отличается от установки стандартных объектов COM, которая для регистрации сервера COM предусматривает вызов утилиты RegSvr32 из командной строки. Приложение Transaction Server Explorer также просто справляется и с установкой компонентов MTS на удаленных компьютерах, являясь тем самым приятной альтернативой тем трудностям конфигурации, которые испытали на себе многие программисты, пытавшиеся настроить DCOM-подключение.

Приложение клиента

В листинге 23.16 представлен исходный текст приложения клиента для этого компонента MTS. Его назначение состоит в отображении механизма, обеспечиваемого компонентом MTS, на пользовательский интерфейс, представленный в виде поля для игры в крестики-нолики.

Листинг 23.16. Модуль `UiMain.pas` — главный модуль для приложения клиента

```
unit UiMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Buttons, ExtCtrls, Menus, TTTServer_TLB, ComCtrls;

type
  TRecord = record
    Wins, Loses, Ties: Integer;
  end;
TFrmMain = class(TForm)
    SbTL: TSpeedButton;
    SbTM: TSpeedButton;
    SbTR: TSpeedButton;
    SbMM: TSpeedButton;
    SbBL: TSpeedButton;
    SbBR: TSpeedButton;
    SbMR: TSpeedButton;
    SbBM: TSpeedButton;
    SbML: TSpeedButton;
    Bevel1: TBevel;
    Bevel2: TBevel;
    Bevel3: TBevel;
    Bevel4: TBevel;
    MainMenu1: TMainMenu;
    FileItem: TMenuItem;
    HelpItem: TMenuItem;
    ExitItem: TMenuItem;
    AboutItem: TMenuItem;
    SkillItem: TMenuItem;
    UnconItem: TMenuItem;
    AwakeItem: TMenuItem;
```

```
    NewGameItem: TMenuItem;
    N1: TMenuItem;
    StatusBar: TStatusBar;
    procedure FormCreate(Sender: TObject);
    procedure ExitItemClick(Sender: TObject);
    procedure SkillItemClick(Sender: TObject);
    procedure AboutItemClick(Sender: TObject);
    procedure SBClick(Sender: TObject);
    procedure NewGameItemClick(Sender: TObject);
  private
    FXImage: TBitmap;
    FOImage: TBitmap;
    FCurrentSkill: Integer;
    FGameID: Integer;
    FGameServer: IGameServer;
    FRec: TRecord;
    procedure TagToCoord(ATag: Integer; var Coords: TPoint);
    function CoordToCtl(const Coords: TPoint): TSpeedButton;
    procedure DoGameResult(GameRez: GameResults);
  end;

var
  FrmMain: TFrmMain;

implementation

uses UiAbout;

{$R *.DFM}

{$R xo.res}

const
  RecStr = 'Wins: %d, Loses: %d, Ties: %d';

procedure TFrmMain.FormCreate(Sender: TObject);
begin
  // Загружаем изображения "X" и "O" из файла ресурсовв компонент TBitmaps
  FXImage := TBitmap.Create;
  FXImage.LoadFromResourceName(MainInstance, 'x_img');
  FOImage := TBitmap.Create;
  FOImage.LoadFromResourceName(MainInstance, 'o_img');
  // Устанавливаем стандартный уровень мастерства
  FCurrentSkill := slAwake;
  // Инициализируем UI записи
  with FRec do
   StatusBar.SimpleText := Format(RecStr, [Wins, Loses, Ties]);
  // Получаем экземпляр сервера
  FGameServer := CoGameServer.Create;
  // Запускаем новую игру
```

```
  FGameServer.NewGame(FGameID);
end;

procedure TFrmMain.ExitItemClick(Sender: TObject);
begin
  Close;
end;

procedure TFrmMain.SkillItemClick(Sender: TObject);
begin
  with Sender as TMenuItem do
  begin
    Checked := True;
    FCurrentSkill := Tag;
  end;
end;

procedure TFrmMain.AboutItemClick(Sender: TObject);
begin
  // Отображаем окно About
  with TFrmAbout.Create(Application) do
    try
      ShowModal;
    finally
      Free;
    end;
end;

procedure TFrmMain.TagToCoord(ATag: Integer; var Coords: TPoint);
begin
  case ATag of
    0: Coords := Point(1, 1);
    1: Coords := Point(1, 2);
    2: Coords := Point(1, 3);
    3: Coords := Point(2, 1);
    4: Coords := Point(2, 2);
    5: Coords := Point(2, 3);
    6: Coords := Point(3, 1);
    7: Coords := Point(3, 2);
  else
    Coords := Point(3, 3);
  end;
end;

function TFrmMain.CoordToCtl(const Coords: TPoint): TSpeedButton;
begin
  Result := nil;
  with Coords do
    case X of
      1:
```

```
        case Y of
          1: Result := SbTL;
          2: Result := SbTM;
          3: Result := SbTR;
        end;
      2:
        case Y of
          1: Result := SbML;
          2: Result := SbMM;
          3: Result := SbMR;
        end;
      3:
        case Y of
          1: Result := SbBL;
          2: Result := SbBM;
          3: Result := SbBR;
        end;
    end;
end;

procedure TFrmMain.SBClick(Sender: TObject);
var
  Coords: TPoint;
  GameRez: GameResults;
  SB: TSpeedButton;
begin
  if Sender is TSpeedButton then
  begin
    SB := TSpeedButton(Sender);
    if SB.Glyph.Empty then
    begin
      with SB do
      begin
        TagToCoord(Tag, Coords);
        FGameServer.PlayerMove(FGameID, Coords.X, Coords.Y, GameRez);
        Glyph.Assign(FXImage);
      end;
      if GameRez = grInProgress then
      begin
        FGameServer.ComputerMove(FGameID, FCurrentSkill, Coords.X,
          Coords.Y, GameRez);
CoordToCtl(Coords).Glyph.Assign(FOImage);
      end;
      DoGameResult(GameRez);
    end;
  end;
end;

procedure TFrmMain.NewGameItemClick(Sender: TObject);
var
```

```
  I: Integer;
begin
  FGameServer.NewGame(FGameID);
  for I := 0 to ControlCount - 1 do
    if Controls[I] is TSpeedButton then
      TSpeedButton(Controls[I]).Glyph := nil;
end;

procedure TFrmMain.DoGameResult(GameRez: GameResults);
const
  EndMsg: array[grTie..grComputerWin] of string = (
'Ничья', 'Вы выиграли', 'Компьютер выиграл');
begin
  if GameRez <> grInProgress then
  begin
    case GameRez of
      grComputerWin: Inc(FRec.Loses);
      grPlayerWin: Inc(FRec.Wins);
      grTie: Inc(FRec.Ties);
    end;
    with FRec do
     StatusBar.SimpleText := Format(RecStr, [Wins, Loses, Ties]);
    if MessageDlg(Format('%s! Играем снова?', [EndMsg[GameRez]]),
      mtConfirmation, [mbYes, mbNo], 0) = mrYes then
      NewGameItemClick(nil);
  end;
end;

end.
```

На рис. 23.22 это приложение показано в действии. Пользователь играет “крестиками” (X), а компьютер — “ноликами” (O).

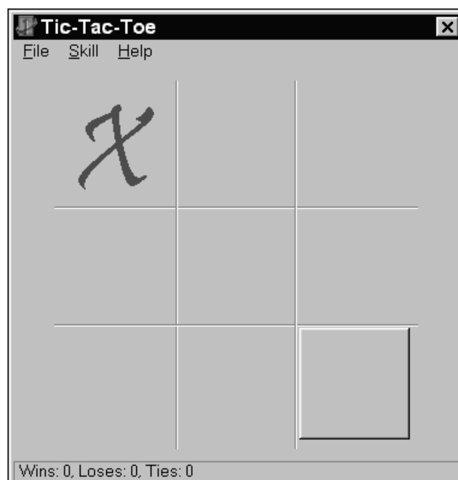


Рис. 23.22. Игра в “крестики-нолики”

Отладка MTS-приложений

Поскольку компоненты MTS работают внутри пространства процесса MTS, а не внутри пространства клиента, то можно подумать, что отладка будет чрезвычайно трудной. Однако в технологии MTS для целей отладки предусмотрена “лазейка”, значительно облегчающая этот процесс. Просто загрузите проект сервера и используйте диалоговое окно Run Parameters, чтобы задать `mtx.exe` как основное приложение. В качестве параметра для приложения `mtx.exe` нужно передать выражение `/p:{`*`package guid`*`}`, где элемент *`package guid`* представляет собой уникальный идентификатор GUID пакета, отображаемый в приложении Transaction Server Explorer. Это диалоговое окно показано на рис. 23.23. Затем определите желаемые точки останова (прерывания) и запустите серверное приложение. Поначалу ничего не произойдет, поскольку приложение клиента еще не выполняется. Теперь запустите в работу клиентское приложение — из окна Windows Explorer или из командной строки — и приступайте к отладке сервера.

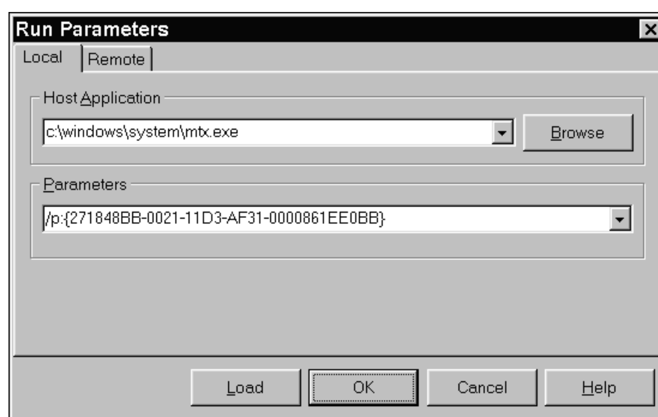


Рис. 23.23. Использование диалогового окна Run Parameters *для организации сеанса отладки сервера MTS*

MTS — это мощное дополнение к семейству технологий COM. Путем добавления к COM-объектам таких новых возможностей, как управление временем жизни, поддержка транзакций и расширение средств защиты, причем без необходимости внесения значительных изменений в уже существующий программный текст, компания Microsoft перевела COM в разряд масштабируемых технологий, пригодных для распределенной разработки широкого масштаба. В этом разделе вы познакомились с основами, на которых построена технология MTS, и с особенностями поддержки MTS в Delphi. Мы также рассмотрели процесс создания MTS-приложений в интегрированной среде разработки Delphi. Надеемся, что вам пригодятся наши советы и рекомендации по разработке оптимизированных компонентов MTS с заданным поведением. Интересно то, что все средства MTS (управление временем жизни объектов, поддержка транзакций и безопасности) реализованы в уже знакомой нам оболочке. Технология MTS и среда Delphi объединены общей главной целью — предоставить способ для накопления опыта работы с технологией COM и создания расширяемых многоуровневых приложений. Только не следует забывать об описанных выше различиях в нюансах разработки обычных компонентов COM и компонентов MTS!

Класс TOleContainer

Теперь, рассмотрев основы технологий ActiveX и OLE, давайте познакомимся с классом Delphi `TOleContainer`. Класс `TOleContainer` определен в модуле `OleCntrs` и инкапсулирует сложности работы с контейнерами документов OLE и ActiveX в виде простого и удобного компонента библиотеки VCL.

На заметку

Если вы уже знакомы с использованием компонента `TOleContainer` в Delphi 1, не нужно пропускать этот раздел. Этот компонент был заново переписан в Delphi 2, поэтому ваши знания о разработке в Delphi 16-разрядных приложений будут практически неприменимы при разработке 32-разрядных приложений. Но не нужно расстраиваться: 32-разрядная версия этого компонента более проста в использовании, и код, который необходимо написать для поддержки объекта в новой версии, гораздо короче того, что приходилось писать ранее.

Простейший пример приложения

Теперь приступим к созданию приложения с OLE-контейнером. Создайте новый проект, щелкните на вкладке System в палитре компонентов и поместите в форму компонент `TOleContainer`. Щелкните правой кнопкой на созданном объекте в окне конструктора форм и выберите в появившемся контекстном меню команду Insert Object. Раскроется диалоговое окно Insert Object (Вставка объекта), показанное на рис. 23.24.

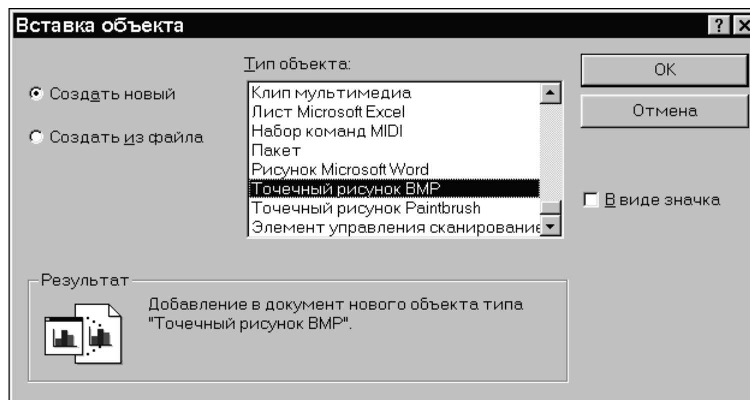


Рис. 23.24. Диалоговое окно Insert Object

Внедрение нового OLE-объекта

По умолчанию в диалоговом окне Insert Object содержатся имена приложений OLE-серверов, зарегистрированных в Windows. Для внедрения нового OLE-объекта необходимо выбрать приложение-сервер из списка Object Type (Тип объекта). Это заставит OLE-сервер запуститься для создания нового OLE-объекта, который будет вставлен в объект `TOleContainer`. При закрытии приложения-сервера объект `TOleContainer` будет обновлен, и в нем появится изображение внедренного объекта. Например, можно создать новый документ Microsoft Word, как показано на рис. 23.25.

На заметку

LE-объект не может быть активизирован во время разработки. Воспользоваться преимуществами активизации объекта `TOleContainer` можно только во время выполнения.

Чтобы открыть диалоговое окно Insert Object во время выполнения, можно вызвать метод `InsertObjectDialog()` класса `TOleContainer`, который определен следующим образом:

```
function InsertObjectDialog: Boolean;
```

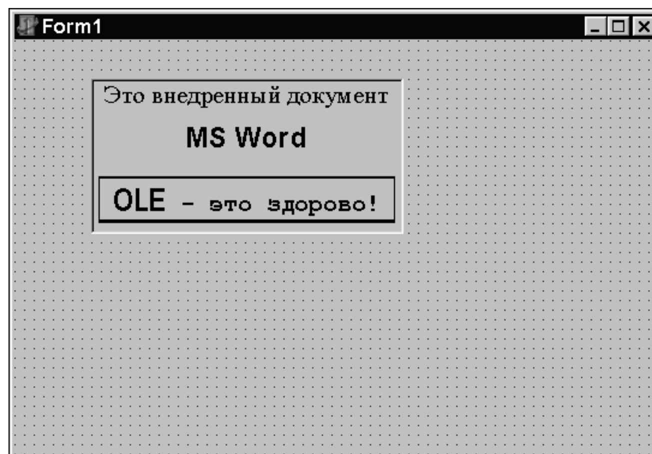


Рис. 23.25. Внедренный документ Microsoft Word

Эта функция возвращает значение `True`, если новый тип OLE-объекта был успешно выбран из списка диалогового окна Insert Object.

Внедрение или связывание существующего OLE-файла

Для внедрения существующего OLE-файла в объект `TOleContainer` установите переключатель в диалоговом окне Insert Object в положение Create From File (Создать из файла), а затем выберите существующий файл, как показано на рис. 23.26. После выбора файл будет вести себя как новый OLE-объект.

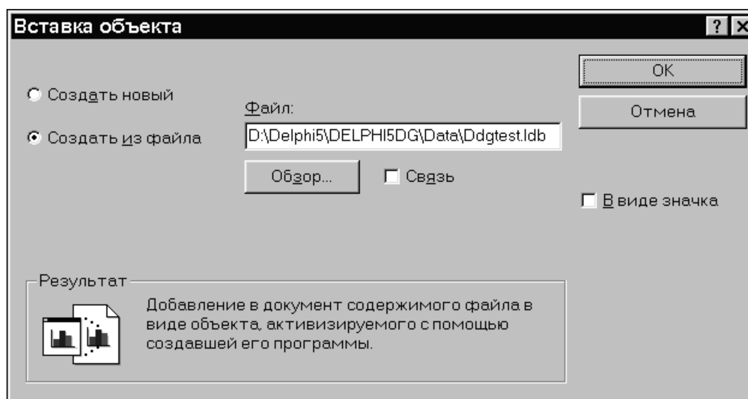


Рис. 23.26. Вставка объекта из файла

Для внедрения файла во время выполнения необходимо вызвать метод `CreateObjectFromFile()` класса `TOleContainer`, который определен следующим образом:

```
procedure CreateObjectFromFile(const FileName: string; Iconic: Boolean);
```

Для связывания (но не внедрения) OLE-объекта просто установите флажок Link (Связь) в диалоговом окне Insert Object (см. рис. 23.26). Как было указано выше, это действие

приводит к созданию связи между приложением и OLE-файлом, благодаря которой можно будет просматривать и редактировать один и тот же связанный объект сразу из нескольких приложений.

Для связывания файла с приложением во время выполнения вызовите метод `CreateLinkToFile()` класса `TOleContainer`, который определен следующим образом:

```
procedure CreateLinkToFile(const FileName: string; Iconic: Boolean);
```

Пример помощнее

Познакомившись с основами технологии OLE и классом `TOleContainer`, можно приступить к рассмотрению примера, который по-настоящему продемонстрирует возможности использования OLE в реальных приложениях.

Начнем с создания нового проекта, основанного на шаблоне MDI-приложения. Основная форма этого приложения была немного модифицирована после использования стандартного MDI-шаблона (рис. 23.27).

Дочерняя MDI-форма показана на рис. 23.28. Это пример формы стиля `fsMDIChild` с компонентном `TOleContainer`, наложенным на форму с параметром `alClient` (Занять всю клиентскую область).

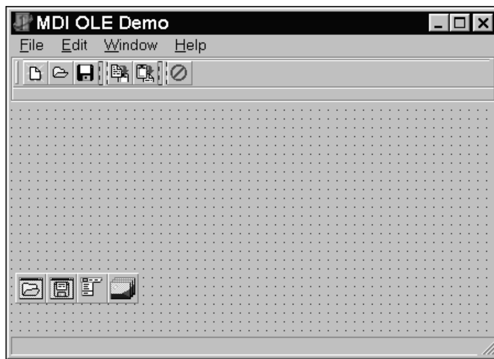


Рис. 23.27. Главное окно демонстрационного примера MDI-приложения с OLE-объектом

Рис. 23.28. Дочерняя форма создаваемого приложения

В листинге 23.17 приведен исходный код модуля дочерней формы MDI-приложения (`ChildWin.pas`). Заметьте, что этот модуль имеет стандартный вид, за исключением добавленного свойства `OLEFileName`, связанного с ним метода и закрытой переменной экземпляра. Свойство `OLEFileName` предназначено для хранения пути и имени OLE-файла, а соответствующий метод доступа к свойству устанавливает заголовок дочерней формы равным имени файла.

Листинг 23.17. Исходный код модуля `ChildWin.pas`

```
unit Childwin;

interface

uses WinTypes, WinProcs, Classes, Graphics, Forms, Controls, OleCtnrs;
```

```
type
  TMDIChild = class(TForm)
    OleContainer: TOleContainer;
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    FOLEFilename: String;
    procedure SetOLEFileName(const Value: String);
  public
    property OLEFileName: String read FOLEFileName write SetOLEFileName;
  end;

implementation

{$R *.DFM}

uses Main, SysUtils;

procedure TMDIChild.SetOLEFileName(const Value: String);
begin
  if Value <> FOLEFileName then begin
    FOLEFileName := Value;
    Caption := ExtractFileName(FOLEFileName);
  end;
end;

procedure TMDIChild.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
end;

end.
```

Создание дочерней формы

При создании с помощью команды File⇨New новой дочерней формы MDI-приложения, посредством вызова описанного выше метода `InsertObjectDialog()` открывается диалоговое окно Insert Object. Заголовок дочерней MDI-формы устанавливается с помощью глобальной переменной `NumChildren`, обеспечивающей уникальный номер. В приведенном ниже фрагменте кода показан метод `CreateMDIChild()` главной формы:

```
procedure TMainForm.FileNewItemClick(Sender: TObject);
begin
  inc(NumChildren);
  { Создание нового дочернего MDI-окна }
  with TMDIChild.Create(Application) do
  begin
    Caption := 'Untitled' + IntToStr(NumChildren);
    { Открываем диалоговое окно вставки OLE-объекта и выполняем
      вставку в дочернюю форму }
    OleContainer.InsertObjectDialog;
  end;
end;
```

Сохранение в файл и чтение из файла

Как уже отмечалось в этой главе, OLE-объекты позволяют записывать и считывать информацию из потоков и, следовательно, из файлов. Компонент `TOleContainer` содержит методы `SaveToStream()`, `LoadFromStream()`, `SaveToFile()` и `LoadFromFile()`, которые упрощают сохранение OLE-объектов в файле или потоке.

Главная форма приложения `MDIOLE` содержит методы для открытия и сохранения файлов с OLE-объектами. В приведенном ниже фрагменте кода показан метод `FileOpenItemClick()`, который вызывается при выборе команды File⇨Open в главной форме. Кроме загрузки сохраненного OLE-объекта из файла, заданного свойством `OpenDialog.FileName`, этот метод также присваивает полю `OleFileName` экземпляра `TMDIChild` имя этого файла. Если при загрузке файла возникает ошибка, экземпляр формы будет освобожден:

```
procedure TMainForm.FileOpenItemClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    with TMDIChild.Create(Application) do
    begin
      try
        OleFileName := OpenDialog.FileName;
        OleContainer.LoadFromFile(OleFileName);
        Show;
      except
        Release;    // Освобождаем форму при ошибке
        raise;      // Генерируем исключительную ситуацию
      end;
    end;
end;
```

В следующем фрагменте показан код, выполняемый при выборе команд меню File⇨Save As и File⇨Save. Заметьте, что метод `FileSaveItemClick()` вызывает метод `FileSaveAsItemClick()`, если активной дочерней MDI-форме еще не присвоено имя.

```
procedure TMainForm.FileSaveAsItemClick(Sender: TObject);
begin
  if (ActiveMDIChild <> nil) and (SaveDialog.Execute) then
    with TMDIChild(ActiveMDIChild) do
    begin
      OleFileName := SaveDialog.FileName;
      OleContainer.SaveToFile(OleFileName);
    end;
end;

procedure TMainForm.FileSaveItemClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    { Если имя не присвоено, выполняем команду "save as" }
    if TMDIChild(ActiveMDIChild).OLEFileName = '' then
      FileSaveAsItemClick(Sender)
    else
```

```
    { В противном случае сохраняем с текущим именем }
    with TMDIChild(ActiveMDIChild) do
      OleContainer.SaveToFile(OLEFileName);
end;
```

Использование буфера обмена для копирования и вставки OLE-объекта

Благодаря универсальному механизму передачи данных, описанному выше в этой главе, для передачи OLE-объектов можно также использовать буфер обмена Windows. И вновь при решении этих задач не обойтись без компонента TOleContainer.

Копирование OLE-объекта из компонента TOleContainer в буфер обмена — тривиальная задача. Для этого достаточно вызвать метод Copy(), как показано в следующем фрагменте кода:

```
procedure TMainForm.CopyItemClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    TMDIChild(ActiveMDIChild).OleContainer.Copy;
end;
```

После помещения OLE-объекта в буфер обмена необходим только один дополнительный шаг для его правильного считывания в компонент TOleContainer. Перед вставкой содержимого буфера обмена в объект TOleContainer следует сначала проверить значение свойства CanPaste, чтобы удостовериться, что данные, содержащиеся в буфере обмена, представляют собой подходящий OLE-объект. После этого можно открыть диалоговое окно Paste Special (Специальная вставка) для вставки объекта в компонент TOleContainer, вызвав метод PasteSpecialDialog(), как показано в приведенном ниже фрагменте кода. Диалоговое окно Paste Special показано на рис. 23.29.

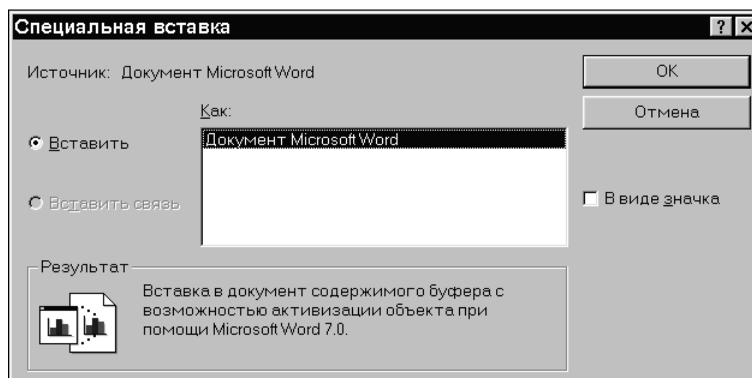


Рис. 23.29. Диалоговое окно Paste Special

```
procedure TMainForm.PasteItemClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    with TMDIChild(ActiveMDIChild).OleContainer do
      { Перед открытием диалогового окна проверьте, что }
      { в буфере обмена находятся подходящие OLE-объекты. }
      if CanPaste then PasteSpecialDialog;
end;
```

После запуска приложения сервер, управляющий OLE-объектом в активной дочерней MDI-форме, объединяется с меню и панелями инструментов приложения. На рис. 23.30 и 23.31 показаны средства активизации объекта OLE — MDI OLE-приложение управляется двумя различными OLE-серверами.

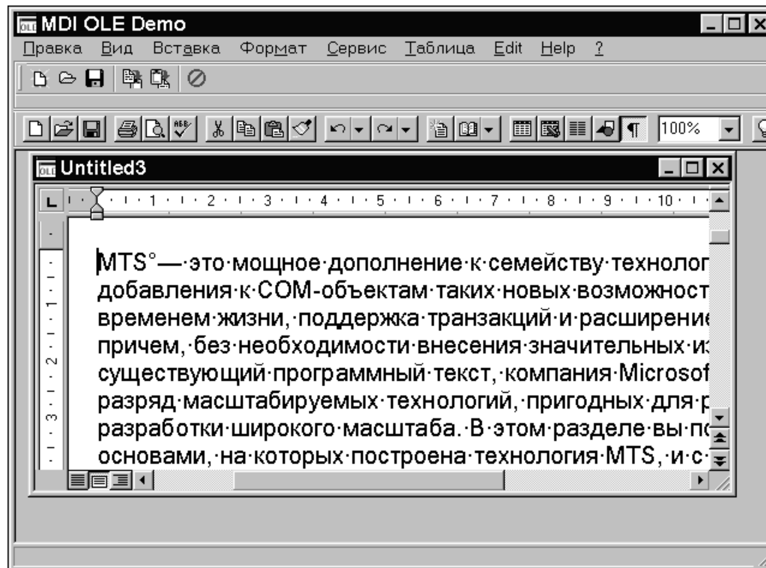


Рис. 23.30. Редактирование внедренного документа Microsoft Word

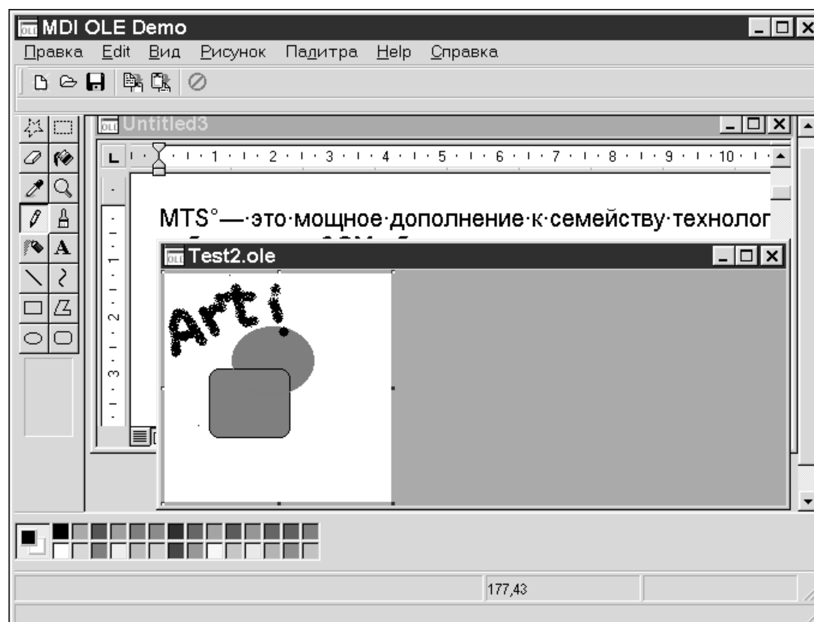


Рис. 23.31. Редактирование внедренного рисунка

Полный исходный код главного модуля MDI OLE-приложения (`Main.pas`) приведен в листинге 23.18.

Листинг 23.18. Исходный код модуля `Main.pas`

```
unit Main;

interface

uses WinTypes, WinProcs, SysUtils, Classes, Graphics, Forms, Controls,
  Menus, StdCtrls, Dialogs, Buttons, Messages, ExtCtrls, ChildWin,
  ComCtrls, ToolWin;

type
  TMainForm = class(TForm)
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    FileNewItem: TMenuItem;
    FileOpenItem: TMenuItem;
    FileCloseItem: TMenuItem;
    Window1: TMenuItem;
    Help1: TMenuItem;
    N1: TMenuItem;
    FileExitItem: TMenuItem;
    WindowCascadeItem: TMenuItem;
    WindowTileItem: TMenuItem;
    WindowArrangeItem: TMenuItem;
    HelpAboutItem: TMenuItem;
    OpenDialog: TOpenDialog;
    FileSaveItem: TMenuItem;
    FileSaveAsItem: TMenuItem;
    Edit1: TMenuItem;
    PasteItem: TMenuItem;
    WindowMinimizeItem: TMenuItem;
    SaveDialog: TSaveDialog;
    CopyItem: TMenuItem;
    CloseAll1: TMenuItem;
    StatusBar: TStatusBar;
    CoolBar1: TCoolBar;
    ToolBar1: TToolBar;
    OpenBtn: TToolButton;
    SaveBtn: TToolButton;
    ToolButton3: TToolButton;
    CopyBtn: TToolButton;
    PasteBtn: TToolButton;
    ToolButton6: TToolButton;
    ExitBtn: TToolButton;
    ImageList1: TImageList;
    procedure FormCreate(Sender: TObject);
    procedure FileNewItemClick(Sender: TObject);
    procedure WindowCascadeItemClick(Sender: TObject);
```

```
    procedure UpdateMenuItems(Sender: TObject);
    procedure WindowTileItemClick(Sender: TObject);
    procedure WindowArrangeItemClick(Sender: TObject);
    procedure FileCloseItemClick(Sender: TObject);
    procedure FileOpenItemClick(Sender: TObject);
    procedure FileExitItemClick(Sender: TObject);
    procedure FileSaveItemClick(Sender: TObject);
    procedure FileSaveAsItemClick(Sender: TObject);
    procedure PasteItemClick(Sender: TObject);
    procedure WindowMinimizeItemClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure HelpAboutItemClick(Sender: TObject);
    procedure CopyItemClick(Sender: TObject);
    procedure CloseAll1Click(Sender: TObject);
  private
    procedure ShowHint(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses About;
var
  NumChildren: Cardinal = 0;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.OnHint := ShowHint;
  Screen.OnActiveFormChange := UpdateMenuItems;
end;

procedure TMainForm.ShowHint(Sender: TObject);
begin
  { Отображение подсказок в строке состояния }
  StatusBar.Panels[0].Text := Application.Hint;
end;

procedure TMainForm.FileNewItemClick(Sender: TObject);
begin
  inc(NumChildren);
  { Создание нового дочернего окна MDI}
  with TMDIChild.Create(Application) do
  begin
    Caption := 'Untitled' + IntToStr(NumChildren);
    { Открываем диалоговое окно вставки OLE-объекта и выполняем
      вставку в дочернюю форму }
    OleContainer.InsertObjectDialog;
```

```
  end;
end;

procedure TMainForm.FileOpenItemClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    with TMDIChild.Create(Application) do
    begin
      try
        OleFileName := OpenDialog.FileName;
        OleContainer.LoadFromFile(OleFileName);
        Show;
      except
        Release;    // Освобождаем форму при ошибке
        raise;      // Генерируем исключительную ситуацию
      end;
    end;
end;

procedure TMainForm.FileCloseItemClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    ActiveMDIChild.Close;
end;

procedure TMainForm.FileSaveAsItemClick(Sender: TObject);
begin
  if (ActiveMDIChild <> nil) and (SaveDialog.Execute) then
    with TMDIChild(ActiveMDIChild) do
    begin
      OleFileName := SaveDialog.FileName;
      OleContainer.SaveToFile(OleFileName);
    end;
end;

procedure TMainForm.FileSaveItemClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    { Если имя не присвоено, выполняем команду "save as" }
    if TMDIChild(ActiveMDIChild).OLEFileName = '' then
    FileSaveAsItemClick(Sender)
    else
      { В противном случае сохраняем с текущим именем }
      with TMDIChild(ActiveMDIChild) do
        OleContainer.SaveToFile(OLEFileName);
end;

procedure TMainForm.FileExitItemClick(Sender: TObject);
begin
  Close;
end;
```

```
procedure TMainForm.PasteItemClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    with TMDIChild(ActiveMDIChild).OleContainer do
      { Перед открытием диалогового окна убедимся, что }
      { в буфере обмена находятся подходящие OLE-объекты. }
      if CanPaste then PasteSpecialDialog;
end;

procedure TMainForm.WindowCascadeItemClick(Sender: TObject);
begin
  Cascade;
end;

procedure TMainForm.WindowTileItemClick(Sender: TObject);
begin
  Tile;
end;

procedure TMainForm.WindowArrangeItemClick(Sender: TObject);
begin
  ArrangeIcons;
end;
procedure TMainForm.WindowMinimizeItemClick(Sender: TObject);
var
  I: Integer;
begin
  { Необходимо сделать проход массива MDIChildren в обратном направлении}
  for I := MDIChildCount - 1 downto 0 do
    MDIChildren[I].WindowState := wsMinimized;
end;

procedure TMainForm.UpdateMenuItems(Sender: TObject);
var
  DoIt: Boolean;
begin
  DoIt := MDIChildCount > 0;
  { Только доступные параметры, если это – активный потомок }
  FileCloseItem.Enabled := DoIt;
  FileSaveItem.Enabled := DoIt;
  CloseAll1.Enabled := DoIt;
  FileSaveAsItem.Enabled := DoIt;
  CopyItem.Enabled := DoIt;
  PasteItem.Enabled := DoIt;
  CopyBtn.Enabled := DoIt;
  SaveBtn.Enabled := DoIt;
  PasteBtn.Enabled := DoIt;
  WindowCascadeItem.Enabled := DoIt;
  WindowTileItem.Enabled := DoIt;
  WindowArrangeItem.Enabled := DoIt;
  WindowMinimizeItem.Enabled := DoIt;
```

```
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  Screen.OnActiveFormChange := nil;
end;

procedure TMainForm.HelpAboutItemClick(Sender: TObject);
begin
  with TAboutBox.Create(Self) do
  begin
    ShowModal;
    Free;
  end;
end;

procedure TMainForm.CopyItemClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    TMDIChild(ActiveMDIChild).OleContainer.Copy;
end;

procedure TMainForm.CloseAll1Click(Sender: TObject);
begin
  while ActiveMDIChild <> nil do
  begin
    ActiveMDIChild.Release;      // Используем метод Release, а не Free!
    Application.ProcessMessages; // Пусть Windows примет необходимые меры
  end;
end;

end.
```

Резюме

В этой большой по объему главе рассматривались основы технологий COM, OLE и ActiveX. Знание этих основ поможет понять процессы, происходящие при выполнении приложений, созданных с использованием данных технологий. Кроме того, вы получили некоторое представление о различных типах COM-ориентированных клиентов и серверов, а также разновидностях технологий автоматизации в Delphi. В этой главе достаточное внимание было уделено теории и практике применения технологии MTS. Помимо глубокого рассмотрения технологий COM, MTS и автоматизации, желающие могли познакомиться с работой компонента библиотеки VCL `TOleContainer`.

За дополнительной информацией о COM- и ActiveX-технологиях обращайтесь к другим главам этой книги. В главе 24, "Расширение оболочки Windows", описаны примеры создания реального COM-сервера, а в главе 25, "Создание элементов управления ActiveX" — методы создания управляющих элементов ActiveX в Delphi.

Глава

24

Расширение оболочки Windows

Вывод пиктограммы на панель задач	315
Панели инструментов рабочего стола Windows	329
Ярлыки Windows	343
Расширения оболочки	361
Резюме	386

Впервые введенная в Windows 95, оболочка Windows поддерживается в Windows NT 3.51 (и более поздних версиях), Windows 98, а также в Windows 2000. Сильно отличаясь от диспетчера программ (Program Manager) из Windows 3.x, оболочка Windows включает ряд новых средств, значительно расширяющих ее возможности и позволяющих удовлетворить самые разные запросы пользователей. Однако многие из этих средств, к сожалению, очень плохо документированы. Цель данной главы — восполнить этот пробел: здесь предлагается информация и примеры, которые позволят эффективно использовать такие средства, как пиктограммы индикаторной области панели задач, панели инструментов приложений и расширения оболочки.

Вывод пиктограммы на панель задач

В этом разделе рассматривается методика инкапсуляции пиктограммы индикаторной области панели задач в компонент Delphi. В процессе построения компонента `TTrayNotifyIcon` вы узнаете, что необходимо для создания пиктограммы индикаторной области панели задач с точки зрения API, и как решаются некоторые "неразрешимые" проблемы, возникающие при попытке обеспечить функциональность пиктограммы в рамках компонента. Для тех, кто не знаком с понятием "пиктограмма индикаторной области панели задач", напоминаем, что это небольшие значки, расположенные с правой стороны панели задач Windows (рис. 24.1). Предполагается, что панель задач находится в нижней части экрана.

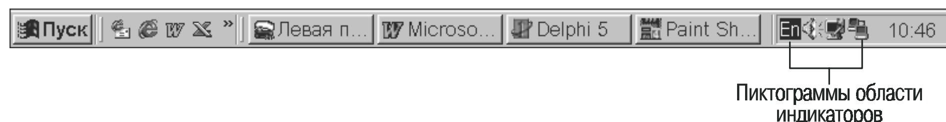


Рис. 24.1. Пиктограммы индикаторной области панели задач находятся в правом нижнем углу экрана

Интерфейс API

Хотите верьте, хотите нет, но в процессах создания, изменения и удаления пиктограмм панели задач выполняется обращение только к одной функции Win32 API — к `Shell_NotifyIcon()`. Эта и другие функции, имеющие отношение к оболочке Windows, находятся в модуле `ShellAPI`. Функция `Shell_NotifyIcon()` определяется следующим образом:

```
function Shell_NotifyIcon(dwMessage: DWORD; lpData:
  PNotifyIconData): BOOL; stdcall;
```

Параметр `dwMessage` описывает действие, выполняемое над пиктограммой, и может принимать одно из значений, приведенных в табл. 24.1.

Таблица 24.1. Значения параметра dwMessage

Константа	Значение	Назначение
NIM_ADD	0	Добавляет пиктограмму на панель задач
NIM_MODIFY	1	Модифицирует свойства существующей пиктограммы
NIM_DELETE	2	Удаляет пиктограмму с панели задач

Параметр lpData является указателем на запись типа TNotifyIconData. Эта запись определяется следующим образом:

```
type
  TNotifyIconData = record
    cbSize: DWORD;
    Wnd: HWND;
    uID: UINT;
    uFlags: UINT;
    uCallbackMessage: UINT;
    hIcon: HICON;
    szTip: array [0..63] of AnsiChar;
  end;
```

В поле cbSize хранится размер записи; это поле инициализируется с помощью функции SizeOf(TNotifyIconData).

В параметре Wnd указывается дескриптор окна, которому посылаются "обратные" (callback) сообщения от пиктограммы панели задач. (Хотя здесь используется слово *обратный*, тем не менее реально никакого обратного вызова не происходит; однако в документации по Win32 для сообщений, посылаемых окну от имени пиктограммы панели задач, используется именно такой термин.)

В поле uID задается уникальный идентификатор (ID) пиктограммы, определяемый программистом. Если в приложении используется несколько пиктограмм, нужно идентифицировать каждую из них — для этого и предназначено поле uID.

Значение в поле uFlags определяет, какие из полей записи TNotifyIconData должны учитываться функцией Shell_NotifyIcon(), а следовательно, к каким из свойств пиктограммы будут применены действия, определяемым параметром dwMessage. Этот параметр может быть любой комбинацией флагов, приведенных в табл. 24.2 (для объединения флагов воспользуйтесь ключевым словом or).

Таблица 24.2. Значения поля uFlags

Константа	Значение	Назначение
NIF_MESSAGE	0	Используется поле uCallbackMessage
NIF_ICON	2	Используется поле hIcon
NIF_TIP	4	Используется поле szTip

В поле uCallbackMessage помещается номер Windows-сообщения. Именно это сообщение будет послано окну, определяемому полем Wnd. Для получения значения этого поля обычно вызывается функция RegisterWindowMessage() (или же используется смещение от значения WM_USER). Параметр lParam посылаемого сообщения будет иметь то же значение, что и поле uID, а параметр wParam будет содержать сообщение от мыши, сгенерированное при помещении ее указателя на пиктограмму.

В поле hIcon задается дескриптор пиктограммы, помещаемой на панель задач.

Поле szTip должно содержать строку подсказки, оканчивающуюся нулевым символом. Эта строка будет выводиться на экран при подведении указателя мыши к пиктограмме.

Компонент TTrayNotifyIcon инкапсулирует функцию Shell_NotifyIcon() в методе SendTrayMessage(), текст которого приведен ниже.

```
procedure TTrayNotifyIcon.SendTrayMessage(Msg: DWORD; Flags: UINT);
{ Этот метод содержит вызов API-функции Shell_NotifyIcon. }
begin
  { Заполняем поля записи соответствующими значениями. }
  with Tnd do
  begin
    cbSize := SizeOf(Tnd);
    StrPLCopy(szTip, PChar(FHint), SizeOf(szTip));
    uFlags := Flags;
    uID := UINT(Self);
    Wnd := IconMgr.HWindow;
    uCallbackMessage := Tray_Callback;
    hIcon := ActiveIconHandle;
  end;
  Shell_NotifyIcon(Msg, @Tnd);
end;
```

В этом методе в поле szTip копируется значение закрытого поля строкового типа FHint.

Параметр uID используется для хранения ссылки на параметр Self. Поскольку эти данные будут включаться во все последующие сообщения пиктограммы панели задач, выделение среди сообщений от нескольких пиктограмм сообщений от каждого отдельного компонента не составит никакого труда.

Параметру Wnd присваивается значение свойства HWindow глобальной переменной IconMgr типа TIconMgr. Реализация этого объекта приведена ниже в этой главе, но пока важно знать, что все сообщения пиктограмм панели задач отправляются именно через этот компонент.

Параметру uCallbackMessage присваивается значение сообщения DDGM_TRAYICON, определенное посредством вызова функции API RegisterWindowMessage(). Это гарантирует, что сообщение DDGM_TRAYICON будет иметь уникальный идентификатор сообщения в масштабе всей системы. Данная задача решается с помощью следующего фрагмента кода:

```
const
  { Строка для идентификации зарегистрированного сообщения Windows. }
  TrayMsgStr = 'DDG.TrayNotifyIconMsg';

initialization
  { Получение уникального идентификатора Windows-сообщения для
    обратного вызова от индикаторной области панели задач. }
  DDGM_TRAYICON := RegisterWindowMessage(TrayMsgStr);
```

В параметр hIcon помещается значение, возвращаемое методом ActiveIconHandle(). Этот метод возвращает дескриптор пиктограммы, выбранной в данный момент в свойстве Icon рассматриваемого компонента.

Обработка сообщений

Ранее уже упоминалось, что все сообщения индикаторной области панели задач посылаются окну, поддерживаемому глобальным объектом IconMgr. Этот объект создается и освобождается в разделах initialization и finalization модуля нашего компонента, как показано в приведенном ниже фрагменте кода.

```
initialization
  { Получение уникального идентификатора Windows-сообщения для
    обратного вызова индикаторной области панели задач. }
  DDGM_TRAYICON := RegisterWindowMessage(TrayMsgStr);
  IconMgr := TIconManager.Create;
finalization
  IconMgr.Free;
```

Этот объект относительно небольшой. Он определяется следующим образом:

```
type
  TIconManager = class
  private
    FHWindow: HWnd;
    procedure TrayWndProc(var Message: TMessage);
  public
    constructor Create;
    destructor Destroy; override;
    property HWindow: HWnd read FHWindow write FHWindow;
  end;
```

Окно, которому будут отсылаться сообщения индикаторной области, создается в конструкторе этого объекта с помощью функции `AllocateHWnd()`:

```
constructor TIconManager.Create;
begin
  FHWindow := AllocateHWnd(TrayWndProc);
end;
```

Метод `TrayWndProc()` служит оконной процедурой для создаваемого в конструкторе окна. Более подробно данный метод рассматривается ниже в этой главе.

Пиктограммы и подсказки

Наиболее простой путь отображения пиктограмм и подсказок состоит в использовании свойств данного компонента. Кроме того, свойство `Icon` имеет тип `TIcon`, а это означает, что при определении его значения можно воспользоваться преимуществами встроенного в Delphi редактора свойств, предназначенного для пиктограмм. Поскольку пиктограмма индикаторной области панели задач видима даже во время разработки, необходимо убедиться в том, что пиктограмма и подсказка могут динамически изменяться. Эта проверка не потребует особых усилий: нужно лишь убедиться в том, что метод `SendTrayMessage()` вызывается (с помощью сообщения `NIM_MODIFY`) в методе `write` свойств `Hint` и `Icon`.

Ниже приведена реализация методов `write` для этих свойств:

```
procedure TTrayNotifyIcon.SetIcon(Value: TIcon);
{ Метод Write для свойства Icon. }
begin
  FIcon.Assign(Value);  // Устанавливаем новую пиктограмму
  if FIconVisible then
    { Изменяем пиктограмму на панели задач. }
    SendTrayMessage(NIM_MODIFY, NIF_ICON);
```

```
end;

procedure TTrayNotifyIcon.SetHint(Value: String);
{ Метод установки для свойства Hint. }
begin
  if FHint <> Value then
  begin
    FHint := Value;
    if FIconVisible then
      { Изменяем подсказку пиктограммы на панели задач. }
      SendTrayMessage(NIM_MODIFY, NIF_TIP);
  end;
end;
```

Обработка щелчков мышью

При работе с пиктограммами одной из наиболее сложных задач является обеспечение правильной обработки щелчков мышью. Вы, вероятно, уже заметили, что многие пиктограммы индикаторной области панели задач реагируют на щелчки мышью тремя различными способами:

- открывают окно при одинарном щелчке;
- открывают другое окно (обычно окно свойств) при двойном щелчке;
- вызывают локальное меню при щелчке правой кнопкой.

Сложность состоит в создании события, представляющего двойной щелчок и одновременно блокирующего генерацию события одинарного щелчка.

По схеме генерации сообщений Windows, при выполнении пользователем двойного щелчка левой кнопкой мыши обладающее фокусом окно получит сразу два сообщения: WM_LBUTTONDOWN и WM_LBUTTONDBLCLK. Для того чтобы дать возможность программе обработать только двойной щелчок мышью, необходимо предусмотреть механизм задержки обработки сообщения WM_LBUTTONDOWN на период времени, достаточный, чтобы убедиться в
 на период времени, достаточный, чтобы убедиться в отсутствии сообщения о выполнении двойного щелчка.

Совсем нетрудно определить время ожидания, необходимое для того, чтобы убедиться, что сообщение WM_LBUTTONDBLCLK не последует за сообщением WM_LBUTTONDOWN. Для этого используется API-функция GetDoubleClickTime(), не имеющая никаких параметров и возвращающая максимальный промежуток времени (в миллисекундах) между двумя щелчками двойного щелчка мышью, допускаемый панелью управления (Control Panel). Удобным средством для построения механизма ожидания, устанавливаемого на количество миллисекунд, задаваемое функцией GetDoubleClickTime(), является компонент TTimer. Он создается и инициируется в конструкторе компонента TTrayNotifyIcon следующим образом:

```
FTimer := TTimer.Create(Self);
with FTimer do
begin
  Enabled := False;
  Interval := GetDoubleClickTime;
  OnTimer := OnButtonTimer;
end;
```

По истечении интервала ожидания будет вызван метод OnButtonTimer(), который рассматривается ниже в этой главе.

Как уже упоминалось, сообщения индикаторной области панели задач фильтруются с помощью метода TrayWndProc() глобального объекта IconMgr. Рассмотрим текст этого метода.

```
procedure TIconManager.TrayWndProc(var Message: TMessage);
{ Позволяет обрабатывать все "обратные" сообщения индикаторной
  области панели задач в контексте компонента. }
var
  Pt: TPoint;
  TheIcon: TTrayNotifyIcon;
begin
  with Message do
  begin
    { Проверяем, есть ли обратное сообщение от индикаторной
      области панели задач. }
    if (Msg = DDGM_TRAYICON) then
    begin
      TheIcon := TTrayNotifyIcon(WParam);
      case lParam of
        { Запускаем таймер при первом щелчке мышью. Событие OnClick
          будет вызвано методом OnTimer; при этом гарантируется,
          что двойного щелчка не было. }
        WM_LBUTTONDOWN: TheIcon.FTimer.Enabled := True;
        { Устанавливаем флаг двойного щелчка в положение "нет щелчка".
          В этом случае одиночный щелчок будет подавлен. }
        WM_LBUTTONDBLCLK:
          begin
            TheIcon.FNoShowClick := True;
            if Assigned(TheIcon.FOnDblClick) then TheIcon.FOnDblClick(Self);
          end;
        WM_RBUTTONDOWN:
          begin
            if Assigned(TheIcon.FPopupMenu) then
            begin
              { Вызываем функцию SetForegroundWindow, требуемую API. }
              SetForegroundWindow(IconMgr.HWindow);
              { Открываем локальное меню в позиции курсора. }
              GetCursorPos(Pt);
              TheIcon.FPopupMenu.Popup(Pt.X, Pt.Y);
              { Отправляем сообщение, требуемое API для переключения задачи. }
              PostMessage(IconMgr.HWindow, WM_USER, 0, 0);
            end;
          end;
      end;
    end
    else
      { Если сообщение не относится к индикаторной области панели задач,
        вызываем процедуру DefWindowProc. }
      Result := DefWindowProc(FHWindow, Msg, wParam, lParam);
  end;
end;
```

В этом фрагменте кода запрограммированы реакции на сообщения, вызываемые различными событиями: при одинарном щелчке просто запускается таймер, при двойном — перед генерацией события OnDblClick устанавливается флаг, указывающий на то, что произошел двойной щелчок, а при щелчке правой кнопкой мыши вызывается контекстное меню, определяемое свойством PopupMenu. Теперь рассмотрим метод OnButtonTimer(), код которого приведен ниже.

```
procedure TTrayNotifyIcon.OnButtonTimer(Sender: TObject);
begin
  { Блокируем таймер, поскольку его нужно запустить лишь один раз. }
  FTimer.Enabled := False;
  { Если повторного щелчка не произошло, запускаем обработку
    одиночного щелчка. }
  if (not FNoShowClick) and Assigned(FOnClick) then
    FOnClick(Self);
  FNoShowClick := False;    // Сбрасываем флаг
end;
```

В первую очередь этот метод блокирует таймер. Тем самым гарантируется, что щелчок мыши генерирует только одно событие. Затем проверяется состояние флага FNoShowClick. Помните, что этот флаг устанавливается в методе OwnerWndProc() при обработке сообщения, вызванного двойным щелчком мыши. Таким образом, событие OnClick будет сгенерировано только при условии отсутствия события OnDblClk.

Скрытие приложения

Еще одним аспектом приложений, связанных с областью индикации панели задач, является то, что они не отображаются в виде кнопок на панели задач. Чтобы обеспечить приложение подобной функциональностью, в компонент TTrayNotifyIcon введено свойство HideTask, позволяющее пользователю самому решить, отображать или нет приложение на панели задач. Код метода записи для этого свойства приведен ниже. Главную роль в данном коде играет строка, содержащая вызов процедуры Win32 API ShowWindow(), которой передается свойство Handle объекта Application, а также константа, определяющая, будет приложение отображаться на панели задач или нет.

```
procedure TTrayNotifyIcon.SetHideTask(Value: Boolean);
{ Метод Write для свойства HideTask. }
const
  { Флаги, определяющие, будет приложение отображено или скрыто. }
  ShowArray: array[Boolean] of integer = (sw_ShowNormal, sw_Hide);
begin
  if FHideTask <> Value then begin
    FHideTask := Value;
    { Ничего не делаем в режиме разработки. }
    if not (csDesigning in ComponentState) then
      ShowWindow(Application.Handle, ShowArray[FHideTask]);
  end;
end;
```

В листинге 24.1 приведен код модуля TrayIcon.pas, представляющий собой реализацию компонента TTrayNotifyIcon.

Листинг 24.1. Модуль `TrayIcon.pas`: исходный текст компонента `TTrayNotifyIcon`

```
unit TrayIcon;

interface

uses Windows, SysUtils, Messages, ShellAPI, Classes, Graphics,
  Forms, Menus, StdCtrls, ExtCtrls;

type
  ENotifyIconError = class(Exception);

  TTrayNotifyIcon = class(TComponent)
  private
    FDefaultIcon: THandle;
    FIcon: TIcon;
    FHideTask: Boolean;
    FHint: string;
    FIconVisible: Boolean;
    FPopupMenu: TPopupMenu;
    FOnClick: TNotifyEvent;
    FOnDblClick: TNotifyEvent;
    FNoShowClick: Boolean;
    FTimer: TTimer;
    Tnd: TNotifyIconData;
    procedure SetIcon(Value: TIcon);
    procedure SetHideTask(Value: Boolean);
    procedure SetHint(Value: string);
    procedure SetIconVisible(Value: Boolean);
    procedure SetPopupMenu(Value: TPopupMenu);
    procedure SendTrayMessage(Msg: DWORD; Flags: UINT);
    function ActiveIconHandle: THandle;
    procedure OnButtonTimer(Sender: TObject);
  protected
    procedure Loaded; override;
    procedure LoadDefaultIcon; virtual;
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property Icon: TIcon read FIcon write SetIcon;
    property HideTask: Boolean read FHideTask write SetHideTask
 default False;
    property Hint: String read FHint write SetHint;
    property IconVisible: Boolean read FIconVisible write SetIconVisible
      default False;
    property PopupMenu: TPopupMenu read FPopupMenu write SetPopupMenu;
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
```

```
    property OnDblClick: TNotifyEvent read FOnDblClick write FOnDblClick;
  end;

implementation

{ TIconManager }
{ Этот класс создает скрытое окно, которое обрабатывает и направляет
  сообщения пиктограмм индикаторной области панели задач. }
type
  TIconManager = class
  private
    FHWindow: HWnd;
    procedure TrayWndProc(var Message: TMessage);
  public
    constructor Create;
    destructor Destroy; override;
    property HWindow: HWnd read FHWindow write FHWindow;
  end;

var
  IconMgr: TIconManager;
  DDGM_TRAYICON: Integer;

constructor TIconManager.Create;
begin
  FHWindow := AllocateHWnd(TrayWndProc);
end;

destructor TIconManager.Destroy;
begin
  if FHWindow <> 0 then DeallocateHWnd(FHWindow);
  inherited Destroy;
end;

procedure TIconManager.TrayWndProc(var Message: TMessage);
{ Эта процедура позволяет обрабатывать все обратные сообщения
  пиктограмм индикаторной области в контексте компонента. }
var
  Pt: TPoint;
  TheIcon: TTrayNotifyIcon;
begin
  with Message do
  begin
    { Проверяем, это - сообщение пиктограммы индикаторной
       области панели задач? }
    if (Msg = DDGM_TRAYICON) then
    begin
      TheIcon := TTrayNotifyIcon(WParam);
      case lParam of
        { Запускаем таймер при первом щелчке кнопкой мыши. Событие
```

```
          OnClick будет сгенерировано методом OnTimer; при этом
          гарантируется отсутствие двойного щелчка. }
        WM_LBUTTONDOWN: TheIcon.FTimer.Enabled := True;
        { Устанавливаем флаг двойного щелчка в положение "нет щелчка".
          При этом будет подавлен одиночный щелчок. }
        WM_LBUTTONDBLCLK:
          begin
            TheIcon.FNoShowClick := True;
            if Assigned(TheIcon.FOnDblClick) then TheIcon.FOnDblClick(Self);
          end;
        WM_RBUTTONDOWN:
          begin
            if Assigned(TheIcon.FPopupMenu) then
            begin
              { Вызываем функцию SetForegroundWindow, требуемую API. }
              SetForegroundWindow(IconMgr.HWindow);
              { Отображаем локальное меню в позиции курсора. }
              GetCursorPos(Pt);
              TheIcon.FPopupMenu.Popup(Pt.X, Pt.Y);
              { Отсылаем сообщение для переключения задачи. }
              PostMessage(IconMgr.HWindow, WM_USER, 0, 0);
            end;
          end;
      end;
    end
    else
      { Если сообщение не связано с индикаторной областью панели задач,
        вызываем процедуру DefWindowProc. }
      Result := DefWindowProc(FHWindow, Msg, wParam, lParam);
  end;
end;

{ TTrayNotifyIcon }

constructor TTrayNotifyIcon.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FIcon := TIcon.Create;
  FTimer := TTimer.Create(Self);
  with FTimer do
  begin
    Enabled := False;
    Interval := GetDoubleClickTime;
    OnTimer := OnButtonTimer;
  end;
  { Загружаем стандартную пиктограмму окна... }
  LoadDefaultIcon;
end;

destructor TTrayNotifyIcon.Destroy;
```

```
begin
  if FIconVisible then SetIconVisible(False);    // Удаляем пиктограмму
  FIcon.Free;                                    // Освобождаем ресурсы
  FTimer.Free;
  inherited Destroy;
end;

function TTrayNotifyIcon.ActiveIconHandle: THandle;
{ Возвращаем дескриптор активной пиктограммы. }
begin
  { Если ни одна из пиктограмм не загружена,
    возвращаем стандартную пиктограмму. }
  if (FIcon.Handle <> 0) then
    Result := FIcon.Handle
  else
    Result := FDefaultIcon;
end;

procedure TTrayNotifyIcon.LoadDefaultIcon;
{ Загружает стандартную пиктограмму окна, которая всегда "под рукой".
  Это позволит компоненту использовать логотип Windows в качестве стандартной
  пиктограммы, если в свойстве Icon не выбрана ни одна из пиктограмм. }
begin
  FDefaultIcon := LoadIcon(0, IDI_WINLOGO);
end;

procedure TTrayNotifyIcon.Loaded;
{ Вызывается после загрузки компонента из потока. }
begin
  inherited Loaded;
  { Если предполагается, что пиктограмма видима, создаем ее. }
  if FIconVisible then
    SendTrayMessage(NIM_ADD, NIF_MESSAGE or NIF_ICON or NIF_TIP);
end;

procedure TTrayNotifyIcon.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and (AComponent = PopupMenu) then
    PopupMenu := nil;
end;

procedure TTrayNotifyIcon.OnButtonTimer(Sender: TObject);
{ Таймер используется для отслеживания времени между двумя щелчками
  двойного щелчка. Реакция на первый щелчок задерживается на время,
  достаточное для того, чтобы быть уверенным в отсутствии второго щелчка.
  Смысл всех этих манипуляций состоит лишь в одном: обеспечить
  независимость событий OnClicks и OnDblClicks. }
begin
```

```
  { Блокируем таймер, поскольку он должен запускаться лишь один раз. }
  FTimer.Enabled := False;
  { Если повторного щелчка не произошло, генерируется событие
    одиночного щелчка. }
  if (not FNoShowClick) and Assigned(FOnClick) then
    FOnClick(Self);
  FNoShowClick := False;        // Сброс флага
end;

procedure TTrayNotifyIcon.SendTrayMessage(Msg: DWORD; Flags: UINT);
{ Этот метод содержит вызов API-функции Shell_NotifyIcon. }
begin
  { Заполняем поля записи соответствующими значениями. }
  with Tnd do
  begin
    cbSize := SizeOf(Tnd);
    StrPLCopy(szTip, PChar(FHint), SizeOf(szTip));
    uFlags := Flags;
    uID := UINT(Self);
    Wnd := IconMgr.HWindow;
    uCallbackMessage := DDGM_TRAYICON;
    hIcon := ActiveIconHandle;
  end;
  Shell_NotifyIcon(Msg, @Tnd);
end;

procedure TTrayNotifyIcon.SetHideTask(Value: Boolean);
{ Метод write для свойства HideTask. }
const
  { Флаги, определяющие, будет приложение отображено или скрыто. }
  ShowArray: array[Boolean] of integer = (sw_ShowNormal, sw_Hide);
begin
  if FHideTask <> Value then
  begin
    FHideTask := Value;
    { Ничего не делаем в режиме разработки. }
    if not (csDesigning in ComponentState) then
      ShowWindow(Application.Handle, ShowArray[FHideTask]);
  end;
end;

procedure TTrayNotifyIcon.SetHint(Value: string);
{ Метод записи для свойства Hint. }
begin
  if FHint <> Value then
  begin
    FHint := Value;
    if FIconVisible then
      { Изменяем подсказку пиктограммы в индикаторной области панели задач. }
      SendTrayMessage(NIM_MODIFY, NIF_TIP);
```

```
  end;
end;

procedure TTrayNotifyIcon.SetIcon(Value: TIcon);
{ Метод записи для свойства Icon. }
begin
  FIcon.Assign(Value);    // Определить новую пиктограмму
  { Изменяем пиктограмму в индикаторной области панели задач }
  if FIconVisible then SendTrayMessage(NIM_MODIFY, NIF_ICON);
end;

procedure TTrayNotifyIcon.SetIconVisible(Value: Boolean);
{ Метод записи для свойства IconVisible }
const
  { Флаги, определяющие добавление или удаление пиктограммы с панели задач }
  MsgArray: array[Boolean] of DWORD = (NIM_DELETE, NIM_ADD);
begin
  if FIconVisible <> Value then
  begin
    FIconVisible := Value;
    { Устанавливаем пиктограмму. }
    SendTrayMessage(MsgArray[Value], NIF_MESSAGE or NIF_ICON or NIF_TIP);
  end;
end;

procedure TTrayNotifyIcon.SetPopupMenu(Value: TPopupMenu);
{ Метод записи для свойства PopupMenu. }
begin
  FPopupMenu := Value;
  if Value <> nil then Value.FreeNotification(Self);
end;

const
  { Строка, идентифицирующая зарегистрированное сообщение Windows. }
  TrayMsgStr = 'DDG.TrayNotifyIconMsg';

initialization
  { Получаем уникальный идентификатор Windows-сообщения для
    обратного вызова индикаторной области панели задач. }
  DDGM_TRAYICON := RegisterWindowMessage(TrayMsgStr);
  IconMgr := TIconManager.Create;
finalization
  IconMgr.Free;
end.
```

На рис. 24.2 показан внешний вид пиктограммы, созданной в индикаторной области панели задач компонентом `TTrayNotifyIcon`.

Рис. 24.2. Пиктограмма, созданная компонентом `TTrayNotifyIcon`

Кстати, так как пиктограмма инициализируется внутри конструктора компонента, а конструкторы выполняются и во время разработки, компонент отображает предназначенную для индикаторной области пиктограмму даже во время разработки приложения.

Пример приложения

Для более близкого знакомства с компонентом TTrayNotifyIcon рассмотрим его работу в контексте приложения. На рис. 24.3 изображено главное окно, а в листинге 24.2 приведен код главного модуля этого приложения.

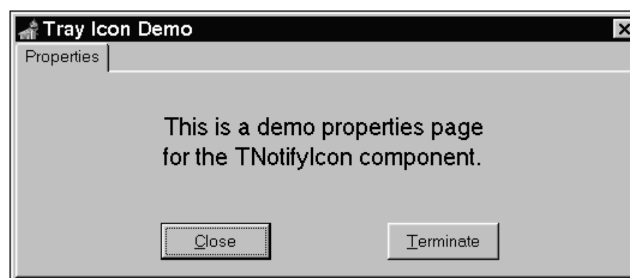


Рис. 24.3. Приложение, демонстрирующее использование компонента TTrayNotifyIcon

Листинг 24.2. Модуль Main.pas — главный модуль демонстрационного приложения

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ShellAPI, TrayIcon, Menus, ComCtrls;

type
  TMainForm = class(TForm)
    pmiPopup: TPopupMenu;
    pgclPageCtl: TPageControl;
    TabSheet1: TTabSheet;
    btnClose: TButton;
    btnTerm: TButton;
    Terminate1: TMenuItem;
    Label1: TLabel;
    N1: TMenuItem;
    Propeties1: TMenuItem;
    TrayNotifyIcon1: TTrayNotifyIcon;
    procedure NotifyIcon1Click(Sender: TObject);
    procedure NotifyIcon1DblClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure btnTermClick(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
```

```
    procedure FormCreate(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.NotifyIcon1Click(Sender: TObject);
begin
  ShowMessage('Single click'); // Одиночный щелчок
end;

procedure TMainForm.NotifyIcon1DblClick(Sender: TObject);
begin
  Show;
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caNone;
  Hide;
end;

procedure TMainForm.btnTermClick(Sender: TObject);
begin
  Application.Terminate;
end;

procedure TMainForm.btnCloseClick(Sender: TObject);
begin
  Hide;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  TrayNotifyIcon1.IconVisible := True;
end;

end.
```

Панели инструментов рабочего стола Windows

Панели инструментов рабочего стола приложений Windows (Application Desktop Toolbar — AppBar) представляют собой окна рабочего стола, которые могут прикрепляться к одной из границ экрана. Хотя, возможно, вы и не знакомы с этим термином, но с объектами,

определяемыми этим понятием, вы встречаетесь каждый раз, работая с компьютером. Пример окна AppBar — панель задач Windows. Как показано на рис. 24.4, панель задач на самом деле даже несколько больше, чем просто окно AppBar, поскольку включает кнопку Start, индикаторную область с пиктограммами и другие элементы управления.

Рис. 24.4. Панель задач оболочки Windows

Помимо свойства прикрепляться к границам экрана, окна типа AppBar могут проявлять и ряд других свойств, присущих панели задач (например, автоматическое сокрытие или возможность перетаскивания с помощью мыши). Однако, что вас по-настоящему удивит, так это размеры API-интерфейса окон данного типа — всего лишь одна функция! Следствием подобной компактности, естественно, является ограниченность реализуемых интерфейсом возможностей. Роль интерфейса здесь скорее консультативная, нежели функциональная, т.е. вместо того чтобы управлять окном AppBar с помощью команд типа "сделай это" или "сделай то", окну AppBar адресуются запросы типа "можно ли сделать это?" и "можно ли сделать то?".

Интерфейс API

Подобно пиктограммам индикаторной области панели задач, для панелей инструментов рабочего стола предусмотрена только одна функция Win32 API — `SHAppBarMessage()`. Она определяется в модуле `ShellAPI` следующим образом:

```
function SHAppBarMessage(dwMessage: DWORD; var pData: TAppBarData): UINT;
  stdcall;
```

Первый параметр `dwMessage` может принимать одно из значений, приведенных в табл. 24.3.

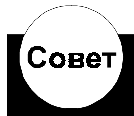
Таблица 24.3. Сообщения окна AppBar

Константа	Значение	Назначение
ABM_NEW	$0	Регистрирует новое окно AppBar и определяет новое "обратное" сообщение
ABM_REMOVE	$1	Отменяет регистрацию существующего окна AppBar
ABM_QUERYPOS	$2	Запрашивает новую позицию и размер окна AppBar
ABM_SETPOS	$3	Устанавливает новую позицию и размер окна AppBar
ABM_GETSTATE	$4	Считывает состояния "автосокрытие" и "расположить поверх всех" панели инструментов оболочки
ABM_GETTASKBARPOS	$5	Считывает позицию панели инструментов оболочки
ABM_ACTIVATE	$6	Уведомляет Windows, что создано новое окно AppBar
ABM_GETAUTOHIDEBAR	$7	Получает дескриптор скрытого окна AppBar, прикрепленного к одной из границ экрана
ABM_SETAUTOHIDEBAR	$8	Регистрирует скрытое окно AppBar на определенной границе экрана
ABM_WINDOWPOSCHANGED	$9	Информирует Windows об изменении позиции окна AppBar

Параметр pData функции SHAppBarMessage() представляет собой запись типа TAppBarData, которая определена в модуле ShellAPI следующим образом:

```
type
  PAppBarData = ^TAppBarData;
  TAppBarData = record
    cbSize: DWORD;
    hWnd: HWND;
    uCallbackMessage: UINT;
    uEdge: UINT;
    rc: TRect;
    lParam: LPARAM; { Информация, связанная с сообщением }
  end;
```

В этой записи в поле cbSize хранится размер записи; в поле hWnd — дескриптор заданного окна AppBar; в поле uCallbackMessage — значение сообщения, которое будет послано окну AppBar вместе с уведомляющими сообщениями; в поле rc — ограничивающий прямоугольник опрашиваемого окна AppBar; в поле lParam — дополнительная информация, специфичная для данного сообщения.



Более подробную информацию о функции Win32 API SHAppBarMessage() и о записи типа TAppBarData можно найти в интерактивной справочной системе Win32.

Компонент TAppBar: форма окна AppBar

Учитывая небольшие размеры API-функции окна AppBar, ее нетрудно встроить в VCL-форму. В этом разделе речь пойдет о помещении окна AppBar в элемент управления, производный от компонента TCustomForm. Поскольку компонент TCustomForm является формой, мы будем работать с таким элементом управления как с формой верхнего уровня в окне конструктора форм (Form Designer), а не как с отдельным компонентом в обычной форме.

Большая часть работы в окне AppBar выполняется путем посылки оболочке Windows записи TAppBarData, что осуществляется с помощью вызова функции API SHAppBarMessage(). Компонент TAppBar поддерживает внутреннюю запись TAppBarData с именем FABD. В конструкторе компонента и методе CreateWnd() запись FABD настраивается на вызов функции SendAppBarMsg() с целью создания окна AppBar. В частности, здесь поле cbSize инициализируется, поле uCallbackMessage получает значение, возвращаемое функцией API RegisterWindowMessage(), а поле hWnd получает дескриптор текущего окна формы. Функция SendAppBarMessage() инкапсулирует функцию SHAppBarMessage() и определяется следующим образом:

```
function TAppBar.SendAppBarMsg(Msg: DWORD): UINT;
begin
  Result := SHAppBarMessage(Msg, FABD);
end;
```

При успешном создании окна AppBar вызывается метод SetAppBarEdge(), используемый для помещения окна AppBar в начальное положение. Этот метод, в свою очередь, вызывает метод SetAppBarPos(), передавая флаг, указывающий на требуемую границу экрана, у кото-

рой должно разместиться окно AppBar. Флаги ABE_TOP, ABE_BOTTOM, ABE_LEFT и ABE_RIGHT представляют соответственно верхнюю, нижнюю, левую и правую границы экрана. Реализация этого метода показана в приведенном ниже фрагменте кода:

```
procedure TAppBar.SetAppBarPos(Edge: UINT);
begin
  if csDesigning in ComponentState then Exit;
  FABD.uEdge := Edge;        // Устанавливаем границу
  with FABD.rc do
  begin
    // Устанавливаем координаты полного экрана
    Top := 0;
    Left := 0;
    Right := Screen.Width;
    Bottom := Screen.Height;
    { Посылаем сообщение ABM_QUERYPOS для получения размеров
      соответствующего прямоугольника у границы экрана }
    SendAppBarMsg(ABM_QUERYPOS);
    { Перестраиваем прямоугольник на основе данных, полученных
      в результате отправки сообщения ABM_QUERYPOS }
    case Edge of
      ABE_LEFT: Right := Left + FDockedWidth;
      ABE_RIGHT: Left := Right - FDockedWidth;
      ABE_TOP: Bottom := Top + FDockedHeight;
      ABE_BOTTOM: Top := Bottom - FDockedHeight;
    end;
    // Устанавливаем позицию окна AppBar
    SendAppBarMsg(ABM_SETPOS);
  end;
  // Согласовываем свойство BoundsRect с
  // ограничивающим прямоугольником, передаваемым системе
  BoundsRect := FABD.rc;
end;
```

Сначала в этом методе поле uEdge записи FABD устанавливается равным значению параметра Edge. Затем поле rc заполняется значениями координат полного экрана и посылается сообщение ABM_QUERYPOS. Это сообщение переустанавливает размеры прямоугольника, определяемого полем rc, которое теперь содержит координаты прямоугольника, состыкованного с краем, определяемым полем uEdge. После этого поле rc перестраивается еще раз — до нужных размеров подстраиваются высота и ширина. Теперь поле rc содержит окончательный размер ограничивающей рамки для окна AppBar. Затем оболочке Windows посылается сообщение ABM_SETPOS, уведомляющее о создании нового прямоугольника, и, наконец, сам прямоугольник устанавливается с помощью свойства BoundsRect элемента управления.

Выше в этой главе уже упоминалось, что уведомляющие сообщения будут посылаться окну, определяемому значением FABD.hWnd, с использованием идентификатора сообщения, содержащегося в поле FABD.uCallbackMessage. Эти уведомляющие сообщения обрабатываются методом WndProc(), текст которого приведен ниже.

```
procedure TAppBar.WndProc(var M: TMessage);
var
  State: UINT;
  WndPos: HWnd;
begin
  if M.Msg = AppBarMsg then
  begin
    case M.WParam of
    {Отправляем, если изменилось состояние “всегда сверху” или “автосокрытие”.}
      ABN_STATECHANGE:
        begin
          { Проверяем состояние флага "всегда сверху" ABS_ALWAYSONTOP. }
          State := SendAppBarMsg(ABM_GETSTATE);
          if ABS_ALWAYSONTOP and State = 0 then
            SetTopMost(False)
          else
            SetTopMost(True);
        end;
      { Запущено (или закрыто последнее) полноэкранное приложение.}
      ABN_FULLSCREENAPP:
        begin
          { Устанавливаем соответствующий Z-порядок окна AppBar. }
          State := SendAppBarMsg(ABM_GETSTATE);
          if M.lParam <> 0 then begin
            if ABS_ALWAYSONTOP and State = 0 then
              SetTopMost(False)
            else
              SetTopMost(True);
        end
        else
          if State and ABS_ALWAYSONTOP <> 0 then
            SetTopMost(True);
      end;
    { Посылается при любом изменении положения окна AppBar. }
    ABN_POSCHANGED:
      begin
        { Панель задач или какая-нибудь другая панель изменили
          размеры и расположение. }
        SetAppBarPos(FABD.uEdge);
      end;
    end;
  end
  else
    inherited WndProc(M);
end;
```

Этот метод обрабатывает ряд уведомляющих сообщений, позволяющих окну AppBar реагировать на изменения, происходящие в оболочке во время работы приложения. Остальная часть кода компонента TAppBar приведена в листинге 24.3.

Листинг 24.3. Модуль `AppBars.pas`, содержащий базовый класс для поддержки окна AppBar

```
unit AppBars;

interface

uses Windows, Messages, SysUtils, Forms, ShellAPI, Classes, Controls;

type
  TAppBarEdge = (abeTop, abeBottom, abeLeft, abeRight);

  EAppBarError = class(Exception);

  TAppBar = class(TCustomForm)
  private
    FABD: TAppBarData;
    FDockedHeight: Integer;
    FDockedWidth: Integer;
    FEdge: TAppBarEdge;
    FOnEdgeChanged: TNotifyEvent;
    FTopMost: Boolean;
    procedure WMActivate(var M: TMessage); message WM_ACTIVATE;
    procedure WMWindowPosChanged(var M: TMessage); message WM_WINDOWPOSCHANGED;
    function SendAppBarMsg(Msg: DWORD): UINT;
    procedure SetAppBarEdge(Value: TAppBarEdge);
    procedure SetAppBarPos(Edge: UINT);
    procedure SetTopMost(Value: Boolean);
    procedure SetDockedHeight(const Value: Integer);
    procedure SetDockedWidth(const Value: Integer);
  protected
    procedure CreateParams(var Params: TCreateParams); override;
    procedure CreateWnd; override;
    procedure DestroyWnd; override;
    procedure WndProc(var M: TMessage); override;
  public
    constructor CreateNew(AOwner: TComponent; Dummy: Integer = 0); override;
    property DockManager;
  published
    property Action;
    property ActiveControl;
    property AutoScroll;
    property AutoSize;
    property BiDiMode;
    property BorderWidth;
    property Color;
    property Ctl3D;
    property DockedHeight: Integer read FDockedHeight write SetDockedHeight
    default 35;
    property DockedWidth: Integer read FDockedWidth write SetDockedWidth
```

```
    default 40;
  property UseDockManager;
  property DockSite;
  property DragKind;
  property DragMode;
  property Edge: TAppBarEdge read FEdge write SetAppBarEdge default abeTop;
  property Enabled;
  property ParentFont default False;
  property Font;
  property HelpFile;
  property HorzScrollBar;
  property Icon;
  property KeyPreview;
  property ObjectMenuItem;
  property ParentBiDiMode;
  property PixelsPerInch;
  property PopupMenu;
  property PrintScale;
  property Scaled;
  property ShowHint;
  property TopMost: Boolean read FTopMost write SetTopMost default False;
  property VertScrollBar;
  property Visible;
  property OnActivate;
  property OnCanResize;
  property OnClick;
  property OnClose;
  property OnCloseQuery;
  property OnConstrainedResize;
  property OnCreate;
  property OnDblClick;
  property OnDestroy;
  property OnDeactivate;
  property OnDockDrop;
  property OnDockOver;
  property OnDragDrop;
  property OnDragOver;
  property OnEdgeChanged: TNotifyEvent read FOnEdgeChanged
    write FOnEdgeChanged;
  property OnEndDock;
  property OnGetSiteInfo;
  property OnHide;
  property OnHelp;
  property OnKeyDown;
  property OnKeyPress;
  property OnKeyUp;
  property OnMouseDown;
  property OnMouseMove;
  property OnMouseUp;
  property OnMouseWheel;
```

```
    property OnMouseWheelDown;
    property OnMouseWheelUp;
    property OnPaint;
    property OnResize;
    property OnShortCut;
    property OnShow;
    property OnStartDock;
    property OnUnDock;
  end;

implementation

var
  AppBarMsg: UINT;

constructor TAppBar.CreateNew(AOwner: TComponent; Dummy: Integer);
begin
  FDockedHeight := 35;
  FDockedWidth := 40;
  inherited CreateNew(AOwner, Dummy);
  ClientHeight := 35;
  Width := 100;
  BorderStyle := bsNone;
  BorderIcons := [];
  // Инициализируется запись TAppBarData
  FABD.cbSize := SizeOf(FABD);
  FABD.uCallbackMessage := AppBarMsg;
end;

procedure TAppBar.WMWindowPosChanged(var M: TMessage);
begin
  inherited;
  { Нужно проинформировать оболочку об изменении положения окна AppBar. }
  SendAppBarMsg(ABM_WINDOWPOSCHANGED);
end;

procedure TAppBar.WMActivate(var M: TMessage);
begin
  inherited;
  { Нужно проинформировать оболочку о том, что окно AppBar активизировано. }
  SendAppBarMsg(ABM_ACTIVATE);
end;

procedure TAppBar.WndProc(var M: TMessage);
var
  State: UINT;
begin
  if M.Msg = AppBarMsg then
  begin
    case M.WParam of
```

```
      { Посылаем сообщение, если изменилось состояние "всегда сверху"
        или автоматическое сокрытие.}
      ABN_STATECHANGE:
        begin
          { Проверяем состояние флага "всегда сверху" ABS_ALWAYSONTOP.}
          State := SendAppBarMsg(ABM_GETSTATE);
          if ABS_ALWAYSONTOP and State = 0 then
            SetTopMost(False)
          else
            SetTopMost(True);
        end;
      { Запущено (или закрыто последнее) полноэкранное приложение.}
      ABN_FULLSCREENAPP:
        begin
          { Устанавливаем соответствующий Z-порядок окна. }
          State := SendAppBarMsg(ABM_GETSTATE);
          if M.lParam <> 0 then begin
            if ABS_ALWAYSONTOP and State = 0 then
              SetTopMost(False)
            else
              SetTopMost(True);
         end
         else
           if State and ABS_ALWAYSONTOP <> 0 then
             SetTopMost(True);
        end;
      // Посылается при любом изменении положения окна AppBar
      ABN_POSCHANGED:
        { Панель задач или какая-нибудь другая панель изменили
          свои размеры и положение. }
        SetAppBarPos(FABD.uEdge);
    end;
  end
  else
    inherited WndProc(M);
end;

function TAppBar.SendAppBarMsg(Msg: DWORD): UINT;
begin
  { Не создавать AppBar во время разработки. }
  if csDesigning in ComponentState then Result := 0
  else Result := SHAppBarMessage(Msg, FABD);
end;

procedure TAppBar.SetAppBarPos(Edge: UINT);
begin
  if csDesigning in ComponentState then Exit;
  FABD.uEdge := Edge;         // Устанавливаем границу экрана
  with FABD.rc do
  begin
```

```
    // Устанавливаем координаты в соответствии с размерами полного экрана
    Top := 0;
    Left := 0;
    Right := Screen.Width;
    Bottom := Screen.Height;
    { Посылаем сообщение ABM_QUERYPOS для получения прямоугольника
      соответствующих размеров у края экрана. }
    SendAppBarMsg(ABM_QUERYPOS);
    { Перестраиваем прямоугольник в соответствии с информацией,
      полученной в ответ на сообщение ABM_QUERYPOS. }
    case Edge of
      ABE_LEFT: Right := Left + FDockedWidth;
      ABE_RIGHT: Left := Right - FDockedWidth;
      ABE_TOP: Bottom := Top + FDockedHeight;
      ABE_BOTTOM: Top := Bottom - FDockedHeight;
    end;
    // Устанавливаем позицию панели
    SendAppBarMsg(ABM_SETPOS);
  end;
  // Согласовываем свойство BoundsRect с
  // ограничивающим прямоугольником, переданным системе
  BoundsRect := FABD.rc;
end;

procedure TAppBar.SetTopMost(Value: Boolean);
const
  WndPosArray: array[Boolean] of HWND = (HWND_BOTTOM, HWND_TOPMOST);
begin
  if FTopMost <> Value then
  begin
    FTopMost := Value;
    if not (csDesigning in ComponentState) then
      SetWindowPos(Handle, WndPosArray[Value], 0, 0, 0, 0, SWP_NOMOVE or
        SWP_NOSIZE or SWP_NOACTIVATE);
  end;
end;

procedure TAppBar.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params);
  if not (csDesigning in ComponentState) then
  begin
    Params.ExStyle := Params.ExStyle or WS_EX_TOPMOST or WS_EX_WINDOWEDGE;
    Params.Style := Params.Style or WS_DLGFRAME;
  end;
end;

procedure TAppBar.CreateWnd;
begin
  inherited CreateWnd;
```

```
  FABD.hWnd := Handle;
  if not (csDesigning in ComponentState) then
  begin
    if SendAppBarMsg(ABM_NEW) = 0 then
      raise EAppBarError.Create('Failed to create AppBar');
    // Инициализируем позицию
    SetAppBarEdge(FEdge);
  end;
end;

procedure TAppBar.DestroyWnd;
begin
  // Нужно проинформировать оболочку о том, что панель AppBar удаляется
  SendAppBarMsg(ABM_REMOVE);
  inherited DestroyWnd;
end;

procedure TAppBar.SetAppBarEdge(Value: TAppBarEdge);
const
  EdgeArray: array[TAppBarEdge] of UINT =
    (ABE_TOP, ABE_BOTTOM, ABE_LEFT, ABE_RIGHT);
begin
  SetAppBarPos(EdgeArray[Value]);
  FEdge := Value;
  if Assigned(FOnEdgeChanged) then FOnEdgeChanged(Self);
end;

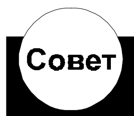
procedure TAppBar.SetDockedHeight(const Value: Integer);
begin
  if FDockedHeight <> Value then
  begin
    FDockedHeight := Value;
    SetAppBarEdge(FEdge);
  end;
end;

procedure TAppBar.SetDockedWidth(const Value: Integer);
begin
  if FDockedWidth <> Value then
  begin
    FDockedWidth := Value;
    SetAppBarEdge(FEdge);
  end;
end;

initialization
  AppBarMsg := RegisterWindowMessage('DDG AppBar Message');
end.
```

Использование компонента TAppBar

Если вы установили программный продукт с компакт-диска, прилагаемого к этой книге, использование компонента TAppBar не вызовет никаких трудностей: достаточно выбрать опцию AppBar во вкладке DDG диалогового окна File⇨New dialog — и вызванный мастер немедленно сгенерирует модуль, содержащий компонент TAppBar.



В главе 26, “Использование интерфейса Open Tools API”, будет показано, как создать мастер, который автоматически генерирует компонент TAppBar. Для задач, решаемых в этой главе, можно вполне обойтись и без рассмотрения реализации мастера. Просто примите сейчас к сведению то, что часть работы по созданию формы и соответствующего модуля выполняется “за кулисами”.

Ниже приведен код небольшого приложения, использующего компонент TAppBar для создания панели инструментов на рабочем столе приложения с кнопками для выполнения различных команд редактирования и работы с файлами: Open, Save, Cut, Copy и Paste. С помощью этих кнопок можно управлять компонентом TMemo, расположенным в главной форме. Код главного модуля приложения приведен в листинге 24.4, а на рис. 24.5 это приложение показано в действии — создана панель AppBar, которая прикреплена к верхней границе экрана.

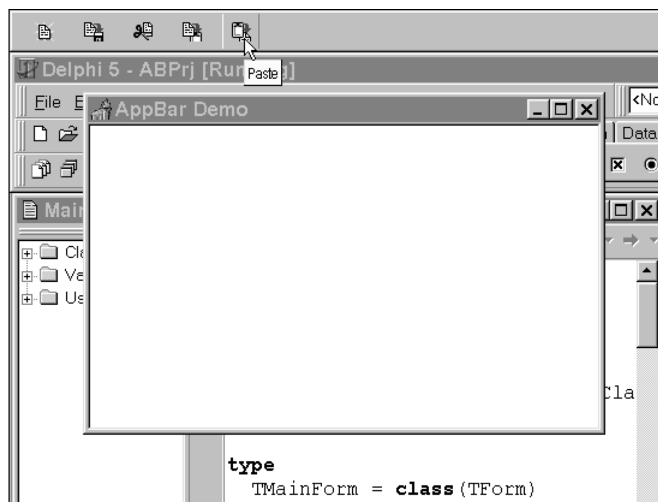


Рис. 24.5. Компонент TAppBar *в действии*

Листинг 24.4. Модуль ApBarFrm.pas — главный модуль приложения, демонстрирующего использование окна AppBar

```
unit ApBarFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, AppBars, Menus, Buttons;
```

```
type
  TAppBarForm = class(TAppBar)
    sbOpen: TSpeedButton;
    sbSave: TSpeedButton;
    sbCut: TSpeedButton;
    sbCopy: TSpeedButton;
    sbPaste: TSpeedButton;
    OpenDialog: TOpenDialog;
    pmPopup: TPopupMenu;
    Top1: TMenuItem;
    Bottom1: TMenuItem;
    Left1: TMenuItem;
    Right1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    procedure Right1Click(Sender: TObject);
    procedure sbOpenClick(Sender: TObject);
    procedure sbSaveClick(Sender: TObject);
    procedure sbCutClick(Sender: TObject);
    procedure sbCopyClick(Sender: TObject);
    procedure sbPasteClick(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormEdgeChanged(Sender: TObject);
  private
    FLastChecked: TMenuItem;
    procedure MoveButtons;
  end;

var
  AppBarForm: TAppBarForm;

implementation

uses Main;

{$R *.DFM}

{ TAppBarForm }
procedure TAppBarForm.MoveButtons;
{ Этот метод выглядит сложным, но на самом деле он просто размещает
  кнопки в зависимости от того, к какой стороне экрана прикреплена
  панель AppBar.}
var
  DeltaCenter, NewPos: Integer;
begin
  if Edge in [abeTop, abeBottom] then
  begin
    DeltaCenter := (ClientHeight - sbOpen.Height) div 2;
    sbOpen.SetBounds(10, DeltaCenter, sbOpen.Width, sbOpen.Height);
    NewPos := sbOpen.Width + 20;
```

```
    sbSave.SetBounds(NewPos, DeltaCenter, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Width + 10;
    sbCut.SetBounds(NewPos, DeltaCenter, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Width + 10;
    sbCopy.SetBounds(NewPos, DeltaCenter, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Width + 10;
    sbPaste.SetBounds(NewPos, DeltaCenter, sbOpen.Width, sbOpen.Height);
  end
  else
  begin
    DeltaCenter := (ClientWidth - sbOpen.Width) div 2;
    sbOpen.SetBounds(DeltaCenter, 10, sbOpen.Width, sbOpen.Height);
    NewPos := sbOpen.Height + 20;
    sbSave.SetBounds(DeltaCenter, NewPos, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Height + 10;
    sbCut.SetBounds(DeltaCenter, NewPos, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Height + 10;
    sbCopy.SetBounds(DeltaCenter, NewPos, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Height + 10;
    sbPaste.SetBounds(DeltaCenter, NewPos, sbOpen.Width, sbOpen.Height);
  end;
end;

procedure TAppBarForm.Right1Click(Sender: TObject);
begin
  FLastChecked.Checked := False;
  (Sender as TMenuItem).Checked := True;
  case TMenuItem(Sender).Caption[2] of
    'T': Edge := abeTop;
    'B': Edge := abeBottom;
    'L': Edge := abeLeft;
    'R': Edge := abeRight;
  end;
  FLastChecked := TMenuItem(Sender);
end;

procedure TAppBarForm.sbOpenClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    MainForm.FileName := OpenDialog.FileName;
end;

procedure TAppBarForm.sbSaveClick(Sender: TObject);
begin
  MainForm.memEditor.Lines.SaveToFile(MainForm.FileName);
end;

procedure TAppBarForm.sbCutClick(Sender: TObject);
begin
  MainForm.memEditor.CutToClipboard;
```

```
end;
procedure TAppBarForm.sbCopyClick(Sender: TObject);
begin
  MainForm.memEditor.CopyToClipboard;
end;

procedure TAppBarForm.sbPasteClick(Sender: TObject);
begin
  MainForm.memEditor.PasteFromClipboard;
end;

procedure TAppBarForm.Exit1Click(Sender: TObject);
begin
  Application.Terminate;
end;

procedure TAppBarForm.FormCreate(Sender: TObject);
begin
  FLastChecked := Top1;
end;

procedure TAppBarForm.FormEdgeChanged(Sender: TObject);
begin
  MoveButtons;
end;

end.
```

Ярлыки Windows

В оболочке Windows предусмотрен ряд интерфейсов, которые можно использовать для управления различными ее свойствами. Эти интерфейсы определены в модуле ShlObj. Развернутого обсуждения всех объектов этого модуля хватило бы на целую книгу, но в данном разделе мы остановимся лишь на одном из самых полезных (и чаще всего используемых) — интерфейсе IShellLink.

Интерфейс IShellLink позволяет создавать в приложениях ярлыки Windows и управлять ими. С такими объектами вы постоянно встречаетесь при работе на компьютере: большинство пиктограмм на рабочем столе — это именно ярлыки. Кроме того, такие команды меню, как Send To (Отправить) или Documents (Документы) — это тоже ярлыки. Интерфейс IShellLink определяется следующим образом:

```
const

type
  IShellLink = interface(IUnknown)
    ['{000214EE-0000-0000-C000-000000000046}']
    function GetPath(pszFile: PAnsiChar; cchMaxPath: Integer;
      var pfd: TWin32FindData; fFlags: DWORD): HResult; stdcall;
    function GetIDList(var ppidl: PItemIDList): HResult; stdcall;
```

```
    function SetIDList(pidl: PItemIDList): HResult; stdcall;
    function GetDescription(pszName: PAnsiChar; cchMaxName: Integer):
       HResult; stdcall;
    function SetDescription(pszName: PAnsiChar): HResult; stdcall;
    function GetWorkingDirectory(pszDir: PAnsiChar; cchMaxPath: Integer):
      HResult; stdcall;
    function SetWorkingDirectory(pszDir: PAnsiChar): HResult; stdcall;
    function GetArguments(pszArgs: PAnsiChar; cchMaxPath: Integer):
      HResult; stdcall;
    function SetArguments(pszArgs: PAnsiChar): HResult; stdcall;
    function GetHotkey(var pwHotkey: Word): HResult; stdcall;
    function SetHotkey(wHotkey: Word): HResult; stdcall;
    function GetShowCmd(out piShowCmd: Integer): HResult; stdcall;
    function SetShowCmd(iShowCmd: Integer): HResult; stdcall;
    function GetIconLocation(pszIconPath: PAnsiChar; cchIconPath: Integer;
      out piIcon: Integer): HResult; stdcall;
    function SetIconLocation(pszIconPath: PAnsiChar; iIcon: Integer):
      HResult; stdcall;
    function SetRelativePath(pszPathRel: PAnsiChar; dwReserved: DWORD):
      HResult; stdcall;
    function Resolve(Wnd: HWND; fFlags: DWORD): HResult; stdcall;
    function SetPath(pszFile: PAnsiChar): HResult; stdcall;
    end;
```

На заметку

Интерфейс IShellLink и все его методы подробно описаны в интерактивной справочной системе Win32, и поэтому в данной книге эта информация опущена.

Создание экземпляра интерфейса IShellLink

В отличие от расширений оболочки, с которыми вы познакомитесь ниже в этой главе, при работе с интерфейсом IShellLink вам не нужно реализовывать интерфейс — он уже реализован в самой оболочке Windows, и остается лишь создать его экземпляр. Для этого служит COM-функция CoCreateInstance(). Предлагаем рассмотреть пример создания подобного экземпляра.

```
var
  SL: IShellLink;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
  // Здесь используется экземпляр интерфейса SL
end;
```

На заметку

Не забудьте, что, прежде чем использовать какую-либо из функций OLE, необходимо инициализировать COM-библиотеку с помощью функции CoInitialize(). По окончании работы для освобождения ресурсов нужно вызвать функцию CoUninitialize(). Если ваше приложение использует модуль ComObj и содержит вызов функции Application.Initialize(), то упомянутые функции будут вызваны автоматически. В противном случае вам придется вызывать эти функции самостоятельно.

Использование интерфейса IShellLink

Иногда кажется, что ярлыки Windows обладают просто магическими возможностями: щелкаем правой кнопкой мыши на поверхности рабочего стола, выбираем команду создания нового ярлыка, и после этого происходит *нечто*, сопровождающееся появлением пиктограммы на поверхности рабочего стола. Но стоит узнать, что происходит в действительности, как от загадочности не остается и следа. *Ярлык* на самом деле представляет собой файл с расширением .lnk, расположенный в определенной папке. При запуске Windows в известных папках выполняется поиск .lnk-файлов. К числу этих специальных папок (папок системной оболочки) относятся такие, как Network Neighborhood, Send To, Startup, Desktop и т.д. Windows хранит отношения "ярлык–папка" в системном реестре — главным образом в следующей ветви:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer
➥\Shell Folders).
```

Для того чтобы создать в некоторой папке ярлык, достаточно просто поместить в эту папку .lnk-файл. Для получения путей к специальным папкам можно, конечно, обратиться к системному реестру, однако гораздо эффективнее использовать функцию Win32 API SHGetSpecialFolderPath(), которая определяется следующим образом:

```
function SHGetSpecialFolderPath(hwndOwner: HWND; lpszPath: PChar;
  nFolder: Integer; fCreate: BOOL): BOOL; stdcall;
```

Параметр hwndOwner содержит дескриптор окна, которое будет служить в качестве владельца любых диалоговых окон, вызываемых этой функцией.

Параметр lpszPath является указателем на буфер, предназначенный для занесения пути. Размер этого буфера должен быть не меньше значения MAX_PATH.

В параметре nFolder определяется имя специальной папки, путь к которой требуется получить. В табл. 24.4 приведены значения, которые может принимать этот параметр.

С помощью параметра fCreate определяется, будет ли создана папка в случае, если таковая отсутствует.

Таблица 24.4. Возможные значения параметра nFolder

Флаг	Описание
CSIDL_ALTSTARTUP	Папка, соответствующая нелокализованной группе Startup конкретного пользователя
CSIDL_APPDATA	Папка, выделенная как общее место хранения данных приложения
CSIDL_BITBUCKET	Папка, включающая файловые объекты из корзины пользователя. Расположение этой папки не зафиксировано в системном реестре; чтобы не допустить ее перемещения или удаления, она помечена атрибутами "скрытый" и "системный"
CSIDL_COMMON_ALTSTARTUP	Папка, соответствующая нелокализованной группе Startup всех пользователей
CSIDL_COMMON_DESKTOPDIRECTORY	Папка, содержащая объекты (файлы и папки), которые отображаются на рабочем столе для всех пользователей

Продолжение табл. 24.4

Флаг	Описание
CSIDL_COMMON_FAVORITES	Папка для хранения “Избранного” всех пользователей
CSIDL_COMMON_PROGRAMS	Папка, содержащая папки групп программ из меню Start (для всех пользователей)
CSIDL_COMMON_STARTMENU	Папка, содержащая программы и папки, отображаемые в меню Start (для всех пользователей)
CSIDL_COMMON_STARTUP	Папка, содержащая программы из папки Startup (для всех пользователей)
CSIDL_CONTROLS	Виртуальная папка, содержащая пиктограммы приложений панели управления
CSIDL_COOKIES	Папка для хранения файлов Internet типа cookies
CSIDL_DESKTOP	Виртуальная папка рабочего стола (Windows Desktop) для хранения корневого пространства имен
CSIDL_DESKTOPDIRECTORY	Папка, где физически расположены файловые объекты рабочего стола (не путать с папкой Desktop!)
CSIDL_DRIVES	Папка Мой компьютер (My Computer), содержащая все объекты на локальном компьютере: дисковые устройства, принтеры и панель управления. Эта папка может также включать подключенные сетевые устройства
CSIDL_FAVORITES	Папка, в которой хранится “Избранное” конкретного пользователя
CSIDL_FONTS	Виртуальная папка, содержащая шрифты
CSIDL_HISTORY	Папка, в которой хранятся ссылки на адреса узлов Internet, посещенных пользователем
CSIDL_INTERNET	Виртуальная папка, представляющая Internet
CSIDL_INTERNET_CACHE	Папка, служащая общим хранилищем для временных файлов Internet
CSIDL_NETHOOD	Папка, содержащая объекты сетевого окружения (Network Neighborhood)
CSIDL_NETWORK	Виртуальная папка Network Neighborhood, представляющая собой высший уровень сетевой иерархии
CSIDL_PERSONAL	Папка, используемая как общее хранилище для документов
CSIDL_PRINTERS	Виртуальная папка, содержащая информацию обо всех установленных принтерах
CSIDL_PRINTHOOD	Папка, используемая как общее хранилище для ярлыков принтеров
CSIDL_PROGRAMS	Папка, содержащая группы программ (которые также являются папками) конкретного пользователя
CSIDL_RECENT	Папка, содержащая ссылки на документы, с которыми пользователь работал в последнее время

Окончание табл. 24.4

Флаг	Описание
CSIDL_SENDTO	Папка, содержащая объекты, появляющиеся при выборе команды меню Send To
CSIDL_STARTMENU	Папка, содержащая пункты меню Start
CSIDL_STARTUP	Папка, соответствующая группе программ Startup (для конкретного пользователя). Программы из этой папки автоматически запускаются каждый раз при регистрации в Windows NT либо при запуске Windows 95 или Windows 98
CSIDL_TEMPLATES	Папка для хранения шаблонов документов

Создание ярлыка

Интерфейс IShellLink инкапсулирует в объект ярлыка, но не определяет, как он может быть считан или записан на диск. Поэтому реализация этого интерфейса требует дополнительной поддержки интерфейса IPersistFile, который используется для обеспечения доступа к файлам. Интерфейс IPersistFile содержит методы чтения и записи данных на диск и определяется следующим образом:

```
type
  IPersistFile = interface(IPersist)
    ['{0000010B-0000-0000-C000-000000000046}']
    function IsDirty: HResult; stdcall;
    function Load(pszFileName: POleStr; dwMode: Longint): HResult; stdcall;
    function Save(pszFileName: POleStr; fRemember: BOOL): HResult; stdcall;
    function SaveCompleted(pszFileName: POleStr): HResult; stdcall;
    function GetCurFile(out pszFileName: POleStr): HResult; stdcall;
  end;
```

На заметку

Более подробное описание интерфейса IPersistFile и всех его методов можно найти в интерактивной справочной системе Win32.

Поскольку класс, реализующий интерфейс IShellLink, всегда содержит реализацию интерфейса IPersistFile, экземпляр интерфейса IShellLink можно использовать для получения экземпляра интерфейса IPersistFile путем выполнения операции преобразования типа с помощью оператора as, как показано ниже.

```
var
  SL: IShellLink;
  PF: IPersistFile;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
  PF := SL as IPersistFile;
  // используем экземпляры интерфейсов PF и SL
end;
```

Как уже упоминалось выше в этой главе, использование объектов COM-интерфейсов не отличается от использования обычных объектов Object Pascal. В следующем фрагменте кода приведен пример создания ярлыка для приложения Notepad (Блокнот), который помещается на поверхность рабочего стола системы:

```
procedure MakeNotepad;
const
  { Примечание: Предположим, что программа Notepad расположена в папке Windows на
диске C: }
  AppName = 'c:\windows\notepad.exe';
var
  SL: IShellLink;
  PF: IPersistFile;
  LnkName: WideString;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
  { При реализации интерфейса IShellLink обязательно
    реализуется и интерфейс IPersistFile }
  PF := SL as IPersistFile;
  { Устанавливаем путь к требуемому файлу: }
  OleCheck(SL.SetPath(PChar(AppName)));
  { Указываем размещение и имя файла создаваемого ярлыка: }
  LnkName := GetFolderLocation('Desktop') + '\' +
    ChangeFileExt(ExtractFileName(AppName), '.lnk');
  { Сохраняем файл ярлыка }
  PF.Save(PWideChar(LnkName), True);
end;
```

В этой процедуре метод `SetPath()` интерфейса `IShellLink` используется для указания пути к выполняемому файлу или документу, для которого создается ярлык (в данном случае это программа Notepad). Затем определяется полный путь и имя файла ярлыка, для чего используется информация, возвращаемая функцией `GetFolderLocation('Desktop')` (описанной выше в этом разделе). Кроме того, для изменения расширения файла `Notepad.exe` с `.exe` на `.lnk` используется функция `ChangeFileExt()`. Новое имя файла сохраняется в переменной `LnkName`. Затем с помощью метода `Save()` новый ярлык сохраняется в дисковом файле. Как вы уже знаете, при завершении приведенной выше процедуры и выходе экземпляров интерфейсов `SL` и `PF` за пределы области видимости соответствующие ссылки освобождаются.

Получение и запись информации ярлыка

Как видно из определения интерфейса `IShellLink`, он включает несколько методов вида Get*XXX*() и Set*XXX*(), позволяющих считывать и записывать параметры, определяющие различные аспекты ярлыка. Рассмотрим следующее объявление записи, поля которой можно просматривать и изменять:

```
type
  TShellLinkInfo = record
    PathName: string;
    Arguments: string;
    Description: string;
```

```
    WorkingDirectory: string;
    IconLocation: string;
    IconIndex: Integer;
    ShowCmd: Integer;
    HotKey: Word;
  end;
```

Располагая подобной записью, можно создать функции для работы с параметрами ярлыка (для считывания значений параметров в поля этой записи или для установки параметров ярлыка в соответствии с содержимым полей записи). Исходный текст этих функций содержится в модуле `WinShell.pas`, приведенном в листинге 24.5.

Листинг 24.5. Модуль `WinShell.pas`, содержащий функции доступа к параметрам ярлыка

```
unit WinShell;

interface

uses SysUtils, Windows, Registry, ActiveX, ShlObj;

type
  EShellOleError = class(Exception);

  TShellLinkInfo = record
    PathName: string;
    Arguments: string;
    Description: string;
    WorkingDirectory: string;
    IconLocation: string;
    IconIndex: integer;
    ShowCmd: integer;
    HotKey: word;
  end;

  TSpecialFolderInfo = record
    Name: string;
    ID: Integer;
  end;

const
  SpecialFolders: array[0..29] of TSpecialFolderInfo = (
    (Name: 'Alt Startup'; ID: CSIDL_ALTSTARTUP),
    (Name: 'Application Data'; ID: CSIDL_APPDATA),
    (Name: 'Recycle Bin'; ID: CSIDL_BITBUCKET),
    (Name: 'Common Alt Startup'; ID: CSIDL_COMMON_ALTSTARTUP),
    (Name: 'Common Desktop'; ID: CSIDL_COMMON_DESKTOPDIRECTORY),
    (Name: 'Common Favorites'; ID: CSIDL_COMMON_FAVORITES),
    (Name: 'Common Programs'; ID: CSIDL_COMMON_PROGRAMS),
    (Name: 'Common Start Menu'; ID: CSIDL_COMMON_STARTMENU),
    (Name: 'Common Startup'; ID: CSIDL_COMMON_STARTUP),
```

```
    (Name: 'Controls'; ID: CSIDL_CONTROLS),
    (Name: 'Cookies'; ID: CSIDL_COOKIES),
    (Name: 'Desktop'; ID: CSIDL_DESKTOP),
    (Name: 'Desktop Directory'; ID: CSIDL_DESKTOPDIRECTORY),
    (Name: 'Drives'; ID: CSIDL_DRIVES),
    (Name: 'Favorites'; ID: CSIDL_FAVORITES),
    (Name: 'Fonts'; ID: CSIDL_FONTS),
    (Name: 'History'; ID: CSIDL_HISTORY),
    (Name: 'Internet'; ID: CSIDL_INTERNET),
    (Name: 'Internet Cache'; ID: CSIDL_INTERNET_CACHE),
    (Name: 'Network Neighborhood'; ID: CSIDL_NETHOOD),
    (Name: 'Network Top'; ID: CSIDL_NETWORK),
    (Name: 'Personal'; ID: CSIDL_PERSONAL),
    (Name: 'Printers'; ID: CSIDL_PRINTERS),
    (Name: 'Printer Links'; ID: CSIDL_PRINTHOOD),
    (Name: 'Programs'; ID: CSIDL_PROGRAMS),
    (Name: 'Recent Documents'; ID: CSIDL_RECENT),
    (Name: 'Send To'; ID: CSIDL_SENDTO),
    (Name: 'Start Menu'; ID: CSIDL_STARTMENU),
    (Name: 'Startup'; ID: CSIDL_STARTUP),
    (Name: 'Templates'; ID: CSIDL_TEMPLATES));

function CreateShellLink(const AppName, Desc: string; Dest: Integer): string;
function GetSpecialFolderPath(Folder: Integer; CanCreate: Boolean): string;
procedure GetShellLinkInfo(const LinkFile: WideString;
  var SLI: TShellLinkInfo);
procedure SetShellLinkInfo(const LinkFile: WideString;
  const SLI: TShellLinkInfo);

implementation
uses ComObj;

function GetSpecialFolderPath(Folder: Integer; CanCreate: Boolean): string;
var
  FilePath: array[0..MAX_PATH] of char;
begin
  { Получение пути выбранного местоположения. }
  SHGetSpecialFolderPathW(0, FilePath, Folder, CanCreate);
  Result := FilePath;
end;

function CreateShellLink(const AppName, Desc: string; Dest: Integer): string;
{ Создает ярлык для приложения или документа, определяемого константой
  AppName и описанием в строке Desc. Ярлык будет помещен в папку, заданную
  параметром Dest и указанную одной из строковых констант массива,
 определенного в заголовке этого модуля. Возвращает полное имя файла ярлыка. }
var
  SL: IShellLink;
  PF: IPersistFile;
  LnkName: WideString;
```

```
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
  { Реализация интерфейса IShellLink всегда поддерживает интерфейс
    PersistFile. Получаем указатель на этот интерфейс. }
  PF := SL as IPersistFile;
  { Устанавливаем путь к нужному файлу: }
  OleCheck(SL.SetPath(PChar(AppName)));
  if Desc <> '' then
    OleCheck(SL.SetDescription(PChar(Desc)));    // Устанавливаем описание
  { Создаем путь и имя файла ярлыка. }
  LnkName := GetSpecialFolderPath(Dest, True) + '\' +
    ChangeFileExt(AppName, 'lnk');
  { Сохраняем файл ярлыка: }
  PF.Save(PWideChar(LnkName), True);     // Сохраняем файл ярлыка
  Result := LnkName;
end;

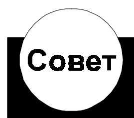
procedure GetShellLinkInfo(const LinkFile: WideString;
  var SLI: TShellLinkInfo);
{ Получаем информацию о существующем ярлыке. }
var
  SL: IShellLink;
  PF: IPersistFile;
  FindData: TWin32FindData;
  AStr: array[0..MAX_PATH] of char;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
  { Реализация интерфейса IShellLink всегда поддерживает интерфейс
    IPersistFile. Получаем указатель на него. }
  PF := SL as IPersistFile;
  { Загружаем файл в объект IPersistFile }
  OleCheck(PF.Load(PWideChar(LinkFile), STGM_READ));
  { Ищем ярлык путем вызова функции Resolve интерфейса. }
  OleCheck(SL.Resolve(0, SLR_ANY_MATCH or SLR_NO_UI));
  { Получаем всю информацию о ярлыке! }
  with SLI do
  begin
    OleCheck(SL.GetPath(AStr, MAX_PATH, FindData, SLGP_SHORTPATH));
    PathName := AStr;
    OleCheck(SL.GetArguments(AStr, MAX_PATH));
    Arguments := AStr;
    OleCheck(SL.GetDescription(AStr, MAX_PATH));
    Description := AStr;
    OleCheck(SL.GetWorkingDirectory(AStr, MAX_PATH));
    WorkingDirectory := AStr;
    OleCheck(SL.GetIconLocation(AStr, MAX_PATH, IconIndex));
    IconLocation := AStr;
    OleCheck(SL.GetShowCmd(ShowCmd));
```

```
    OleCheck(SL.GetHotKey(HotKey));
  end;
end;

procedure SetShellLinkInfo(const LinkFile: WideString;
  const SLI: TShellLinkInfo);
{ Записывает информацию в существующий файл ярлыка. }
var
  SL: IShellLink;
  PF: IPersistFile;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
     IShellLink, SL));
  { Реализация интерфейса IShellLink всегда поддерживает интерфейс
    IPersistFile. Получаем указатель на него. }
  PF := SL as IPersistFile;
  { Загружаем файл в объект IPersistFile. }
  OleCheck(PF.Load(PWideChar(LinkFile), STGM_SHARE_DENY_WRITE));
  { Отыскиваем требуемый ярлык посредством вызова функции интерфейса Resolve. }
  OleCheck(SL.Resolve(0, SLR_ANY_MATCH or SLR_UPDATE or SLR_NO_UI));
  { Записываем всю необходимую информацию! }
  with SLI, SL do
  begin
    OleCheck(SetPath(PChar(PathName)));
    OleCheck(SetArguments(PChar(Arguments)));
    OleCheck(SetDescription(PChar(Description)));
    OleCheck(SetWorkingDirectory(PChar(WorkingDirectory)));
    OleCheck(SetIconLocation(PChar(IconLocation), IconIndex));
    OleCheck(SetShowCmd(ShowCmd));
    OleCheck(SetHotKey(HotKey));
  end;
  PF.Save(PWideChar(LinkFile), True);    // Сохраняем файл
end;

end.
```

Из используемых в данном модуле методов интерфейса IShellLink лишь метод Resolve() нуждается в некоторых разъяснениях. Его следует вызывать после того, как с помощью интерфейса IPersistFile объекта IShellLink будет загружен файл ярлыка. Этот метод ищет заданный файл ярлыка и заполняет объект IShellLink значениями, содержащимися в найденном файле.



Просматривая текст функции GetShellLinkInfo() (см. листинг 24.5) обратите внимание на то, что для получаемых значений используется локальный массив AStr. Для этой же цели можно было бы воспользоваться функцией SetLength(), выделяющей память для хранения строк, однако предпочтение было отдано массиву AStr — поскольку применение функции SetLength() для такого большого количества строк приводит к значительной фрагментации кучи приложения. Использование массива AStr, как промежуточного звена, препятствует этому явлению. Более того, поскольку длина строк устанавливается лишь один раз, использование массива AStr несколько ускоряет процесс.

Пример приложения

Проще всего познакомиться с возможностями рассмотренных в предыдущем разделе функций и интерфейсов, включив их в какое-нибудь приложение. В качестве примера рассмотрим проект ShellLink. Главная форма этого проекта показана на рис. 24.6.

Рис. 24.6. Главная форма ShellLink*, в которой отображена информация об одном из ярлыков на рабочем столе системы*

Текст главного модуля приложения приведен в листинге 24.6 (модуль Main.pas). Кроме модуля Main.pas в проект входят два дополнительных модуля — NewLinkU.pas (листинг 24.7) и PickU.pas (листинг 24.8).

Листинг 24.6. Модуль Main.pas — главный модуль проекта ShellLink

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls, ExtCtrls, Spin, WinShell, Menus;

type
  TMainForm = class(TForm)
    Panel1: TPanel;
    btnOpen: TButton;
    edLink: TEdit;
    btnNew: TButton;
    btnSave: TButton;
    Label3: TLabel;
    Panel2: TPanel;
    Label1: TLabel;
    Label2: TLabel;
    Label4: TLabel;
    Label5: TLabel;
```

```
    Label6: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    Label9: TLabel;
    edIcon: TEdit;
    edDesc: TEdit;
    edWorkDir: TEdit;
    edArg: TEdit;
    cbShowCmd: TComboBox;
    hkHotKey: THotKey;
    speIcnIdx: TSpinEdit;
    pnlIconPanel: TPanel;
    imgIconImage: TImage;
    btnExit: TButton;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Open1: TMenuItem;
    Save1: TMenuItem;
    NewLInk1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    edPath: TEdit;
    procedure btnOpenClick(Sender: TObject);
    procedure btnNewClick(Sender: TObject);
    procedure edIconChange(Sender: TObject);
    procedure btnSaveClick(Sender: TObject);
    procedure btnExitClick(Sender: TObject);
    procedure About1Click(Sender: TObject);
  private
    procedure GetControls(var SLI: TShellLinkInfo);
    procedure SetControls(const SLI: TShellLinkInfo);
    procedure ShowIcon;
    procedure OpenLinkFile(const LinkFileName: String);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses PickU, NewLinkU, AboutU, CommCtrl, ShellAPI;

type
  THotKeyRec = record
    Char, ModCode: Byte;
  end;
```

```
procedure TMainForm.SetControls(const SLI: TShellLinkInfo);
{ Устанавливаем значения элементов управления интерфейса
  пользователя на основе содержимого параметра SLI. }
var
  Mods: THKModifiers;
begin
  with SLI do
  begin
    edPath.Text := PathName;
    edIcon.Text := IconLocation;
    { Если путь пиктограммы отсутствует, а ярлык указывает на выполняемый файл,
      то для пути пиктограммы используем имя выполняемого файла.
      Это нужно сделать для того, чтобы можно было игнорировать индекс
      пиктограммы в случае, если не заполнено поле пути пиктограммы,
      а выполняемый файл может содержать несколько пиктограмм. }
    if (IconLocation = '') and
      (CompareText(ExtractFileExt(PathName), 'EXE') = 0) then
      edIcon.Text := PathName;
    edWorkDir.Text := WorkingDirectory;
    edArg.Text := Arguments;
    speIcnIdx.Value := IconIndex;
    edDesc.Text := Description;
    { Константы SW_* начинаются с 1 }
    cbShowCmd.ItemIndex := ShowCmd - 1;
    { Код горячей клавиши в младшем байте }
    hkHotKey.HotKey := Lo(HotKey);
    { Получаем информацию о флагах модификаторов в старшем байте. }
    Mods := [];
    if (HOTKEYF_ALT and Hi(HotKey)) <> 0 then include(Mods, hkAlt);
    if (HOTKEYF_CONTROL and Hi(HotKey)) <> 0 then include(Mods, hkCtrl);
    if (HOTKEYF_EXT and Hi(HotKey)) <> 0 then include(Mods, hkExt);
    if (HOTKEYF_SHIFT and Hi(HotKey)) <> 0 then include(Mods, hkShift);
    { Устанавливаем набор модификаторов. }
    hkHotKey.Modifiers := Mods;
  end;
  ShowIcon;
end;

procedure TMainForm.GetControls(var SLI: TShellLinkInfo);
{ Получаем значения элементов управления пользовательского
  интерфейса и используем их для установки значений записи SLI. }
var
  CtlMods: THKModifiers;
  HR: THotKeyRec;
begin
  with SLI do
  begin
    PathName := edPath.Text;
    IconLocation := edIcon.Text;
    WorkingDirectory := edWorkDir.Text;
```

```
    Arguments := edArg.Text;
    IconIndex := speIcnIdx.Value;
    Description := edDesc.Text;
    { Константы SW_* начинаются с 1 }
    ShowCmd := cbShowCmd.ItemIndex + 1;
    { Получаем символ горячей клавиши }
    word(HR) := hkHotKey.HotKey;
    { Определяем, какие используются модификаторы клавиш. }
    CtlMods := hkHotKey.Modifiers;
    with HR do begin
        ModCode := 0;
        if (hkAlt in CtlMods) then ModCode := ModCode or HOTKEYF_ALT;
        if (hkCtrl in CtlMods) then ModCode := ModCode or HOTKEYF_CONTROL;
        if (hkExt in CtlMods) then ModCode := ModCode or HOTKEYF_EXT;
        if (hkShift in CtlMods) then ModCode := ModCode or HOTKEYF_SHIFT;
    end;
    HotKey := word(HR);
  end;
end;

procedure TMainForm.ShowIcon;
{ Извлекаем пиктограмму из соответствующего файла
  и отображаем ее в объекте IconImage. }
var
  HI: THandle;
  IcnFile: string;
  IconIndex: word;
begin
  { Получаем имя файла пиктограммы }
  IcnFile := edIcon.Text;
  { Если это поле пустое, используем имя выполняемого файла. }
  if IcnFile = '' then
    IcnFile := edPath.Text;
  { Проверка существования файла. }
  if FileExists(IcnFile) then
  begin
    IconIndex := speIcnIdx.Value;
    { Извлекаем пиктограмму из файла. }
    HI := ExtractAssociatedIcon(hInstance, PChar(IcnFile), IconIndex);
    { Присваиваем дескриптор пиктограммы объекту IconImage. }
    imgIconImage.Picture.Icon.Handle := HI;
  end;
end;

procedure TMainForm.OpenLinkFile(const LinkFileName: string);
{ Открываем файл ярлыка, выбираем информацию и отображаем ее
  с помощью элементов управления пользовательского интерфейса. }
var
  SLI: TShellLinkInfo;
begin
```

```
  edLink.Text := LinkFileName;
  try
    GetShellLinkInfo(LinkFileName, SLI);
  except
    on EShellOleError do
      MessageDlg('Ошибка при открытии файла ярлыка ', mtError, [mbOk], 0);
  end;
  SetControls(SLI);
end;

procedure TMainForm.btnOpenClick(Sender: TObject);
{ Обработчик OnClick для кнопки btnOpen. }
var
  LinkFile: String;
begin
  if GetLinkFile(LinkFile) then
    OpenLinkFile(LinkFile);
end;

procedure TMainForm.btnNewClick(Sender: TObject);
{ Обработчик OnClick для кнопки btnNew. }
var
  FileName: string;
  Dest: Integer;
begin
  if GetNewLinkName(FileName, Dest) then
    OpenLinkFile(CreateShellLink(FileName, '', Dest));
end;

procedure TMainForm.edIconChange(Sender: TObject);
{ Обработчик OnChange для строки редактирования edIcon и счетчика speIcnIdx. }
begin
  ShowIcon;
end;

procedure TMainForm.btnSaveClick(Sender: TObject);
{ Обработчик OnClick для кнопки btnSave. }
var
  SLI: TShellLinkInfo;
begin
  GetControls(SLI);
  try
    SetShellLinkInfo(edLink.Text, SLI);
  except
    on EShellOleError do
      MessageDlg('При записи информации возникла ошибка ', mtError, [mbOk], 0);
  end;
end;
```

```
procedure TMainForm.btnExitClick(Sender: TObject);
{ Обработчик OnClick для кнопки btnExit. }
begin
  Close;
end;

procedure TMainForm.About1Click(Sender: TObject);
{ Обработчик OnClick для команды меню Help⇨About. }
begin
  AboutBox;
end;

end.
```

Листинг 24.7. Модуль `NewLinkU.pas` — модуль формы, предназначенной для создания нового ярлыка

```
unit NewLinkU;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, Buttons, StdCtrls;
type
  TNewLinkForm = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    edLinkTo: TEdit;
    btnOk: TButton;
    btnCancel: TButton;
    cbLocation: TComboBox;
    sbOpen: TSpeedButton;
    OpenDialog: TOpenDialog;
    procedure sbOpenClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  end;

function GetNewLinkName(var LinkTo: string; var Dest: Integer): Boolean;

implementation

uses WinShell;

{$R *.DFM}

function GetNewLinkName(var LinkTo: string; var Dest: Integer): Boolean;
{ Получаем имя файла и папку назначения для хранения нового ярлыка.
  Функция модифицирует параметры только в том случае, если Result = True }
```

```
begin
  with TNewLinkForm.Create(Application) do
  try
    cbLocation.ItemIndex := 0;
    Result := ShowModal = mrOk;
    if Result then
    begin
      LinkTo := edLinkTo.Text;
      Dest := cbLocation.ItemIndex;
    end;
  finally
    Free;
  end;
end;

procedure TNewLinkForm.sbOpenClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    edLinkTo.Text := OpenDialog.FileName;
end;

procedure TNewLinkForm.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := Low(SpecialFolders) to High(SpecialFolders) do
    cbLocation.Items.Add(SpecialFolders[I].Name);
end;

end.
```

Листинг 24.8. Модуль `PickU.pas` — модуль формы, разрешающей пользователю выбрать местоположение ярлыка

```
unit PickU;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, FileCtrl;

type
  TLinkForm = class(TForm)
    lbLinkFiles: TFileListBox;
    btnOk: TButton;
    btnCancel: TButton;
    cbLocation: TComboBox;
    Label1: TLabel;
```

```
    procedure lbLinkFilesDblClick(Sender: TObject);
    procedure cbLocationChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  end;

function GetLinkFile(var S: String): Boolean;

implementation

{$R *.DFM}

uses WinShell, ShlObj;

function GetLinkFile(var S: String): Boolean;
{ Имя файла ярлыка возвращается в параметр S.
  Параметр S модифицируется только в случае, если Result = True. }
begin
  with TLinkForm.Create(Application) do
    try
      { Проверяем, что расположение выбрано }
      cbLocation.ItemIndex := 0;
      { Получаем путь выделенного расположения }
      cbLocationChange(nil);
      Result := ShowModal = mrOk;
      { Получаем полный путь для файла ярлыка }
      if Result then
        S := lbLinkFiles.Directory + '\' +
          lbLinkFiles.Items[lbLinkFiles.ItemIndex];
    finally
      Free;
    end;
end;

procedure TLinkForm.lbLinkFilesDblClick(Sender: TObject);
begin
  ModalResult := mrOk;
end;

procedure TLinkForm.cbLocationChange(Sender: TObject);
var
  Folder: Integer;
begin
  { Получаем путь выбранного положения }
  Folder := SpecialFolders[cbLocation.ItemIndex].ID;
  lbLinkFiles.Directory := GetSpecialFolderPath(Folder, False);
end;

procedure TLinkForm.FormCreate(Sender: TObject);
var
  I: Integer;
```

```
begin
  for I := Low(SpecialFolders) to High(SpecialFolders) do
    cbLocation.Items.Add(SpecialFolders[I].Name);
end;

end.
```

Расширения оболочки

Для получения предельной расширяемости в оболочке Windows предусмотрены средства, позволяющие создать код, который выполняется в рамках процесса самой оболочки и в ее пространстве имен. *Расширения оболочки* (shell extensions) реализуются в виде внутренних COM-серверов, которые создаются и используются оболочкой Windows.

На заметку Поскольку расширения оболочки являются COM-серверами, их понимание требует знания основ технологии COM. Необходимую информацию можно получить в главе 23, "COM-ориентированные технологии".

Существует несколько типов расширений оболочки, связанных с различными аспектами деятельности системной оболочки. Любое расширение оболочки (другое название — *обработчик*) должно реализовывать по крайней мере один COM-интерфейс. Оболочка поддерживает следующие типы расширений.

- **Обработчики перемещений** реализуют интерфейс ICopyHook. Эти расширения позволяют получать соответствующие уведомления при копировании, перемещении, удалении или переименовании папок, а также предотвращать выполнение всех этих операций.
- **Обработчики контекстного меню** реализуют интерфейсы IContextMenu и IShellExtInit. Эти расширения позволяют добавлять команды в контекстное меню определенного файлового объекта оболочки.
- **Обработчики перетаскивания** также реализуют интерфейсы IContextMenu и IShellExtInit. Эти расширения практически эквивалентны реализации обработчиков контекстного меню, за исключением того, что они вызываются в том случае, когда пользователь перетаскивает объект в новое место.
- **Обработчики пиктограмм** реализуют интерфейсы IExtractIcon и IPersistFile. Эти обработчики позволяют присваивать разные пиктограммы различным экземплярам файлового объекта одного типа.
- **Обработчики вкладок свойств** реализуют интерфейсы IShellPropSheetExt и IShellExtInit. Они позволяют добавлять вкладки к диалоговым окнам свойств, ассоциированным с типом файла.
- **Обработчики цели** реализуют интерфейсы IDropTarget и IPersistFile. Эти расширения позволяют определить действия оболочки при перетаскивании и опускании одного объекта оболочки на другой.
- **Обработчики объектов данных** реализуют интерфейсы IDataObject и IPersistFile. Эти расширения предоставляют объекты данных, которые используются при перетаскивании с помощью мыши, копировании (команда Copy) и вставке (команда Paste) файлов.

Отладка расширений оболочки

Прежде чем углубляться в детали создания расширений оболочки, остановимся на теме их отладки. Поскольку расширения оболочки выполняются в пределах собственного процесса оболочки Windows, возникает закономерный вопрос: "Как же "внедриться" в оболочку с целью отладки ее расширений?"

Решение проблемы основано на том, что оболочка представляет собой выполняемый файл (не слишком отличающийся от любого другого приложения), который называется `explorer.exe`. Как и любой другой выполняемый файл, этот файл имеет свойства. Но именно здесь и заключается его отличие от других выполняемых файлов: оболочку вызывает лишь первый экземпляр `explorer.exe`. Все последующие экземпляры просто вызывают дополнительные окна Explorer.

Используя один малоизвестный трюк, можно закрыть оболочку, не выходя из Windows. Для отладки расширения в среде Delphi можете воспользоваться следующим алгоритмом.

1. С помощью команды Run⇨Parameter (Выполнить⇨Параметры) сделайте файл `explorer.exe` основным приложением для расширения оболочки. Убедитесь, что задан полный путь к файлу (например, `c:\windows\explorer.exe`).
2. В Windows выполните команду Start⇨Shut Down (Пуск⇨Завершение работы). При этом открывается соответствующее диалоговое окно.
3. В диалоговом окне Shut Down Windows (Завершение работы Windows) щелкните на кнопке No (Отмена), удерживая при этом нажатой комбинацию клавиш <Ctrl+Alt+Shift>. Оболочка закроется, но вы останетесь в среде Windows.
4. Используя комбинацию клавиш <Alt+Tab>, переключитесь в Delphi и запустите расширение оболочки. При этом под управлением отладчика Delphi будет вызвана новая копия оболочки. Теперь можно установить в нужных местах программы точки останова и отлаживать расширение оболочки как обычное приложение.
5. Если необходимо выйти из Windows, эта задача вполне осуществима и без использования сервиса оболочки: с помощью комбинации клавиш <Ctrl+Esc>, вызовите меню Tasks и выберите в нем команду Windows⇨Shut Down.

В последующих разделах этой главы описанные выше расширения оболочки будут показаны "в разрезе". Мы рассмотрим использование обработчиков перемещения, контекстного меню и пиктограмм.

Мастер COM-объектов

Прежде чем приступать к обсуждению каждой из библиотек, поддерживающих то или иное расширение оболочки, рассмотрим процедуру их создания. Поскольку расширения оболочки являются внутренними COM-серверами, можно позволить интегрированной среде разработки Delphi выполнить бо́льшую часть рутинной работы по созданию исходного кода. Для каждого из расширений работа начинается со следующих двух действий:

1. Во вкладке ActiveX диалогового окна New Item дважды щелкните на пиктограмме ActiveX Library. При этом создается новая библиотека DLL COM-сервера, в которую можно будет вставлять COM-объекты.
2. Во вкладке ActiveX диалогового окна New Item дважды щелкните на пиктограмме COM Object. При этом вызывается мастер создания COM-сервера. В диалоговом окне мастера введите имя и описание для создаваемого расширения оболочки и выберите вариант модели работы с потоками Apartment. По щелчку на кнопке OK будет создан новый модуль, содержащий код для создаваемого COM-объекта.

Обработчики перемещений

Как уже упоминалось выше в этой главе, расширения оболочки, связанные с перемещениями, позволяют устанавливать обработчики, которые будут получать уведомления всякий раз при выполнении операции копирования, удаления, перемещения или переименования. После получения такого уведомления обработчик может не допустить выполнения соответствующей процедуры. Учтите, что этот обработчик вызывается только для объектов папок и принтеров и не вызывается для файлов и других объектов.

Первым шагом при построении обработчика перемещений является создание объекта, производного от класса TComObject, в котором следует реализовать интерфейс ICopyHook. Этот интерфейс определяется в модуле ShlObj следующим образом:

```
type
  ICopyHook = interface(IUnknown)
    ['{000214EF-0000-0000-C000-000000000046}']
    function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
      pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD;
      pszDestFile: PAnsiChar; dwDestAttribs: DWORD): UINT; stdcall;
  end;
```

Метод CopyCallback()

Как видите, ICopyHook — достаточно простой интерфейс; в нем реализована лишь одна функция CopyCallback(). Эта функция вызывается при каких-либо манипуляциях с папкой. Ниже описаны параметры этой функции.

Параметр Wnd — это дескриптор окна, которое будет использовано обработчиком перемещений в качестве родительского для отображаемых им окон.

Параметр wFunc указывает на выполняемую операцию; этот параметр может принимать одно из значений, приведенных в табл. 24.5.

Таблица 24.5. Значения параметра wFunc функции CopyCallback()

Константа	Значение	Назначение
FO_COPY	$2	Копирует файл, задаваемый параметром pszSrcFile, в файл, определяемый параметром pszDestFile
FO_DELETE	$3	Удаляет файл, определяемый параметром pszSrcFile
FO_MOVE	$1	Перемещает файл, определяемый параметром pszSrcFile в файл, определяемый параметром pszDestFile
FO_RENAME	$4	Переименовывает файл, определяемый параметром pszSrcFile
PO_DELETE	$13	Удаляет принтер, задаваемый параметром pszSrcFile
PO_PORTCHANGE	$20	Изменяет порт принтера. Параметры pszSrcFile и pszDestFile содержат списки, состоящие из строк, завершающихся двумя нулевыми символами. Каждый список содержит имя принтера, за которым следует имя порта. Имя порта из параметра pszSrcFile является именем порта текущего принтера, а имя порта из параметра pszDestFile является именем порта нового принтера

Окончание табл. 24.5

Константа	Значение	Назначение
PO_RENAME	$14	Переименовывает принтер, задаваемый параметром pszSrcFile
PO_REN_PORT	$34	Комбинация параметров PO_RENAME и PO_PORTCHANGE

Параметр wFlags содержит флаги, управляющие операцией. Этот параметр может быть комбинацией значений, представленных в табл. 24.6.

Таблица 24.6. Значения параметра wFlags функции CopyCallback()

Константа	Значение	Назначение
FOF_ALLOWUNDO	$40	Сохраняет информацию для отмены действия (если это возможно)
FOF_MULTIDESTFILES	$1	Вместо одной папки для хранения всех исходных файлов функция SHFileOperation() определяет множество конечных файлов (по одному для каждого исходного файла). Обработчик перемещения обычно игнорирует это значение
FOF_NOCONFIRMATION	$10	В любом диалоговом окне ответом является вариант Yes to All (Да, для всех)
FOF_NOCONFIRMMKDIR	$200	Если операция требует создания новой папки, подтверждающее сообщение отображаться не будет
FOF_RENAMEONCOLLISION	$8	Предлагает для файла новое имя (например, Copy #1 of... (Копия #1...)) при операциях копирования, перемещения или переименования, если файл с указанным именем приемника уже существует
FOF_SILENT	$4	Не отображает индикатор процесса
FOF_SIMPLEPROGRESS	$100	Отображает индикатор процесса, но не показывает имена файлов

Используются также следующие параметры: pszSourceFile — имя исходной папки; dwSrcAttribs — атрибуты исходной папки; pszDestFile — имя папки приемника; dwDestAttribs — атрибуты папки приемника.

В отличие от большинства методов, этот интерфейс не возвращает результирующего OLE-кода. Вместо этого он возвращает одно из значений, определенных в модуле Windows (они перечислены в табл. 24.7).

Таблица 24.7. Значения параметра wFlags, возвращаемые функцией CopyCallBack()

Константа	Значение	Назначение
IDYES	6	Разрешает операцию
IDNO	7	Препятствует выполнению операции над текущим файлом, однако продолжает работу с остальными операциями (например, в случае копирования с использованием командного файла)
IDCANCEL	2	Препятствует выполнению текущей операции и отменяет все остальные операции

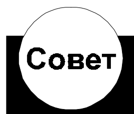
Реализация объекта TCopyHook

Поскольку объект TCopyHook реализует интерфейс лишь с одним методом, его описание выглядит достаточно компактно:

```
type
  TCopyHook = class(TComObject, ICopyHook)
  protected
    function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
    pszSrcFile: PAnsiChar;
    dwSrcAttribs: DWORD; pszDestFile: PAnsiChar; dwDestAttribs: DWORD): UINT;
      stdcall;
  end;
```

Реализация метода CopyCallback() также невелика по объему. Для подтверждения выполнения какой-либо операции вызывается функция Win32 API MessageBox(). Кстати, значение, возвращаемое методом CopyCallback(), аналогично значению, возвращаемому функцией MessageBox():

```
function TCopyHook.CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
  pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD; pszDestFile: PAnsiChar;
  dwDestAttribs: DWORD): UINT;
const
  MyMessage: string = 'Are you sure you want to mess with "%s"?';
begin
  // Подтвердить операцию
  Result := MessageBox(Wnd, PChar(Format(MyMessage, [pszSrcFile])),
    'D4DG Shell Extension', MB_YESNO);
end;
```



Вы, возможно, обратили внимание на то, что для вывода сообщений вместо функции Delphi MessageDlg() или ShowMessage() используется функция Win32 API MessageBox(). Причина проста — все дело в размере кода и эффективности его выполнения. Вызов любой функции из модуля Dialogs или Forms привел бы к тому, что к создаваемой библиотеке DLL была бы подсоединена значительная часть кода из библиотеки VCL. Отказавшись от использования функций из этих двух модулей, вы сократите размер созданной библиотеки DLL примерно на 70 Кбайт.

Возможно, в это трудно поверить, но об объекте TCopyHook сказано все что нужно. Да, вот еще одна деталь: прежде чем использовать любое расширение оболочки, оно должно быть зарегистрировано в системном реестре.

Регистрация

Кроме обычной регистрации, необходимой любому COM-серверу, обработчик перемещения должен иметь дополнительную точку входа в системный реестр в следующей ветви:

```
HKEY_CLASSES_ROOT\directory\shellex\CopyHookHandlers
```

Более того, в Windows NT требуется, чтобы все расширения оболочки были зарегистрированы как утвержденные (approved). В этом случае регистрация осуществляется и в дополнительной ветви:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion
↳\Shell Extensions\Approved
```

Для регистрации расширений оболочки можно использовать несколько способов. Они могут быть зарегистрированы с помощью `.reg`-файла или программы инсталляции. И, наконец, библиотека DLL с расширением оболочки может быть саморегистрирующейся. Последнее решение, хотя и требует несколько бо́льших усилий, является все же наилучшим, поскольку при этом расширение является самодостаточным пакетом, размещенным в единственном файле.

В предыдущей главе шла речь о том, что COM-объекты всегда создаются из фабрик классов. В среде VCL объекты фабрики класса ответственны также и за регистрацию создаваемого COM-объекта. Если COM-объект требует точки входа в системный реестр (как в случае с расширением оболочки), то для их установки достаточно переопределить стандартный метод `UpdateRegistry()` фабрики класса. В листинге 24.9 приведен код модуля `CopyMain`, содержащего специализированную фабрику класса для выполнения регистрации.

Листинг 24.9. Модуль `CopyMain.pas` — главный модуль реализации обработчика перемещений

```
unit CopyMain;

interface

uses Windows, ComObj, ShlObj;

type
  TCopyHook = class(TComObject, ICopyHook)
  protected
    function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
      pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD;
      pszDestFile: PAnsiChar; dwDestAttribs: DWORD): UINT; stdcall;
  end;

  TCopyHookFactory = class(TComObjectFactory)
  protected
    function GetProgID: string; override;
    procedure ApproveShellExtension(Register: Boolean; const ClsID: string);
      virtual;
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;

implementation

uses ComServ, SysUtils, Registry;

{ TCopyHook }

// Этот метод вызывается оболочкой для операций с папками
function TCopyHook.CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
  pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD; pszDestFile: PAnsiChar;
  dwDestAttribs: DWORD): UINT;
const
  MyMessage: string = 'Are you sure you want to mess with "%s"?';
```

```
begin
  // Подтверждаем операцию
  Result := MessageBox(Wnd,    PChar(Format(MyMessage, [pszSrcFile])),
    'D4DG Shell Extension', MB_YESNO);
end;

{ TCopyHookFactory }

function TCopyHookFactory.GetProgID: string;
begin
  { Идентификатор программы (ProgID) для расширения оболочки не требуется. }
  Result := '';
end;

procedure TCopyHookFactory.UpdateRegistry(Register: Boolean);
var
  ClsID: string;
begin
  ClsID := GUIDToString(ClassID);
  inherited UpdateRegistry(Register);
  ApproveShellExtension(Register, ClsID);
  if Register then
    { Добавляем в точку входа CopyHookHandlers системного реестра
      идентификатор класса расширения оболочки. }
    CreateRegKey('directory\shellex\CopyHookHandlers\' + ClassName, '',
      ClsID)
  else
    DeleteRegKey('directory\shellex\CopyHookHandlers\' + ClassName);
end;

procedure TCopyHookFactory.ApproveShellExtension(Register: Boolean;
  const ClsID: string);
{ Эту точку входа системного реестра необходимо заполнить для того,
  чтобы расширение работало корректно под управлением Windows NT. }
const
  SApproveKey =
    'SOFTWARE\Microsoft\Windows\CurrentVersion\Shell Extensions\Approved';
begin
  with TRegistry.Create do
    try
      RootKey := HKEY_LOCAL_MACHINE;
      if not OpenKey(SApproveKey, True) then Exit;
      if Register then WriteString(ClsID, Description)
      else DeleteValue(ClsID);
    finally
      Free;
    end;
end;

const
```

```
  CLSID_CopyHook: TGUID = '{66CD5F60-A044-11D0-A9BF-00A024E3867F}';

initialization
  TCopyHookFactory.Create(ComServer, TCopyHook, CLSID_CopyHook,
    'D4DG_CopyHook', 'D4DG Copy Hook Shell Extension Example',
    ciMultiInstance, tmApartment);
end.
```

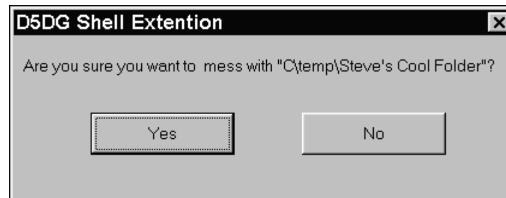


Рис. 24.7. Обработчик уведомлений в действии

Движущей силой, заставляющей фабрику класса `TCopyHookFactory` работать, является тот факт, что именно ее экземпляр, а не обычный объект `TComObjectFactory`, создается в разделе инициализации (`initialization`) этого модуля. На рис. 24.7 показано, что происходит при попытке переименовать папку в системной оболочке после установки библиотеки DLL с расширением перемещений.

Обработчики контекстных меню

Обработчики контекстных меню позволяют добавлять команды в локальное меню, связанные с файловыми объектами оболочки. Пример контекстного меню для выполняемого файла показан на рис. 24.8.

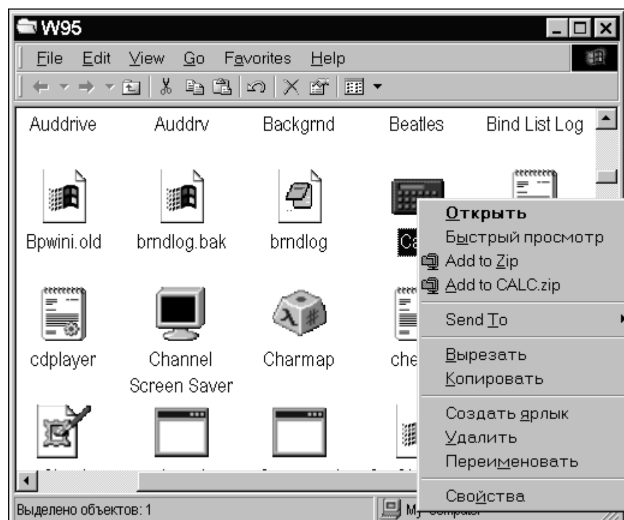


Рис. 24.8. Контекстное меню выполняемого файла

Действие расширений оболочки, связанных с контекстными меню, основано на реализации интерфейсов IShellExtInit и IContextMenu. В нашем примере эти интерфейсы реализованы с целью создания обработчика контекстного меню для файлов Borland Package Library (BPL). В контекстное меню для файлов этого типа будет добавлена команда, позволяющая получить информацию о содержащемся в файле пакете. Объект обработчика контекстного меню назовем TContextMenu и, подобно обработчику перемещений, сделаем класс TContextMenu производным от класса TComObject.

Интерфейс IShellExtInit

Для инициализации расширения оболочки используется интерфейс IShellExtInit, который определен в модуле ShlObj следующим образом:

```
type
  IShellExtInit = interface(IUnknown)
    ['{000214E8-0000-0000-C000-000000000046}']
    function Initialize(pidlFolder: PItemIDList; lpdobj: IDataObject;
      hKeyProgID: HKEY): HResult; stdcall;
  end;
```

В этом интерфейсе использован единственный метод — Initialize() — который и инициализирует обработчик контекстного меню. Ниже описаны параметры этого метода.

Параметр pidlFolder является указателем на структуру PItemIDList (список идентификаторов элементов) для папки, содержащей элемент, к которому относится отображаемое контекстное меню. Параметр lpdobj хранит объект интерфейса IDataObject, используемый для получения объектов, над которыми выполняется действие. Параметр hKeyProgID содержит ключ системного реестра для объекта файлового типа или для папки.

Реализация метода Initialize() приведена ниже. На первый взгляд этот код кажется довольно сложным, однако на самом деле все сводится к трем действиям: вызову функции lpobj.GetData() для получения данных из интерфейса IDataObject и двум вызовам функции DragQueryFile() (один — для получения количества файлов, а другой — для получения имени файла). Имя файла сохраняется в поле FFileName объекта lpdobj.

```
function TContextMenu.Initialize(pidlFolder: PItemIDList; lpdobj: IDataObject;
  hKeyProgID: HKEY): HResult;
var
  Medium: TStgMedium;
  FE: TFormatEtc;
begin
  try
    { Аварийный выход из функции, если указатель на объект lpdobj
      равен значению nil. }
    if lpdobj = nil then
    begin
      Result := E_FAIL;
      Exit;
    end;
    with FE do
    begin
      cfFormat := CF_HDROP;
```

```
      ptd := nil;
      dwAspect := DVASPECT_CONTENT;
      lindex := -1;
      tymed := TYMED_HGLOBAL;
    end;
    { Передаем данные, на которые ссылается указатель типа IDataObject,
      на носитель данных типа HGLOBAL в формате CF_HDROP }
    Result := lpdobj.GetData(FE, Medium);
    if Failed(Result) then Exit;
    try
      { Если выбран только один файл, считываем его имя и сохраняем
        в поле szFile. В противном случае отмечаем неудачное
        завершение работы функции. }
      if DragQueryFile(Medium.hGlobal, $FFFFFFFF, nil, 0) = 1 then
      begin
        DragQueryFile(Medium.hGlobal, 0, FFileName, SizeOf(FFileName));
        Result := NOERROR;
      end
      else
        Result := E_FAIL;
    finally
      ReleaseStgMedium(medium);
    end;
  except
    Result := E_UNEXPECTED;
  end;
end;
```

Интерфейс IContextMenu

Интерфейс IContextMenu используется для управления всплывающим меню, связанным с данным файлом в системной оболочке. Этот интерфейс также определен в модуле ShlObj следующим образом:

```
type
  IContextMenu = interface(IUnknown)
    ['{000214E4-0000-0000-C000-000000000046}']
    function QueryContextMenu(Menu: HMENU; indexMenu,
      idCmdFirst, idCmdLast, uFlags: UINT): HResult; stdcall;
    function InvokeCommand(var lpici: TCMInvokeCommandInfo): HResult; stdcall;
    function GetCommandString(idCmd, uType: UINT; pwReserved:
      PUINT; pszName: LPSTR; cchMax: UINT): HResult; stdcall;
  end;
```

После инициализации обработчика через интерфейс IShellExtInit вызывается метод IContextMenu.QueryContextMenu(). В список параметров, передаваемых методу, входит обработчик меню, индекс, соответствующий первому пункту меню, минимальное и максимальное значения идентификаторов команд меню, а также флаги атрибутов меню. Ниже приведена реализация метода в объекте TContextMenu. Этот метод используется для добавления ко-

манды меню Package info в обработчик меню, передаваемый с помощью параметра Menu (заметьте, что значение, возвращаемое функцией QueryContextMenu(), на единицу больше индекса команды меню, введенной последней).

```
function TContextMenu.QueryContextMenu(Menu: HMENU; indexMenu,
  idCmdFirst, idCmdLast, uFlags: UINT): HResult;
begin
  FMenuIdx := indexMenu;
  // Добавляем в контекстное меню одну команду
  InsertMenu (Menu, FMenuIdx, MF_STRING or MF_BYPOSITION,
    idCmdFirst, 'Package Info...');
  { Возвращаем значение, на 1 большее значения индекса
    последней введенной команды }
  Result := FMenuIdx + 1;
end;
```

Следующий вызываемый оболочкой метод — GetCommandString() — предназначен для получения независимой от языка командной строки или справочной строки для конкретной команды меню. В список параметров, передаваемых методу, входит смещение команды меню, флаги, указывающие на тип получаемой информации, зарезервированный параметр, строковый буфер и его размер. Приведенная ниже реализация этого метода в объекте TContextMenu отображает справочную строку для данной команды меню.

```
function TContextMenu.GetCommandString (idCmd, uType: UINT; pwReserved: PUINT;
  pszName: LPSTR; cchMax: UINT): HRESULT;
begin
  Result := S_OK;
  try
    { Проверяем, что индекс меню верен и что оболочка
      запрашивает справочную строку. }
    if (idCmd = FMenuIdx) and ((uType and GCS_HELPTEXT) <> 0) then
      { Возвращаем справочную строку для данной команды меню }
      StrLCopy(pszName, 'Получаем информацию для выбранного пакета.', cchMax)
    else
      Result := E_INVALIDARG;
  except
    Result := E_UNEXPECTED;
  end;
end;
```

Метод InvokeCommand() вызывается каждый раз при щелчке на новой команде меню. В качестве параметра этому методу передается запись TCMInvokeCommandInfo, которая определена в модуле ShlObj следующим образом:

```
type
  PCMInvokeCommandInfo = ^TCMInvokeCommandInfo;
  TCMInvokeCommandInfo = packed record
    cbSize: DWORD;
     { Должно иметь значение SizeOf(TCMInvokeCommandInfo). }
    fMask: DWORD;
     { Любая комбинация CMIC_MASK_* }
    hwnd: HWND;
```

```
    { Может принимать значение NULL, означающее отсутствие окна-владельца). }
   lpVerb: LPCSTR;
    { Любая строка AKEINTRESOURCE(idOffset). }
   lpParameters: LPCSTR;
    { Может иметь значение NULL (означает отсутствие параметра). }
   lpDirectory: LPCSTR;
    { Может иметь значение NULL (означает отсутствие заданной папки). }
   nShow: Integer;
    { Одно из значений SW_ API-функции ShowWindow(). }
   dwHotKey: DWORD;
   hIcon: THandle;
  end;
```

Младшее слово поля lpVerb будет содержать индекс выбранной команды меню. Ниже приведена реализация этого метода.

```
function TContextMenu.InvokeCommand (var lpici: TCMInvokeCommandInfo):
  HResult;
begin
  Result := S_OK;
  try
    // Проверяем, отсутствует ли вызов из приложения
    if HiWord(Integer(lpici.lpVerb)) <> 0 then
    begin
      Result := E_FAIL;
      Exit;
    end;
    { Выполняем команду, заданную в поле lpici.lpVerb. Возвращаем
      значение E_INVALIDARG, если получен неверный номер аргумента. }
    if LoWord(lpici.lpVerb) = FMenuIdx then
      ExecutePackInfoApp(FFileName, lpici.hwnd)
    else
      Result := E_INVALIDARG;
  except
    MessageBox(lpici.hwnd, 'Ошибка при получении информации о пакете.',
      'Error', MB_OK or MB_ICONERROR);
    Result := E_FAIL;
  end;
end;
```

Если все проходит успешно, то с помощью функции ExecutePackInfoApp() вызывается приложение PackInfo.exe, отображающее разного рода информацию о пакете. Здесь мы не будем останавливаться на особенностях этого приложения; оно детально рассмотрено в главе 13, “Дополнительный инструментарий разработчика” (I том).

Регистрация

В системном реестре обработчики контекстного меню должны регистрироваться в следующей ветви:

HKEY_CLASSES_ROOT\<*тип файла*>\shellex\ContextMenuHandlers

Как и в случае расширения перемещения, возможность регистрации DLL контекстного меню реализована с помощью потомка объекта `TComObject`. Код модуля, содержащего объект `TContextMenu`, приведен в листинге 24.10. На рис. 24.9 показано локальное меню для `.bpl`-файла с новой командой, а на рис. 24.10 представлен вид окна программы `PackInfo.exe`, вызываемой обработчиком контекстного меню.

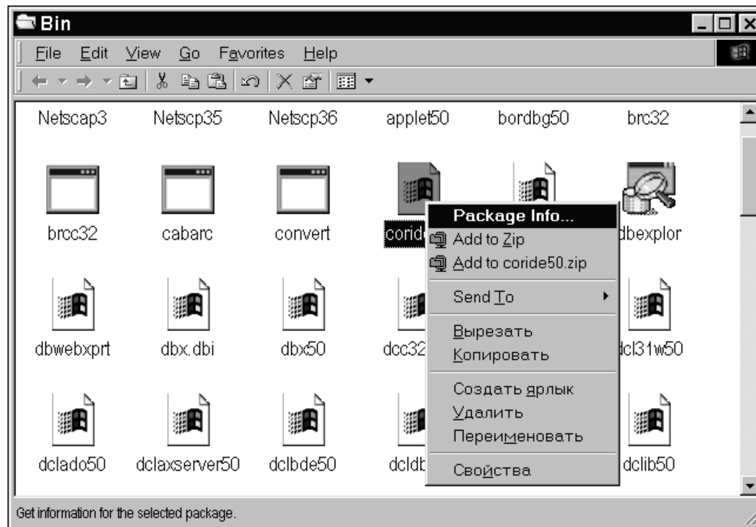


Рис. 24.9. Обработчик контекстного меню в действии

Рис. 24.10. Получение информации о пакете с помощью обработчика контекстного меню

Листинг 24.10. Модуль `ContMain.pas` — главный модуль реализации обработчика контекстного меню

```
unit ContMain;

interface

uses Windows, ComObj, ShlObj, ActiveX;

type
  TContextMenu = class(TComObject, IContextMenu, IShellExtInit)
  private
    FFileName: array[0..MAX_PATH] of char;
    FMenuIdx: UINT;
  protected
    // Методы интерфейса IContextMenu
    function QueryContextMenu(Menu: HMENU; indexMenu, idCmdFirst,
      idCmdLast, uFlags: UINT): HResult; stdcall;
    function InvokeCommand(var lpici: TCMInvokeCommandInfo): HResult; stdcall;
    function GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
      pszName: LPSTR; cchMax: UINT): HResult; stdcall;
    // Метод интерфейса IShellExtInit
    function Initialize(pidlFolder: PItemIDList; lpdobj: IDataObject;
      hKeyProgID: HKEY): HResult; reintroduce; stdcall;
  end;

  TContextMenuFactory = class(TComObjectFactory)
  protected
    function GetProgID: string; override;
    procedure ApproveShellExtension(Register: Boolean; const ClsID: string);
      virtual;
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;

implementation

uses ComServ, SysUtils, ShellAPI, Registry;

procedure ExecutePackInfoApp(const FileName: string; ParentWnd: HWND);
const
  SPackInfoApp = '%sPackInfo.exe';
  SCmdLine = '"%s" %s';
  SErrorStr = 'Failed to execute PackInfo:'#13#10#13#10;
var
  PI: TProcessInformation;
  SI: TStartupInfo;
  ExeName, ExeCmdLine: string;
  Buffer: array[0..MAX_PATH] of char;
begin
```

```
  { Получаем папку данной DLL. Предполагаем, что выполняемая
    EXE-программа находится в этой же папке. }
  GetModuleFileName(HInstance, Buffer, SizeOf(Buffer));
  ExeName := Format(SPackInfoApp, [ExtractFilePath(Buffer)]);
  ExeCmdLine := Format(SCmdLine, [ExeName, FileName]);
    FillChar(SI, SizeOf(SI), 0);
  SI.cb := SizeOf(SI);
  if not CreateProcess(PChar(ExeName), PChar(ExeCmdLine), nil, nil, False,
    0, nil, nil, SI, PI) then
    MessageBox(ParentWnd, PChar(SErrorStr + SysErrorMessage
      (GetLastError)), 'Error', MB_OK or MB_ICONERROR);
end;

{ TContextMenu }

{ TContextMenu.IContextMenu }

function TContextMenu.QueryContextMenu(Menu: HMENU; indexMenu,
  idCmdFirst, idCmdLast, uFlags: UINT): HResult;
begin
  FMenuIdx := indexMenu;
  { Добавляем к контекстному меню одну команду. }
  InsertMenu (Menu, FMenuIdx, MF_STRING or MF_BYPOSITION, idCmdFirst,
    'Package Info...');
  { Возвращаем увеличенное на единицу значение индекса последней
    добавленной в меню команды. }
  Result := FMenuIdx + 1;
end;

function TContextMenu.InvokeCommand(var lpici: TCMInvokeCommandInfo):
  HResult;
begin
  Result := S_OK;
  try
    // Проверяем, нет ли вызова из приложения
  if HiWord(Integer(lpici.lpVerb)) <> 0 then
    begin
      Result := E_FAIL;
      Exit;
    end;
    { Выполняем команду, заданную в поле lpici.lpVerb. Возвращаем значение
      E_INVALIDARG, если был передан неверный номер параметра. }
    if LoWord(lpici.lpVerb) = FMenuIdx then
      ExecutePackInfoApp(FFileName, lpici.hwnd)
    else
      Result := E_INVALIDARG;
  except
    MessageBox(lpici.hwnd, 'Ошибка при получении информации о пакете.',
      'Error', MB_OK or MB_ICONERROR); Result := E_FAIL;
  end;
```

```
end;

function TContextMenu.GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
  pszName: LPSTR; cchMax: UINT): HRESULT;
begin
  Result := S_OK;
  try
    { Проверяем корректность переданного индекса, а также факт
      запроса оболочкой справочной строки. }
    if (idCmd = FMenuIdx) and ((uType and GCS_HELPTEXT) <> 0) then
      { Возвращаем справочную строку для пункта меню. }
      StrLCopy(pszName, 'Get information for the selected package.', cchMax)
    else
      Result := E_INVALIDARG;
  except
    Result := E_UNEXPECTED;
  end;
end;

{ TContextMenu.IShellExtInit }

function TContextMenu.Initialize(pidlFolder: PItemIDList; lpdobj: IDataObject;
  hKeyProgID: HKEY): HResult;
var
  Medium: TStgMedium;
  FE: TFormatEtc;
begin
  try
    // Аварийный выход из функции, если lpdobj=nil
    if lpdobj = nil then
    begin
      Result := E_FAIL;
      Exit;
    end;
    with FE do
    begin
      cfFormat := CF_HDROP;
      ptd := nil;
      dwAspect := DVASPECT_CONTENT;
      lindex := -1;
      tymed := TYMED_HGLOBAL;
    end;
{ Передаем данные, на которые ссылается указатель типа IDataObject,
  на носитель данных типа HGLOBAL в формате CF_HDROP }
    Result := lpdobj.GetData(FE, Medium);
    if Failed(Result) then Exit;
    try
      { Если выбран только один файл, считываем его имя и сохраняем в поле
        szFile. В противном случае выход из функции с указанием ошибки. }
      if DragQueryFile(Medium.hGlobal, $FFFFFFFF, nil, 0) = 1 then
```

```
      begin
        DragQueryFile(Medium.hGlobal, 0, FFileName, SizeOf(FFileName));
        Result := NOERROR;
      end
      else
        Result := E_FAIL;
    finally
      ReleaseStgMedium(medium);
    end;
  except
    Result := E_UNEXPECTED;
  end;
end;

{ TContextMenuFactory }

function TContextMenuFactory.GetProgID: string;
begin
  { Для расширения оболочки, управляющего работой контекстного меню,
    идентификатор программы ProgID не требуется. }
  Result := '';
end;

procedure TContextMenuFactory.UpdateRegistry(Register: Boolean);
var
  ClsID: string;
begin
  ClsID := GUIDToString(ClassID);
  inherited UpdateRegistry(Register);
  ApproveShellExtension(Register, ClsID);
  if Register then
  begin
    // Нужно зарегистрировать тип файла .bpl
    CreateRegKey('.bpl', '', 'DelphiPackageLibrary');
    { Регистрация данной DLL в качестве обработчика
      контекстного меню для .bpl-файлов. }
      CreateRegKey('BorlandPackageLibrary\shellex\ContextMenuHandlers\' +
      ClassName, '', ClsID);
  end
  else begin
    DeleteRegKey('.bpl');
    DeleteRegKey('BorlandPackageLibrary\shellex\ContextMenuHandlers\' +
      ClassName);
  end;
end;

procedure TContextMenuFactory.ApproveShellExtension(Register: Boolean; const
ClsID: string);
{ Эту точку входа системного реестра необходимо заполнить для того,
  чтобы расширение работало корректно под управлением Windows NT.}
```

```
const
  SApproveKey =
    'SOFTWARE\Microsoft\Windows\CurrentVersion\Shell Extensions\Approved';
begin
  with TRegistry.Create do
    try
      RootKey := HKEY_LOCAL_MACHINE;
      if not OpenKey(SApproveKey, True) then Exit;
      if Register then WriteString(ClsID, Description)
      else DeleteValue(ClsID);
    finally
      Free;
    end;
end;

const
  CLSID_CopyHook: TGUID = '{7C5E74A0-D5E0-11D0-A9BF-E886A83B9BE5}';

initialization
  TContextMenuFactory.Create(ComServer, TContextMenu, CLSID_CopyHook,
    'D4DG_ContextMenu', 'D4DG Context Menu Shell Extension Example',
    ciMultiInstance, tmApartment);
end.
```

Обработчики пиктограмм

Обработчики пиктограмм позволяют вызывать разные пиктограммы для использования их в различных экземплярах файлов одного типа. В данном примере описан объект обработчика пиктограмм `TIconHandler`, который обеспечивает различные пиктограммы для разных типов пакетных файлов Borland Package (BPL). В зависимости от типа пакета — времени выполнения, времени разработки, универсальный или какой-либо другой — при отображении этих файлов в папке оболочки будут использоваться различные пиктограммы.

Флаги пакета

Прежде чем представить интерфейсы, необходимые для реализации данного расширения оболочки, рассмотрим метод, используемый для определения типа данного пакетного файла. Этот метод возвращает объект типа `TPackType`, который определяется следующим образом:

```
TPackType = (ptDesign, ptDesignRun, ptNone, ptRun);
```

Приведем код метода:

```
function TIconHandler.GetPackageType: TPackType;
var
  PackMod: HMODULE;
  PackFlags: Integer;
begin
  { Поскольку необходимо получить доступ только к ресурсам пакета,
```

```
    используем функцию LoadLibraryEx с параметром LOAD_LIBRARY_AS_DATAFILE,
    обеспечивающую эффективные средства загрузки пакета. }
  PackMod := LoadLibraryEx(PChar(FFileName), 0, LOAD_LIBRARY_AS_DATAFILE);
  if PackMod = 0 then
  begin
    Result := ptNone;
    Exit;
  end;
  try
    GetPackageInfo(PackMod, nil, PackFlags, PackInfoProc);
  finally
    FreeLibrary(PackMod);
  end;
  { Отключаем маскирование всех флагов, кроме флага времени разработки
    и флага времени выполнения, и возвращаем результат. }
  case PackFlags and (pfDesignOnly or pfRunOnly) of
    pfDesignOnly: Result := ptDesign;
    pfRunOnly: Result := ptRun;
    pfDesignOnly or pfRunOnly: Result := ptDesignRun;
  else
    Result := ptNone;
  end;
end;
```

Работа этого метода заключается в вызове метода GetPackageInfo() из модуля SysUtils для получения флагов пакета. Отметим, что в целях оптимизации производительности вместо встроенной процедуры Delphi LoadPackage() для загрузки библиотеки пакета вызывается функция API LoadLibraryEx(). Внутри функции LoadPackage() содержится вызов API-функции LoadLibrary(), которая загружает BPL-библиотеку. После загрузки библиотеки вызывается функция InitializePackage(), выполняющая код инициализации каждого модуля в пакете. Но поскольку в данном случае нужно лишь получить флаги пакета, которые хранятся в файле ресурсов, связанном с BPL-библиотекой, вполне можно обойтись загрузкой пакета путем вызова функции LoadLibraryEx() с заданием флага LOAD_LIBRARY_AS_DATAFILE.

Интерфейсы обработчика пиктограмм

Как уже упоминалось выше в этой главе, обработчики пиктограмм должны поддерживать оба интерфейса — и IExtractIcon (определен в модуле ShlObj), и IPersistFile (определен в модуле ActiveX). Определения этих интерфейсов имеют следующий вид:

```
type
  IExtractIcon = interface(IUnknown)
    ['{000214EB-0000-0000-C000-000000000046}']
    function GetIconLocation(uFlags: UINT; szIconFile: PAnsiChar; cchMax: UINT;
      out piIndex: Integer; out pwFlags: UINT): HResult; stdcall;
    function Extract(pszFile: PAnsiChar; nIconIndex: UINT; out phiconLarge,
      phiconSmall: HICON; nIconSize: UINT): HResult; stdcall;
  end;

  IPersistFile = interface(IPersist)
```

```
    ['{0000010B-0000-0000-C000-000000000046}']
    function IsDirty: HResult; stdcall;
    function Load(pszFileName: POleStr; dwMode: Longint): HResult; stdcall;
    function Save(pszFileName: POleStr; fRemember: BOOL): HResult; stdcall;
    function SaveCompleted(pszFileName: POleStr): HResult; stdcall;
    function GetCurFile(out pszFileName: POleStr): HResult; stdcall;
  end;
```

Хотя на первый взгляд кажется, что эти интерфейсы выполняют значительный объем работы, уверяем вас, что это впечатление обманчиво: на самом деле только два из приведенных выше методов должны быть реализованы. Первый из них — метод `IPersistFileLoad()`. Этот метод вызывается для инициализации расширения оболочки, и внутри него нужно предусмотреть сохранение имени файла, передаваемого посредством параметра `pszFileName`. Ниже приведена реализация этого метода в объекте `TExtractIcon`:

```
function TIconHandler.Load(pszFileName: POleStr; dwMode: Longint): HResult;
begin
  { Этот метод вызывается для инициализированного обработчика пиктограмм.
    Нужно сохранить имя файла, которое передано в параметре pszFileName. }
  FFileName := pszFileName;
  Result := S_OK;
end;
```

Второй метод, который нужно реализовать — `IExtractIcon.GetIconLocation()`. Передаваемые методу параметры рассматриваются ниже в этой главе.

Параметр `uFlags` указывает на тип отображаемой пиктограммы. Этот параметр может принимать значение `0`, `GIL_FORSHELL` или `GIL_OPENICON`. Значение `GIL_FORSHELL` говорит о том, что пиктограмма будет отображаться в папке системной оболочки. Значение `GIL_OPENICON` указывает, что пиктограмма должна быть в "открытом" состоянии, если доступны изображения как для открытого, так и для закрытого состояний. Если этот флаг не задан, то пиктограмма должна быть в нормальном, т.е. "закрытом", состоянии. Этот флаг обычно используется для объектов папки.

Параметр `szIconFile` представляет собой буфер, в который передается информация о расположении пиктограммы. Параметр `cchMax` содержит размер этого буфера. Параметр `piIndex` — переменная целого типа — получает индекс пиктограммы, уточняющий ее расположение.

Параметр `pwFlags` может принимать нулевое или несколько других значений, приведенных в табл. 24.8.

Таблица 24.8. Значения параметра pwFlags функции GetIconLocation()

Флаг	Значение
`GIL_DONTCACHE`	Битовая карта этой пиктограммы не должна быть кэширована вызывающей процедурой. Важность этой детали продиктована тем, что в последующие версии оболочки может быть введен флаг `GIL_DONTCACHELOCATION`
`GIL_NOTFILENAME`	Местонахождение пиктограммы не описывается парой значений "имя файла/индекс". Для получения изображения пиктограммы вызывающие процедуры должны вызвать метод `IExtractIcon.Extract()`

Окончание табл. 24.8

Флаг	Значение
GIL_PERCLASS	Все объекты данного класса имеют одну и ту же пиктограмму. Этот флаг используется внутренними средствами оболочки. Обычные реализации интерфейса IExtractIcon не требуют этого флага, поскольку для случаев, когда одному объекту соответствует одна пиктограмма, обработчик пиктограммы не требуется. Для реализации пиктограмм типа "одна пиктограмма на один класс" рекомендуется просто зарегистрировать стандартную пиктограмму для данного класса
GIL_PERINSTANCE	Каждый объект этого класса имеет свою собственную пиктограмму. Этот флаг используется оболочкой в случаях применения файла установки setup.exe. В этих случаях оболочка может "знать" о нескольких объектах, имеющих идентичные имена, и использовать различные пиктограммы. Обычные реализации интерфейса IExtractIcon не требуют этого флага
GIL_SIMULATEDOC	Вызывающая процедура должна создать пиктограмму документа, используя указанную пиктограмму

Реализация метода GetIconLocation() в объекте TIconHandler выглядит следующим образом:

```
function TIconHandler.GetIconLocation(uFlags: UINT; szIconFile: PAnsiChar;
  cchMax: UINT; out piIndex: Integer; out pwFlags: UINT): HResult;
begin
  Result := S_OK;
  try
    { Для отыскания пиктограммы возвращаем библиотеку DLL данного модуля. }
    GetModuleFileName(HInstance, szIconFile, cchMax);
    { Сообщаем оболочке о том, что не нужно кэшировать битовые карты
      в случае изменения пиктограммы, и о том, что каждый
      экземпляр может иметь свою собственную пиктограмму. }
    pwFlags := GIL_DONTCACHE or GIL_PERINSTANCE;
    { Индекс пиктограммы согласуется с типом TPackType. }
    piIndex := Ord(GetPackageType);
  except
    { В случае ошибки используем пиктограмму по умолчанию. }
    piIndex := Ord(ptNone);
  end;
end;
```

Пиктограммы связываются с библиотекой DLL расширения оболочки в виде файла ресурсов, и поэтому имя текущего файла, возвращаемое функцией GetModuleFileName(), записывается в буфер szIconFile. Пиктограммы организованы таким образом, чтобы индекс пиктограммы для типа пакета соответствовал индексу типа пакета в перечислении типа TPackType. Поэтому значение, возвращаемое функцией GetPackageType(), присваивается параметру piIndex.

Регистрация

Обработчики пиктограмм должны быть зарегистрированы в системном реестре в следующей ветви:

HKEY_CLASSES_ROOT\\<*тип файла*>\shellex\IconHandler

Как и в случае с другими расширениями, для регистрации обработчика пиктограмм создается потомок класса `TComObjectFactory`. Код модуля обработчика пиктограмм, включающий код объекта `TComObjectFactory`, приведен в листинге 24.11, а на рис. 24.11 показана папка системной оболочки, содержащая пакеты различных типов. Обратите внимание на то, что для отображения различных типов пакетов используются различные пиктограммы.

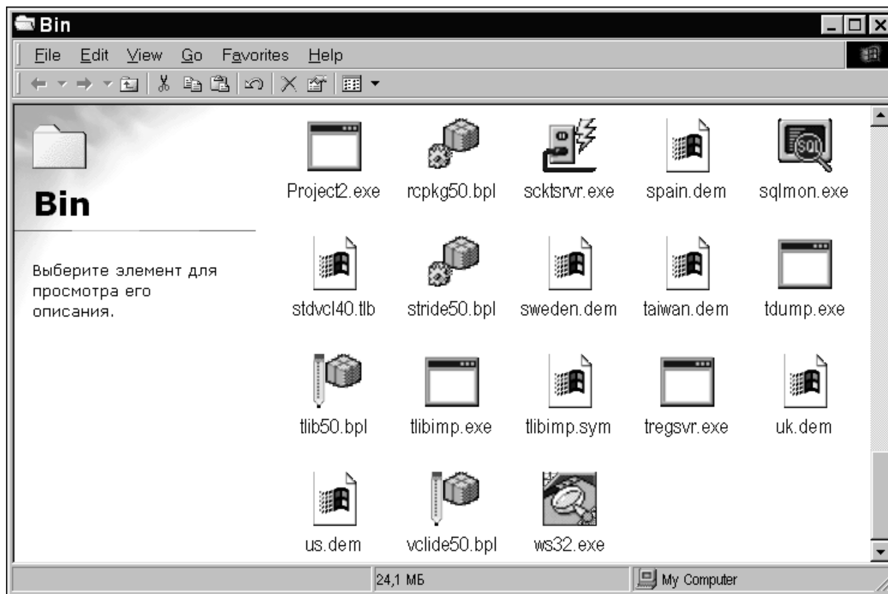


Рис. 24.11. Результат использования обработчика пиктограмм

Листинг 24.11. Модуль `IconMain.pas`, главный модуль реализации обработчика пиктограмм

```
unit IconMain;

interface

uses Windows, ActiveX, ComObj, ShlObj;

type
  TPackType = (ptDesign, ptDesignRun, ptNone, ptRun);

  TIconHandler = class(TComObject, IExtractIcon, IPersistFile)
  private
    FFileName: string;
    function GetPackageType: TPackType;
  protected
    // Методы интерфейса IExtractIcon
    function GetIconLocation(uFlags: UINT; szIconFile: PAnsiChar; cchMax: UINT;
       out piIndex: Integer; out pwFlags: UINT): HResult; stdcall;
    function Extract(pszFile: PAnsiChar; nIconIndex: UINT; out phiconLarge,
      phiconSmall: HICON; nIconSize: UINT): HResult; stdcall;
```

```
    // Метод интерфейса IPersist
    function GetClassID(out classID: TCLSID): HResult; stdcall;
    // Методы интерфейса IPersistFile
    function IsDirty: HResult; stdcall;
    function Load(pszFileName: POleStr; dwMode: Longint): HResult; stdcall;
    function Save(pszFileName: POleStr; fRemember: BOOL): HResult; stdcall;
    function SaveCompleted(pszFileName: POleStr): HResult; stdcall;
    function GetCurFile(out pszFileName: POleStr): HResult; stdcall;
  end;

  TIconHandlerFactory = class(TComObjectFactory)
  protected
    function GetProgID: string; override;
    procedure ApproveShellExtension(Register: Boolean; const ClsID: string);
      virtual;
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;

implementation

uses SysUtils, ComServ, Registry;

{ TIconHandler }
procedure PackInfoProc(const Name: string; NameType: TNameType;
  Flags: Byte; Param: Pointer);
begin
  { Нет необходимости в реализации данной процедуры, поскольку
    нас интересуют только флаги пакета, а не модули. }
end;

function TIconHandler.GetPackageType: TPackType;
var
  PackMod: HMODULE;
  PackFlags: Integer;
begin
  { Поскольку необходимо получить доступ только к ресурсам пакета, то
    используется функция LoadLibraryEx с параметром LOAD_LIBRARY_AS_DATAFILE,
    обеспечивающая эффективные средства для загрузки пакета. }
  PackMod := LoadLibraryEx(PChar(FFileName), 0, LOAD_LIBRARY_AS_DATAFILE);
  if PackMod = 0 then
  begin
    Result := ptNone;
    Exit;
  end;
  try
    GetPackageInfo(PackMod, nil, PackFlags, PackInfoProc);
  finally
    FreeLibrary(PackMod);
  end;
```

```
  { Отключаем маскирование всех флагов, кроме флагов времени
    разработки и времени выполнения, и возвращаем результат. }
  case PackFlags and (pfDesignOnly or pfRunOnly) of
    pfDesignOnly: Result := ptDesign;
    pfRunOnly: Result := ptRun;
    pfDesignOnly or pfRunOnly: Result := ptDesignRun;
  else
    Result := ptNone;
  end;
end;

{ TIconHandler.IExtractIcon }

function TIconHandler.GetIconLocation(uFlags: UINT; szIconFile: PAnsiChar;
  cchMax: UINT; out piIndex: Integer; out pwFlags: UINT): HResult;
begin
  Result := S_OK;
  try
    { Возвращаем эту библиотеку DLL для имени модуля, чтобы найти пиктограмму. }
    GetModuleFileName(HInstance, szIconFile, cchMax);
    { Сообщаем оболочке о том, что не нужно кэшировать битовые карты
      в случае изменения пиктограммы, и о том, что каждый
      экземпляр может иметь свою собственную пиктограмму. }
    pwFlags := GIL_DONTCACHE or GIL_PERINSTANCE;
    { Индекс пиктограммы согласуется с типом TPackType. }
    piIndex := Ord(GetPackageType);
  except
    { В случае ошибки используем стандартную пиктограмму. }
    piIndex := Ord(ptNone);
  end;
end;

function TIconHandler.Extract(pszFile: PAnsiChar; nIconIndex: UINT;
  out phiconLarge, phiconSmall: HICON; nIconSize: UINT): HResult;
begin
  { Этот метод нужно реализовывать только в случае, если пиктограмма
    сохранена в каком-то пользовательском формате. Поскольку данная
    пиктограмма содержится в старой DLL, возвращается значение S_FALSE. }
  Result := S_FALSE;
end;

{ TIconHandler.IPersist }

function TIconHandler.GetClassID(out classID: TCLSID): HResult;
begin
  { Данный метод для обработчиков пиктограмм не вызывается. }
  Result := E_NOTIMPL;
end;

{ TIconHandler.IPersistFile }
```

```
function TIconHandler.IsDirty: HResult;
begin
  { Данный метод для обработчиков пиктограмм не вызывается. }
  Result := S_FALSE;
end;

function TIconHandler.Load(pszFileName: POleStr; dwMode: Longint): HResult;
begin
  { Этот метод вызывается для инициализированного обработчика пиктограмм. Нужно
    сохранить имя файла, которое передается посредством параметра pszFileName. }
  FFileName := pszFileName;
  Result := S_OK;
end;

function TIconHandler.Save(pszFileName: POleStr; fRemember: BOOL): HResult;
begin
  { Данный метод для обработчиков пиктограмм не вызывается. }
  Result := E_NOTIMPL;
end;
function TIconHandler.SaveCompleted(pszFileName: POleStr): HResult;
begin
  { Данный метод для обработчиков пиктограмм не вызывается. }
  Result := E_NOTIMPL;
end;

function TIconHandler.GetCurFile(out pszFileName: POleStr): HResult;
begin
  { Данный метод для обработчиков пиктограмм не вызывается. }
  Result := E_NOTIMPL;
end;

{ TIconHandlerFactory }

function TIconHandlerFactory.GetProgID: string;
begin
  { Для расширений контекстного меню идентификатор
    программы ProgID не требуется. }
  Result := '';
end;

procedure TIconHandlerFactory.UpdateRegistry(Register: Boolean);
var
  ClsID: string;
begin
  ClsID := GUIDToString(ClassID);
  inherited UpdateRegistry(Register);
  ApproveShellExtension(Register, ClsID);
  if Register then
  begin
    { Нужно зарегистрировать новый тип файла .bpl }
```

```
    CreateRegKey('.bpl', '', 'BorlandPackageLibrary');
    { Регистрируем эту библиотеку DLL как обработчик
      пиктограмм для .bpl-файлов. }
    CreateRegKey('BorlandPackageLibrary\shellex\IconHandler', '', ClsID);
  end
  else begin
    DeleteRegKey('.bpl');
    DeleteRegKey('BorlandPackageLibrary\shellex\IconHandler');
  end;
end;

procedure TIconHandlerFactory.ApproveShellExtension(Register: Boolean;
  const ClsID: string);
{ Эту точку входа системного реестра необходимо заполнить для того,
  чтобы расширение работало корректно под управлением Windows NT. }
const
  SApproveKey =
    'SOFTWARE\Microsoft\Windows\CurrentVersion\Shell Extensions\Approved';
begin
  with TRegistry.Create do
    try
      RootKey := HKEY_LOCAL_MACHINE;
      if not OpenKey(SApproveKey, True) then Exit;
      if Register then WriteString(ClsID, Description)
      else DeleteValue(ClsID);
    finally
      Free;
    end;
end;

const
  CLSID_IconHandler: TGUID = '{ED6D2F60-DA7C-11D0-A9BF-90D146FC32B3}';

initialization
  TIconHandlerFactory.Create(ComServer, TIconHandler, CLSID_IconHandler,
     'D4DG_IconHandler', 'D4DG Icon Handler Shell Extension Example',
      ciMultiInstance, tmApartment);
end.
```

Резюме

В этой главе рассмотрены различные аспекты расширения оболочки Windows: помещение пиктограмм в индикаторную область панели задач, создание панели инструментов рабочего стола системы (AppBars), работа с ярлыками и разработка расширений оболочки Windows различных типов. Материал данной главы основывается на сведениях, представленных в предыдущей главе, — прежде всего относящихся к объектам COM и ActiveX. Эти знания вам понадобятся также при изучении разработки элементов управления ActiveX.

Создание элементов управления ActiveX

Глава

25

Зачем создавать элементы управления ActiveX 388

Создание элемента управления ActiveX 388

Активные формы `ActiveForm` 432

Добавление свойств к формам ActiveForm 432

Элементы управления ActiveX в Web 439

Резюме 453

По мнению многих разработчиков, простота создания элементов управления ActiveX — одно из замечательнейших свойств Delphi. ActiveX — это стандарт программирования независимых от языка элементов управления, которые могут функционировать в разнообразных средах, включая Delphi, C++Builder, Visual Basic и Internet Explorer. Эти элементы управления могут быть как простыми (например, статический элемент управления текстом), так и произвольно сложными (например полнофункциональная электронная таблица или текстовый процессор). Традиционно, создание элементов управления ActiveX считается достаточно сложной и запутанной задачей, однако Delphi значительно облегчает этот процесс, позволяя преобразовывать относительно простые в создании компоненты или формы библиотеки VCL в элементы управления ActiveX.

В этой главе нет подробного описания всего, что связано с элементами управления ActiveX, — это тема для отдельной книги. Здесь лишь показано, как создавать элементы управления ActiveX в Delphi и как использовать мастера и инструменты среды разработки Delphi, чтобы заставить созданные элементы управления работать в ваших приложениях.

На заметку

Возможность создания элементов управления ActiveX поддерживается лишь версиями Delphi Professional и Enterprise Editions.

Зачем создавать элементы управления ActiveX

Delphi предоставляет в распоряжение разработчиков такое множество уже готовых разнообразных компонентов и форм в библиотеке VCL, что вполне закономерно возникает вопрос: "А зачем вообще может понадобиться создавать какие-то элементы управления ActiveX?" На то есть несколько причин. Во-первых, если вы — профессиональный разработчик компонентов, преимущества могут оказаться просто ошеломляющими: после преобразования созданных компонентов VCL в элементы управления ActiveX ваш потенциальный рынок сбыта распространится не только на Delphi и C++Builder, но и на практически любые инструменты разработки для среды Win32. А, во-вторых, даже если вы не являетесь профессиональным разработчиком компонентов, то можете использовать все преимущества элементов управления ActiveX для придания дополнительной содержательности и функциональности создаваемым вами Web-страницам.

Создание элемента управления ActiveX

Одношаговые мастера в составе Delphi существенно упрощают создание элементов управления ActiveX. Однако, как будет показано ниже в этой главе, если элементу управления требуется придать достойный вид, то вызов мастера — это только начало работы.

Знакомство с возможностями создания элементов управления ActiveX в Delphi лучше всего начать с показанной на рис. 25.1 вкладки ActiveX диалогового окна New Items, раскрывающегося при выборе команды File⇨New. Многие элементы, присутствующие на этой вкладке, будут описаны далее в этой главе.

Одна из пиктограмм этой вкладки представляет активную форму — ActiveForm (этот тип элементов управления будет рассмотрен ниже). По двойному щелчку на этой пиктограмме запускается мастер, который поможет вам создать активную форму. Следует заметить, что активные формы лишь незначительно отличаются от обычных элементов управления ActiveX, так что оба типа элементов управления далее в этой главе будут называться просто — *элемент управления ActiveX*.

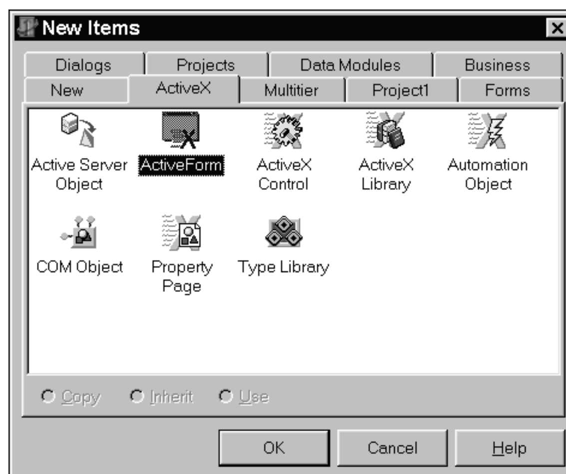


Рис. 25.1. Вкладка ActiveX диалогового окна New Items

По двойному щелчку на следующей пиктограмме, ActiveX Control, запускается мастер ActiveX Control Wizard, который будет описан ниже.

Далее следует пиктограмма ActiveX Library. Дважды щелкните на этой пиктограмме, если требуется создать проект новой библиотеки, экспортирующий четыре функции сервера ActiveX, описанные в главе 23, "COM-ориентированные технологии". Эту операцию можно использовать в качестве отправной точки перед добавлением в проект элемента управления ActiveX.

Пиктограмма Automation Object обеспечивает доступ еще к одному мастеру Delphi, который описан в главе 23.

По двойному щелчку на следующей пиктограмме будет запущен мастер COM Object Wizard, с помощью которого можно создать простой COM-объект. Работа этого мастера описывалась в предыдущей главе.

По двойному щелчку на пиктограмме Property Page к текущему проекту будет добавлена *страница свойств*, с помощью которой элемент управления ActiveX можно редактировать визуально. Пример создания страницы свойств элемента управления ActiveX и ее объединения с проектом будет рассмотрен далее в этой главе.

И, наконец, последняя пиктограмма — Type Library — позволяет добавить к проекту библиотеку типов. Поскольку при использовании мастеров создания элемента управления ActiveX и активной формы (также как и объектов автоматизации) библиотека типов добавляется к проекту автоматически, обращаться к этой пиктограмме придется нечасто.

Мастер создания элемента управления ActiveX

По двойному щелчку на пиктограмме ActiveX Control Wizard, расположенной во вкладке ActiveX диалогового окна New Items, на экране раскрывается диалоговое окно ActiveX Control Wizard, показанное на рис. 25.2.

В этом окне можно выбрать класс компонента VCL, который будет инкапсулирован в элементе управления ActiveX. Можно ввести также имя класса элемента управления ActiveX, имя файла, в котором будет содержаться реализация нового элемента управления ActiveX, и имя проекта, который будет представлять новый элемент управления ActiveX.

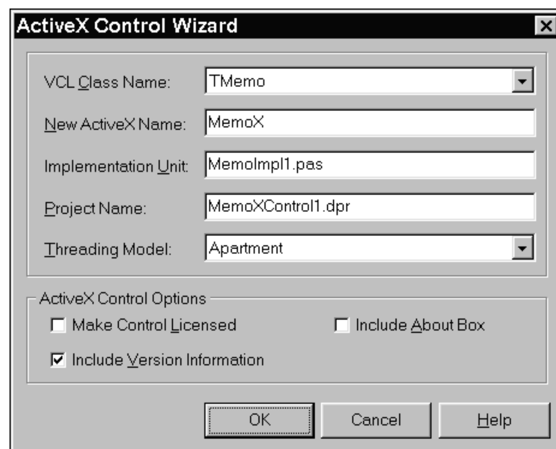


Рис. 25.2. Диалоговое окно ActiveX Control Wizard

Компоненты VCL в мастере элемента управления ActiveX

Просматривая элементы раскрывающегося списка VCL Class Name в окне ActiveX Control Wizard, можно заметить, что в нем содержатся не все существующие элементы управления библиотеки VCL. Для помещения в список этого мастера элемент управления VCL должен удовлетворять трем критериям.

- Содержаться в текущем установленном пакете разработки (и, таким образом, находиться в палитре компонентов).
- Быть потомком компонента `TWinControl`. В настоящее время не оконные элементы управления не могут быть инкапсулированы в элементы управления ActiveX.
- Не подлежать исключению из данного списка при выполнении процедуры `RegisterNonActiveX()`. Эта процедура подробно описана в справочной системе Delphi.

Многие стандартные компоненты VCL исключены из списка, поскольку они как элементы управления ActiveX либо не представляют интереса, либо требуют существенной доработки, превышающей возможности мастера, прежде чем их можно будет использовать в качестве элементов управления ActiveX. Компонент `TDBGrid` — хороший пример элемента управления VCL, который не имеет смысла использовать в качестве элемента управления ActiveX, поскольку для функционирования ему необходим экземпляр другого компонента VCL, `TDataSource` (описываемый как свойство класса `TDBGrid`), а его невозможно использовать как элемент ActiveX. Класс `TTreeView` — это пример элемента управления, требующего значительных изменений для инкапсуляции в элемент управления ActiveX, поскольку узлы компонента `TTreeView` очень сложно представляются в среде ActiveX.

На заметку

Несмотря на то, что мастер ActiveX Control Wizard не позволяет автоматически генерировать элемент управления ActiveX из компонентов, не являющихся производными от класса `TWinControl`, можно создать такой элемент управления вручную, используя схему разработки Delphi ActiveX (DAX).

Параметры элемента управления ActiveX

В нижней части диалогового окна ActiveX Control Wizard можно установить определенные параметры, которые станут частью элемента управления ActiveX. Эти параметры задаются тремя флажками опций.

- Make Control Licensed (Сделать элемент управления лицензированным). Когда этот флажок опции установлен, вместе с проектом элемента управления будет сгенерирован файл лицензии (`.lic`-файл). Для того чтобы другие разработчики могли использовать созданный вами элемент управления ActiveX в собственной среде разработки, вместе с файлом элемента управления ActiveX (OCX) им потребуется получить и файл лицензии.
- Include Version Information (Включить сведения о версии). Если этот флажок опции установлен, то с файлом OCX будет связан ресурс `VersionInfo`. Строка информации о файле в ресурсе `VersionInfo` включает значение `OleSelfRegister`, устанавливаемое равным 1. Этот параметр необходим для некоторых старых платформ поддержки элементов управления ActiveX — например, Visual Basic 4.0. Изменить значение параметра `VersionInfo` проекта можно во вкладке Version Info диалогового окна Project Options.
- Include About Box (Включить окно About). Установите этот флажок опции, если с элементом управления ActiveX нужно связать диалоговое окно About. Это диалоговое окно обычно доступно в приложении-контейнере ActiveX при выборе соответствующей команды в контекстном меню, появляющемся по щелчку правой кнопкой мыши на элементе управления ActiveX. Диалоговое окно About представляет собой обычную форму Delphi, которую можно отредактировать по своему усмотрению.

Инкапсуляция элементов управления VCL

После установки всех параметров элемента управления в диалоговом окне ActiveX Control Wizard и щелчка на кнопке OK мастер приступает к инкапсуляции выбранного элемента управления VCL в элемент управления ActiveX. Результатом работы мастера является проект библиотеки ActiveX, которая включает рабочий элемент управления ActiveX, однако многие интересные действия оказываются скрытыми. Ниже описаны шаги, выполняемые при инкапсуляции элемента управления VCL в элемент управления ActiveX.

1. Мастер определяет, в каком модуле содержится элемент управления VCL. Этот модуль передается компилятору, который генерирует определенную символическую информацию о свойствах, методах и событиях элемента управления VCL.
2. Для проекта создается библиотека типов. В ней содержится интерфейс, включающий свойства и методы, диспинтерфейс, содержащий события, и компонентный класс, представляющий элемент управления ActiveX.
3. Мастер просматривает всю символическую информацию элемента управления VCL и добавляет соответствующие случаю свойства и методы в помещенный в библиотеку типов интерфейс, а также подходящие события, помещаемые в диспинтерфейс.

На заметку

При знакомстве с третьим этапом у читателя, скорее всего, возникнет вопрос: “Что собой представляет свойство, метод или событие, *подходящее* для включения в библиотеку типов?” Для включения в библиотеку типов свойства, параметры и возвращаемые значения методов и событий должны иметь тип, совместимый с автоматизацией. В главе 23, “COM-ориентированные технологии”, отмечалось, что средства автоматизации поддерживают следующие типы: `Byte`, `SmallInt`, `Integer`, `Single`, `Double`, `Currency`, `TDateTime`, `WideString`, `WordBool`, `PSafeArray`, `TDecimal`, `OleVariant`, `IUnknown`, `IDispatch`.

На заметку

Однако из этого правила есть исключения. Кроме типов, совместимых с автоматизацией, разрешены также параметры типов `TStrings`, `TPicture` и `TFont`. Для этих типов мастер будет использовать специальные промежуточные объекты, которые позволят им быть инкапсулированными в совместимые с ActiveX диспинтерфейсы или интерфейсы `IDispatch`.

4. После добавления всех подходящих свойств, методов и событий редактор библиотеки типов генерирует файл, представляющий собой результат трансляции содержимого библиотеки типов в Object Pascal.
5. Затем мастер генерирует файл реализации элемента управления ActiveX. В этом файле содержится объект `TActiveXControl`, в котором реализованы интерфейсы, описанные в библиотеке типов. Мастер автоматически создает *пересылающие методы* (forwarders) для свойств и методов интерфейса. Эти пересылающие методы направляют вызовы методов из элемента управления ActiveX в элемент управления VCL, а события — из элемента управления VCL в элемент управления ActiveX.

Все описанное иллюстрируется в приведенных ниже листингах. Они относятся к проекту элемента управления ActiveX, созданного на базе элемента управления VCL `TMemo`. Этот проект сохранен в файле `Memo.dpr`. В листинге 25.1 приведен код файла проекта, в листинге 25.2 — файл библиотеки типов, а в листинге 25.3 — файл реализации, сгенерированный для созданного элемента управления. Кроме того, на рис. 25.3 показано окно редактора библиотеки типов.

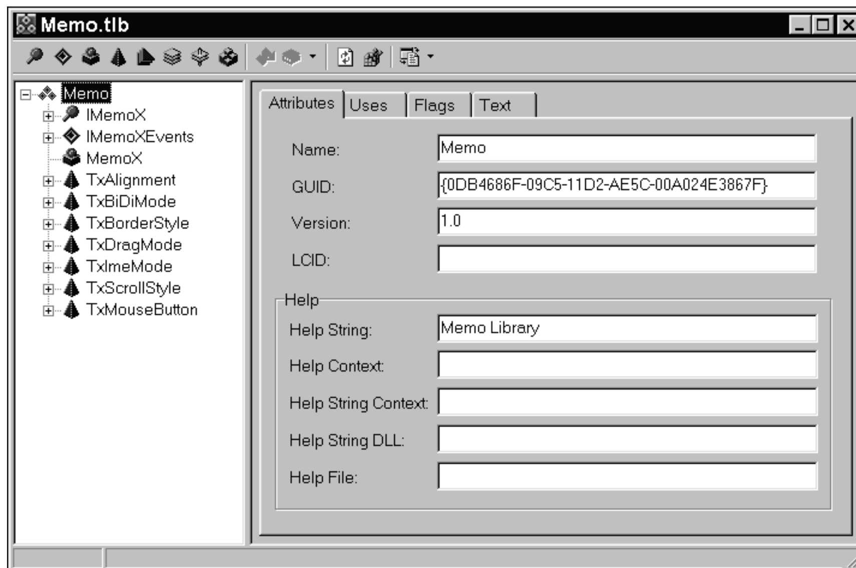


Рис. 25.3. Проект `Memo` в редакторе библиотеки типов

Листинг 25.1. Файл проекта `Memo.dpr`

```
library Memo;

uses
  ComServ,
  Memo_TLB in 'Memo_TLB.pas',
```

```
  MemoImpl in 'MemoImpl.pas' {MemoX: CoClass},
  About in 'About.pas' {MemoXAbout};

{$E ocx}

exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;

{$R *.TLB}

{$R *.RES}

begin
end.
```

Листинг 25.2. Файл библиотеки типов `Memo_TLB.pas`

```
unit Memo_TLB;

// ************************************************************************ //
// ВНИМАНИЕ
// --------
// Типы, объявленные в этом файле, были сгенерированы из данных библиотеки
// типа. Если эта библиотека типов явно или неявно (через ссылку на эту
// библиотеку через другую) будет реимпортирована или с помощью команды
// 'Refresh'в окне Type Library Editor активизирована во время редактирования
// библиотеки типов, содержимое файла будет сгенерировано повторно, а все
// внесенные в нее изменения будут утеряны
// ************************************************************************ //

// PASTLWTR : $Revision:   1.88  $
// Файл сгенерирован 8/23/99 12:22:29 AM из библиотеки типов, описанной ниже.

// ************************************************************************ //
// ПРИМЕЧАНИЕ:
// Элементы, относящиеся к директиве $IFDEF_LIVE_SERVER_AT_DESIGN_TIME,
// используются свойствами, возвращающими объекты, которые должны быть явно
// созданы с помощью вызова функции перед осуществлением любой операции
// доступа к этому свойству. Для предотвращение случайного использования
// в инспекторе объектов эти элементы должны быть недоступны.
// Их можно активизировать, определив IFDEF_LIVE_SERVER_AT_DESIGN_TIME
// или выборочно удаляя их из блоков $IFDEF. Однако, перед использованием
// такие элементы должны быть программно созданы с помощью метода
// соответствующего компонентного класса.
// ************************************************************************ //
// Type Lib: X:\work\d5dg\code\Ch25\Memo\Memo.tlb (1)
```

```
// IID\ LCID: {0DB4686F-09C5-11D2-AE5C-00A024E3867F}\0
// HelpFile:
// DepndLst:
//   (1) v2.0 stdole, (C:\WINDOWS\SYSTEM\STDOLE2.TLB)
//   (2) v4.0 StdVCL, (C:\WINDOWS\SYSTEM\STDVCL40.DLL)
// ************************************************************************ //
{$TYPEDADDRESS OFF} // Модуль должен компилироваться без
                    // проверки типов указателей.
interface

uses Windows, ActiveX, Classes, Graphics, OleServer, OleCtrls, StdVCL;

// ************************************************************************ //
// GUID объявлены в библиотеке типов. Используются следующие префиксы:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// ************************************************************************ //
const
  // Версии библиотеки типов
  MemoMajorVersion = 1;
  MemoMinorVersion = 0;
  LIBID_Memo: TGUID = '{0DB4686F-09C5-11D2-AE5C-00A024E3867F}';
  IID_IMemoX: TGUID = '{0DB46870-09C5-11D2-AE5C-00A024E3867F}';
  DIID_IMemoXEvents: TGUID = '{0DB46872-09C5-11D2-AE5C-00A024E3867F}';
  CLASS_MemoX: TGUID = '{0DB46874-09C5-11D2-AE5C-00A024E3867F}';

// ************************************************************************ //
// Объявление перечислений, определенных в библиотеке типов
// ************************************************************************ //
// Константы для перечисления TxAlignment
type
  TxAlignment = TOleEnum;
const
  taLeftJustify = $00000000;
  taRightJustify = $00000001;
  taCenter = $00000002;

// Константы для перечисления TxBiDiMode
type
  TxBiDiMode = TOleEnum;
const
  bdLeftToRight = $00000000;
  bdRightToLeft = $00000001;
  bdRightToLeftNoAlign = $00000002;
  bdRightToLeftReadingOnly = $00000003;

// Константы для перечисления TxBorderStyle
type
  TxBorderStyle = TOleEnum;
```

```
const
  bsNone = $00000000;
  bsSingle = $00000001;

// Константы для перечисления TxDragMode
type
  TxDragMode = TOleEnum;
const
  dmManual = $00000000;
  dmAutomatic = $00000001;

// Константы для перечисления TxImeMode
type
  TxImeMode = TOleEnum;
const
  imDisable = $00000000;
  imClose = $00000001;
  imOpen = $00000002;
  imDontCare = $00000003;
  imSAlpha = $00000004;
  imAlpha = $00000005;
  imHira = $00000006;
  imSKata = $00000007;
  imKata = $00000008;
  imChinese = $00000009;
  imSHanguel = $0000000A;
  imHanguel = $0000000B;

// Константы для перечисления TxScrollStyle
type
  TxScrollStyle = TOleEnum;
const
  ssNone = $00000000;
  ssHorizontal = $00000001;
  ssVertical = $00000002;
  ssBoth = $00000003;

// Константы для перечисления TxMouseButton
type
  TxMouseButton = TOleEnum;
const
  mbLeft = $00000000;
  mbRight = $00000001;
  mbMiddle = $00000002;

type

// ********************************************************************* //
// Предварительное объявление типов, определенных в библиотеке типов.
// ********************************************************************* //
  IMemoX = interface;
```

```
  IMemoXDisp = dispinterface;
  IMemoXEvents = dispinterface;

// *********************************************************************** //
// Объявление компонентов класса, определенных в библиотеке типов.
// (ПРИМЕЧАНИЕ. Здесь описывается каждый компонент класса со своим
// стандартным интерфейсом.)
// *********************************************************************** //
  MemoX = IMemoX;

// *********************************************************************** //
// Interface: IMemoX
// Flags:     (4416) Hidden Dual OleAutomation Dispatchable
// GUID:      {0DB46870-09C5-11D2-AE5C-00A024E3867F}
// *********************************************************************** //
  IMemoX = interface(IDispatch)
    ['{0DB46870-09C5-11D2-AE5C-00A024E3867F}']
    function  Get_Alignment: TxAlignment; safecall;
    procedure Set_Alignment(Value: TxAlignment); safecall;
    function  Get_BiDiMode: TxBiDiMode; safecall;
    procedure Set_BiDiMode(Value: TxBiDiMode); safecall;
    function  Get_BorderStyle: TxBorderStyle; safecall;
    procedure Set_BorderStyle(Value: TxBorderStyle); safecall;
    function  Get_Color: OLE_COLOR; safecall;
    procedure Set_Color(Value: OLE_COLOR); safecall;
    function  Get_Ctl3D: WordBool; safecall;
    procedure Set_Ctl3D(Value: WordBool); safecall;
    function  Get_DragCursor: Smallint; safecall;
    procedure Set_DragCursor(Value: Smallint); safecall;
    function  Get_DragMode: TxDragMode; safecall;
    procedure Set_DragMode(Value: TxDragMode); safecall;
    function  Get_Enabled: WordBool; safecall;
    procedure Set_Enabled(Value: WordBool); safecall;
    function  Get_Font: IFontDisp; safecall;
    procedure _Set_Font(const Value: IFontDisp); safecall;
    procedure Set_Font(var Value: IFontDisp); safecall;
    function  Get_HideSelection: WordBool; safecall;
    procedure Set_HideSelection(Value: WordBool); safecall;
    function  Get_ImeMode: TxImeMode; safecall;
    procedure Set_ImeMode(Value: TxImeMode); safecall;
    function  Get_ImeName: WideString; safecall;
    procedure Set_ImeName(const Value: WideString); safecall;
    function  Get_MaxLength: Integer; safecall;
    procedure Set_MaxLength(Value: Integer); safecall;
    function  Get_OEMConvert: WordBool; safecall;
    procedure Set_OEMConvert(Value: WordBool); safecall;
    function  Get_ParentColor: WordBool; safecall;
    procedure Set_ParentColor(Value: WordBool); safecall;
    function  Get_ParentCtl3D: WordBool; safecall;
    procedure Set_ParentCtl3D(Value: WordBool); safecall;
    function  Get_ParentFont: WordBool; safecall;
```

```
procedure Set_ParentFont(Value: WordBool); safecall;
function  Get_ReadOnly: WordBool; safecall;
procedure Set_ReadOnly(Value: WordBool); safecall;
function  Get_ScrollBars: TxScrollStyle; safecall;
procedure Set_ScrollBars(Value: TxScrollStyle); safecall;
function  Get_Visible: WordBool; safecall;
procedure Set_Visible(Value: WordBool); safecall;
function  Get_WantReturns: WordBool; safecall;
procedure Set_WantReturns(Value: WordBool); safecall;
function  Get_WantTabs: WordBool; safecall;
procedure Set_WantTabs(Value: WordBool); safecall;
function  Get_WordWrap: WordBool; safecall;
procedure Set_WordWrap(Value: WordBool); safecall;
function  GetControlsAlignment: TxAlignment; safecall;
procedure Clear; safecall;
procedure ClearSelection; safecall;
procedure CopyToClipboard; safecall;
procedure CutToClipboard; safecall;
procedure PasteFromClipboard; safecall;
procedure Undo; safecall;
procedure ClearUndo; safecall;
procedure SelectAll; safecall;
function  Get_CanUndo: WordBool; safecall;
function  Get_Modified: WordBool; safecall;
procedure Set_Modified(Value: WordBool); safecall;
function  Get_SelLength: Integer; safecall;
procedure Set_SelLength(Value: Integer); safecall;
function  Get_SelStart: Integer; safecall;
procedure Set_SelStart(Value: Integer); safecall;
function  Get_SelText: WideString; safecall;
procedure Set_SelText(const Value: WideString); safecall;
function  Get_Text: WideString; safecall;
procedure Set_Text(const Value: WideString); safecall;
function  Get_DoubleBuffered: WordBool; safecall;
procedure Set_DoubleBuffered(Value: WordBool); safecall;
procedure FlipChildren(AllLevels: WordBool); safecall;
function  DrawTextBiDiModeFlags(Flags: Integer): Integer; safecall;
function  DrawTextBiDiModeFlagsReadingOnly: Integer; safecall;
procedure InitiateAction; safecall;
function  IsRightToLeft: WordBool; safecall;
function  UseRightToLeftAlignment: WordBool; safecall;
function  UseRightToLeftReading: WordBool; safecall;
function  UseRightToLeftScrollBar: WordBool; safecall;
function  Get_Cursor: Smallint; safecall;
procedure Set_Cursor(Value: Smallint); safecall;
function  ClassNameIs(const Name: WideString): WordBool; safecall;
procedure AboutBox; safecall;
property Alignment: TxAlignment read Get_Alignment write Set_Alignment;
property BiDiMode: TxBiDiMode read Get_BiDiMode write Set_BiDiMode;
property BorderStyle: TxBorderStyle read Get_BorderStyle write
  Set_BorderStyle;
```

```
    property Color: OLE_COLOR read Get_Color write Set_Color;
    property Ctl3D: WordBool read Get_Ctl3D write Set_Ctl3D;
    property DragCursor: Smallint read Get_DragCursor write Set_DragCursor;
    property DragMode: TxDragMode read Get_DragMode write Set_DragMode;
    property Enabled: WordBool read Get_Enabled write Set_Enabled;
    property Font: IFontDisp read Get_Font write _Set_Font;
    property HideSelection: WordBool read Get_HideSelection write
      Set_HideSelection;
    property ImeMode: TxImeMode read Get_ImeMode write Set_ImeMode;
    property ImeName: WideString read Get_ImeName write Set_ImeName;
    property MaxLength: Integer read Get_MaxLength write Set_MaxLength;
    property OEMConvert: WordBool read Get_OEMConvert write Set_OEMConvert;
    property ParentColor: WordBool read Get_ParentColor write Set_ParentColor;
    property ParentCtl3D: WordBool read Get_ParentCtl3D write Set_ParentCtl3D;
    property ParentFont: WordBool read Get_ParentFont write Set_ParentFont;
    property ReadOnly: WordBool read Get_ReadOnly write Set_ReadOnly;
    property ScrollBars: TxScrollStyle read Get_ScrollBars write
      Set_ScrollBars;
    property Visible: WordBool read Get_Visible write Set_Visible;
    property WantReturns: WordBool read Get_WantReturns write Set_WantReturns;
    property WantTabs: WordBool read Get_WantTabs write Set_WantTabs;
    property WordWrap: WordBool read Get_WordWrap write Set_WordWrap;
    property CanUndo: WordBool read Get_CanUndo;
    property Modified: WordBool read Get_Modified write Set_Modified;
    property SelLength: Integer read Get_SelLength write Set_SelLength;
    property SelStart: Integer read Get_SelStart write Set_SelStart;
    property SelText: WideString read Get_SelText write Set_SelText;
    property Text: WideString read Get_Text write Set_Text;
    property DoubleBuffered: WordBool read Get_DoubleBuffered write
      Set_DoubleBuffered;
    property Cursor: Smallint read Get_Cursor write Set_Cursor;
  end;

// *********************************************************************** //
// DispIntf:  IMemoXDisp
// Flags:     (4416) Hidden Dual OleAutomation Dispatchable
// GUID:      {0DB46870-09C5-11D2-AE5C-00A024E3867F}
// *********************************************************************** //
  IMemoXDisp = dispinterface
    ['{0DB46870-09C5-11D2-AE5C-00A024E3867F}']
    property Alignment: TxAlignment dispid 1;
    property BiDiMode: TxBiDiMode dispid 2;
    property BorderStyle: TxBorderStyle dispid 3;
    property Color: OLE_COLOR dispid -501;
    property Ctl3D: WordBool dispid 4;
    property DragCursor: Smallint dispid 5;
    property DragMode: TxDragMode dispid 6;
    property Enabled: WordBool dispid -514;
    property Font: IFontDisp dispid -512;
    property HideSelection: WordBool dispid 7;
    property ImeMode: TxImeMode dispid 8;
```

```
    property ImeName: WideString dispid 9;
    property MaxLength: Integer dispid 10;
    property OEMConvert: WordBool dispid 11;
    property ParentColor: WordBool dispid 12;
    property ParentCtl3D: WordBool dispid 13;
    property ParentFont: WordBool dispid 14;
    property ReadOnly: WordBool dispid 15;
    property ScrollBars: TxScrollStyle dispid 16;
    property Visible: WordBool dispid 17;
    property WantReturns: WordBool dispid 18;
    property WantTabs: WordBool dispid 19;
    property WordWrap: WordBool dispid 20;
    function GetControlsAlignment: TxAlignment; dispid 21;
    procedure Clear; dispid 22;
    procedure ClearSelection; dispid 23;
    procedure CopyToClipboard; dispid 24;
    procedure CutToClipboard; dispid 25;
    procedure PasteFromClipboard; dispid 27;
    procedure Undo; dispid 28;
    procedure ClearUndo; dispid 29;
    procedure SelectAll; dispid 31;
    property CanUndo: WordBool readonly dispid 33;
    property Modified: WordBool dispid 34;
    property SelLength: Integer dispid 35;
    property SelStart: Integer dispid 36;
    property SelText: WideString dispid 37;
    property Text: WideString dispid -517;
    property DoubleBuffered: WordBool dispid 39;
    procedure FlipChildren(AllLevels: WordBool); dispid 40;
    function  DrawTextBiDiModeFlags(Flags: Integer): Integer; dispid 43;
    function  DrawTextBiDiModeFlagsReadingOnly: Integer; dispid 44;
    procedure InitiateAction; dispid 46;
    function  IsRightToLeft: WordBool; dispid 47;
    function  UseRightToLeftAlignment: WordBool; dispid 52;
    function  UseRightToLeftReading: WordBool; dispid 53;
    function  UseRightToLeftScrollBar: WordBool; dispid 54;
    property Cursor: Smallint dispid 55;
    function ClassNameIs(const Name: WideString): WordBool; dispid 59;
    procedure AboutBox; dispid -552;
  end;

// ******************************************************************** //
// DispIntf:  IMemoXEvents
// Flags:     (4096) Dispatchable
// GUID:      {0DB46872-09C5-11D2-AE5C-00A024E3867F}
// ******************************************************************** //
  IMemoXEvents = dispinterface
    ['{0DB46872-09C5-11D2-AE5C-00A024E3867F}']
    procedure OnChange; dispid 1;
    procedure OnClick; dispid 2;
    procedure OnDblClick; dispid 3;
```

```
    procedure OnKeyPress(var Key: Smallint); dispid 9;
  end;

// *********************************************************************** //
// OLE Control Proxy class declaration
// Control Name     : TMemoX
// Help String      : MemoX Control
// Default Interface: IMemoX
// Def. Intf. DISP? : No
// Event   Interface: IMemoXEvents
// TypeFlags        : (34) CanCreate Control
// *********************************************************************** //
  TMemoXOnKeyPress = procedure(Sender: TObject; var Key: Smallint) of object;

  TMemoX = class(TOleControl)
  private
    FOnChange: TNotifyEvent;
    FOnClick: TNotifyEvent;
    FOnDblClick: TNotifyEvent;
    FOnKeyPress: TMemoXOnKeyPress;
    FIntf: IMemoX;
    function  GetControlInterface: IMemoX;
  protected
    procedure CreateControl;
    procedure InitControlData; override;
  public
    function  GetControlsAlignment: TxAlignment;
    procedure Clear;
    procedure ClearSelection;
    procedure CopyToClipboard;
    procedure CutToClipboard;
    procedure PasteFromClipboard;
    procedure Undo;
    procedure ClearUndo;
    procedure SelectAll;
    procedure FlipChildren(AllLevels: WordBool);
    function  DrawTextBiDiModeFlags(Flags: Integer): Integer;
    function  DrawTextBiDiModeFlagsReadingOnly: Integer;
    procedure InitiateAction;
    function  IsRightToLeft: WordBool;
    function  UseRightToLeftAlignment: WordBool;
    function  UseRightToLeftReading: WordBool;
    function  UseRightToLeftScrollBar: WordBool;
    function  ClassNameIs(const Name: WideString): WordBool;
    procedure AboutBox;
    property  ControlInterface: IMemoX read GetControlInterface;
    property  DefaultInterface: IMemoX read GetControlInterface;
    property CanUndo: WordBool index 33 read GetWordBoolProp;
    property Modified: WordBool index 34 read GetWordBoolProp write
      SetWordBoolProp;
    property SelLength: Integer index 35 read GetIntegerProp write
```

```
    SetIntegerProp;
  property SelStart: Integer index 36 read GetIntegerProp write
    SetIntegerProp;
  property SelText: WideString index 37 read GetWideStringProp write
    SetWideStringProp;
  property Text: WideString index -517 read GetWideStringProp write
    SetWideStringProp;
  property DoubleBuffered: WordBool index 39 read GetWordBoolProp write
    SetWordBoolProp;
published
  property Alignment: TOleEnum index 1 read GetTOleEnumProp write
    SetTOleEnumProp stored False;
  property BiDiMode: TOleEnum index 2 read GetTOleEnumProp write
    SetTOleEnumProp stored False;
  property BorderStyle: TOleEnum index 3 read GetTOleEnumProp write
    SetTOleEnumProp stored False;
  property Color: TColor index -501 read GetTColorProp write SetTColorProp
    stored False;
  property Ctl3D: WordBool index 4 read GetWordBoolProp write
    SetWordBoolProp stored False;
  property DragCursor: Smallint index 5 read GetSmallintProp write
    SetSmallintProp stored False;
  property DragMode: TOleEnum index 6 read GetTOleEnumProp write
    SetTOleEnumProp stored False;
  property Enabled: WordBool index -514 read GetWordBoolProp write
    SetWordBoolProp stored False;
  property Font: TFont index -512 read GetTFontProp write SetTFontProp
    stored False;
  property HideSelection: WordBool index 7 read GetWordBoolProp write
    SetWordBoolProp stored False;
  property ImeMode: TOleEnum index 8 read GetTOleEnumProp write
    SetTOleEnumProp stored False;
  property ImeName: WideString index 9 read GetWideStringProp write
    SetWideStringProp stored False;
  property MaxLength: Integer index 10 read GetIntegerProp write
    SetIntegerProp stored False;
  property OEMConvert: WordBool index 11 read GetWordBoolProp write
    SetWordBoolProp stored False;
  property ParentColor: WordBool index 12 read GetWordBoolProp write
    SetWordBoolProp stored False;
  property ParentCtl3D: WordBool index 13 read GetWordBoolProp write
    SetWordBoolProp stored False;
  property ParentFont: WordBool index 14 read GetWordBoolProp write
    SetWordBoolProp stored False;
  property ReadOnly: WordBool index 15 read GetWordBoolProp write
    SetWordBoolProp stored False;
  property ScrollBars: TOleEnum index 16 read GetTOleEnumProp write
    SetTOleEnumProp stored False;
  property Visible: WordBool index 17 read GetWordBoolProp write
    SetWordBoolProp stored False;
  property WantReturns: WordBool index 18 read GetWordBoolProp write
```

```
      SetWordBoolProp stored False;
    property WantTabs: WordBool index 19 read GetWordBoolProp write
      SetWordBoolProp stored False;
    property WordWrap: WordBool index 20 read GetWordBoolProp write
      SetWordBoolProp stored False;
    property Cursor: Smallint index 55 read GetSmallintProp write
      SetSmallintProp stored False;
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
    property OnDblClick: TNotifyEvent read FOnDblClick write FOnDblClick;
    property OnKeyPress: TMemoXOnKeyPress read FOnKeyPress write FOnKeyPress;
  end;

procedure Register;

implementation

uses ComObj;

procedure TMemoX.InitControlData;
const
  CEventDispIDs: array [0..3] of DWORD = (
    $00000001, $00000002, $00000003, $00000009);
  CTFontIDs: array [0..0] of DWORD = ($FFFFFE00);
  CControlData: TControlData2 = (
    ClassID: '{0DB46874-09C5-11D2-AE5C-00A024E3867F}';
    EventIID: '{0DB46872-09C5-11D2-AE5C-00A024E3867F}';
    EventCount: 4;
    EventDispIDs: @CEventDispIDs;
    LicenseKey: nil (*HR:$80040154*);
    Flags: $0000002D;
    Version: 401;
    FontCount: 1;
    FontIDs: @CTFontIDs);
begin
  ControlData := @CControlData;
  TControlData2(CControlData).FirstEventOfs := Cardinal(@@FOnChange) -
     Cardinal(Self);
end;

procedure TMemoX.CreateControl;

  procedure DoCreate;
  begin
    FIntf := IUnknown(OleObject) as IMemoX;
  end;

begin
  if FIntf = nil then DoCreate;
end;
```

```
function TMemoX.GetControlInterface: IMemoX;
begin
  CreateControl;
  Result := FIntf;
end;

function TMemoX.GetControlsAlignment: TxAlignment;
begin
  Result := DefaultInterface.GetControlsAlignment;
end;

procedure TMemoX.Clear;
begin
  DefaultInterface.Clear;
end;

procedure TMemoX.ClearSelection;
begin
  DefaultInterface.ClearSelection;
end;

procedure TMemoX.CopyToClipboard;
begin
  DefaultInterface.CopyToClipboard;
end;

procedure TMemoX.CutToClipboard;
begin
  DefaultInterface.CutToClipboard;
end;

procedure TMemoX.PasteFromClipboard;
begin
  DefaultInterface.PasteFromClipboard;
end;

procedure TMemoX.Undo;
begin
  DefaultInterface.Undo;
end;

procedure TMemoX.ClearUndo;
begin
  DefaultInterface.ClearUndo;
end;

procedure TMemoX.SelectAll;
begin
  DefaultInterface.SelectAll;
end;
```

```
procedure TMemoX.FlipChildren(AllLevels: WordBool);
begin
  DefaultInterface.FlipChildren(AllLevels);
end;

function TMemoX.DrawTextBiDiModeFlags(Flags: Integer): Integer;
begin
  Result := DefaultInterface.DrawTextBiDiModeFlags(Flags);
end;

function TMemoX.DrawTextBiDiModeFlagsReadingOnly: Integer;
begin
  Result := DefaultInterface.DrawTextBiDiModeFlagsReadingOnly;
end;

procedure TMemoX.InitiateAction;
begin
  DefaultInterface.InitiateAction;
end;

function TMemoX.IsRightToLeft: WordBool;
begin
  Result := DefaultInterface.IsRightToLeft;
end;

function TMemoX.UseRightToLeftAlignment: WordBool;
begin
  Result := DefaultInterface.UseRightToLeftAlignment;
end;

function TMemoX.UseRightToLeftReading: WordBool;
begin
  Result := DefaultInterface.UseRightToLeftReading;
end;

function TMemoX.UseRightToLeftScrollBar: WordBool;
begin
  Result := DefaultInterface.UseRightToLeftScrollBar;
end;

function TMemoX.ClassNameIs(const Name: WideString): WordBool;
begin
  Result := DefaultInterface.ClassNameIs(Name);
end;

procedure TMemoX.AboutBox;
begin
  DefaultInterface.AboutBox;
end;

procedure Register;
```

```
begin
  RegisterComponents('ActiveX',[TMemoX]);
end;

end.
```

На заметку

Если внимательно просмотреть код листинга 25.2, то можно заметить, что кроме информации библиотеки типов Memo_TLB.pas, в нем содержится класс TMemoX, который представляет собой оболочку класса TOleControl для элемента управления ActiveX. Это позволяет добавлять созданный в Delphi элемент управления ActiveX к палитре, просто дополнив пакет времени разработки сгенерированным модулем *xxx*_TLB.

Листинг 25.3. Файл реализации MemoImpl.pas

```
unit MemoImpl;

interface

uses
  Windows, ActiveX, Classes, Controls, Graphics, Menus, Forms,
  StdCtrls, ComServ, StdVCL, AXCtrls, Memo_TLB;

type
  TMemoX = class(TActiveXControl, IMemoX)
  private
    { Закрытые объявления }
    FDelphiControl: TMemo;
    FEvents: IMemoXEvents;
    procedure ChangeEvent(Sender: TObject);
    procedure ClickEvent(Sender: TObject);
    procedure DblClickEvent(Sender: TObject);
    procedure KeyPressEvent(Sender: TObject; var Key: Char);
  protected
    { Защищенные объявления }
    procedure DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage);
      override;
    procedure EventSinkChanged(const EventSink: IUnknown); override;
    procedure InitializeControl; override;
    function ClassNameIs(const Name: WideString): WordBool; safecall;
    function DrawTextBiDiModeFlags(Flags: Integer): Integer; safecall;
    function DrawTextBiDiModeFlagsReadingOnly: Integer; safecall;
    function Get_Alignment: TxAlignment; safecall;
    function Get_BiDiMode: TxBiDiMode; safecall;
    function Get_BorderStyle: TxBorderStyle; safecall;
    function Get_CanUndo: WordBool; safecall;
    function Get_Color: OLE_COLOR; safecall;
    function Get_Ctl3D: WordBool; safecall;
    function Get_Cursor: Smallint; safecall;
    function Get_DoubleBuffered: WordBool; safecall;
    function Get_DragCursor: Smallint; safecall;
    function Get_DragMode: TxDragMode; safecall;
```

```
function Get_Enabled: WordBool; safecall;
function Get_Font: IFontDisp; safecall;
function Get_HideSelection: WordBool; safecall;
function Get_ImeMode: TxImeMode; safecall;
function Get_ImeName: WideString; safecall;
function Get_MaxLength: Integer; safecall;
function Get_Modified: WordBool; safecall;
function Get_OEMConvert: WordBool; safecall;
function Get_ParentColor: WordBool; safecall;
function Get_ParentCtl3D: WordBool; safecall;
function Get_ParentFont: WordBool; safecall;
function Get_ReadOnly: WordBool; safecall;
function Get_ScrollBars: TxScrollStyle; safecall;
function Get_SelLength: Integer; safecall;
function Get_SelStart: Integer; safecall;
function Get_SelText: WideString; safecall;
function Get_Text: WideString; safecall;
function Get_Visible: WordBool; safecall;
function Get_WantReturns: WordBool; safecall;
function Get_WantTabs: WordBool; safecall;
function Get_WordWrap: WordBool; safecall;
function GetControlsAlignment: TxAlignment; safecall;
function IsRightToLeft: WordBool; safecall;
function UseRightToLeftAlignment: WordBool; safecall;
function UseRightToLeftReading: WordBool; safecall;
function UseRightToLeftScrollBar: WordBool; safecall;
procedure _Set_Font(const Value: IFontDisp); safecall;
procedure AboutBox; safecall;
procedure Clear; safecall;
procedure ClearSelection; safecall;
procedure ClearUndo; safecall;
procedure CopyToClipboard; safecall;
procedure CutToClipboard; safecall;
procedure FlipChildren(AllLevels: WordBool); safecall;
procedure InitiateAction; safecall;
procedure PasteFromClipboard; safecall;
procedure SelectAll; safecall;
procedure Set_Alignment(Value: TxAlignment); safecall;
procedure Set_BiDiMode(Value: TxBiDiMode); safecall;
procedure Set_BorderStyle(Value: TxBorderStyle); safecall;
procedure Set_Color(Value: OLE_COLOR); safecall;
procedure Set_Ctl3D(Value: WordBool); safecall;
procedure Set_Cursor(Value: Smallint); safecall;
procedure Set_DoubleBuffered(Value: WordBool); safecall;
procedure Set_DragCursor(Value: Smallint); safecall;
procedure Set_DragMode(Value: TxDragMode); safecall;
procedure Set_Enabled(Value: WordBool); safecall;
procedure Set_Font(var Value: IFontDisp); safecall;
procedure Set_HideSelection(Value: WordBool); safecall;
procedure Set_ImeMode(Value: TxImeMode); safecall;
```

```
    procedure Set_ImeName(const Value: WideString); safecall;
    procedure Set_MaxLength(Value: Integer); safecall;
    procedure Set_Modified(Value: WordBool); safecall;
    procedure Set_OEMConvert(Value: WordBool); safecall;
    procedure Set_ParentColor(Value: WordBool); safecall;
    procedure Set_ParentCtl3D(Value: WordBool); safecall;
    procedure Set_ParentFont(Value: WordBool); safecall;
    procedure Set_ReadOnly(Value: WordBool); safecall;
    procedure Set_ScrollBars(Value: TxScrollStyle); safecall;
    procedure Set_SelLength(Value: Integer); safecall;
    procedure Set_SelStart(Value: Integer); safecall;
    procedure Set_SelText(const Value: WideString); safecall;
    procedure Set_Text(const Value: WideString); safecall;
    procedure Set_Visible(Value: WordBool); safecall;
    procedure Set_WantReturns(Value: WordBool); safecall;
    procedure Set_WantTabs(Value: WordBool); safecall;
    procedure Set_WordWrap(Value: WordBool); safecall;
    procedure Undo; safecall;
  end;

implementation

uses ComObj, About;

{ TMemoX }

procedure TMemoX.DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage);
begin
  { Здесь должно находиться определение страницы свойств. Страницы свойств
    определяются с помощью вызова DefinePropertyPage с идентификатором
    класса страницы. Например, DefinePropertyPage(Class_MemoXPage); }
end;

procedure TMemoX.EventSinkChanged(const EventSink: IUnknown);
begin
  FEvents := EventSink as IMemoXEvents;
end;

procedure TMemoX.InitializeControl;
begin
  FDelphiControl := Control as TMemo;
  FDelphiControl.OnChange := ChangeEvent;
  FDelphiControl.OnClick := ClickEvent;
  FDelphiControl.OnDblClick := DblClickEvent;
  FDelphiControl.OnKeyPress := KeyPressEvent;
end;

function TMemoX.ClassNameIs(const Name: WideString): WordBool;
begin
  Result := FDelphiControl.ClassNameIs(Name);
```

```
end;

function TMemoX.DrawTextBiDiModeFlags(Flags: Integer): Integer;
begin
  Result := FDelphiControl.DrawTextBiDiModeFlags(Flags);
end;

function TMemoX.DrawTextBiDiModeFlagsReadingOnly: Integer;
begin
  Result := FDelphiControl.DrawTextBiDiModeFlagsReadingOnly;
end;

function TMemoX.Get_Alignment: TxAlignment;
begin
  Result := Ord(FDelphiControl.Alignment);
end;

function TMemoX.Get_BiDiMode: TxBiDiMode;
begin
  Result := Ord(FDelphiControl.BiDiMode);
end;

function TMemoX.Get_BorderStyle: TxBorderStyle;
begin
  Result := Ord(FDelphiControl.BorderStyle);
end;

function TMemoX.Get_CanUndo: WordBool;
begin
  Result := FDelphiControl.CanUndo;
end;

function TMemoX.Get_Color: OLE_COLOR;
begin
  Result := OLE_COLOR(FDelphiControl.Color);
end;

function TMemoX.Get_Ctl3D: WordBool;
begin
  Result := FDelphiControl.Ctl3D;
end;

function TMemoX.Get_Cursor: Smallint;
begin
  Result := Smallint(FDelphiControl.Cursor);
end;

function TMemoX.Get_DoubleBuffered: WordBool;
begin
  Result := FDelphiControl.DoubleBuffered;
```

```
end;

function TMemoX.Get_DragCursor: Smallint;
begin
  Result := Smallint(FDelphiControl.DragCursor);
end;

function TMemoX.Get_DragMode: TxDragMode;
begin
  Result := Ord(FDelphiControl.DragMode);
end;

function TMemoX.Get_Enabled: WordBool;
begin
  Result := FDelphiControl.Enabled;
end;

function TMemoX.Get_Font: IFontDisp;
begin
  GetOleFont(FDelphiControl.Font, Result);
end;

function TMemoX.Get_HideSelection: WordBool;
begin
  Result := FDelphiControl.HideSelection;
end;

function TMemoX.Get_ImeMode: TxImeMode;
begin
  Result := Ord(FDelphiControl.ImeMode);
end;

function TMemoX.Get_ImeName: WideString;
begin
  Result := WideString(FDelphiControl.ImeName);
end;

function TMemoX.Get_MaxLength: Integer;
begin
  Result := FDelphiControl.MaxLength;
end;

function TMemoX.Get_Modified: WordBool;
begin
  Result := FDelphiControl.Modified;
end;

function TMemoX.Get_OEMConvert: WordBool;
begin
  Result := FDelphiControl.OEMConvert;
```

```
end;

function TMemoX.Get_ParentColor: WordBool;
begin
  Result := FDelphiControl.ParentColor;
end;

function TMemoX.Get_ParentCtl3D: WordBool;
begin
  Result := FDelphiControl.ParentCtl3D;
end;

function TMemoX.Get_ParentFont: WordBool;
begin
  Result := FDelphiControl.ParentFont;
end;

function TMemoX.Get_ReadOnly: WordBool;
begin
  Result := FDelphiControl.ReadOnly;
end;

function TMemoX.Get_ScrollBars: TxScrollStyle;
begin
  Result := Ord(FDelphiControl.ScrollBars);
end;

function TMemoX.Get_SelLength: Integer;
begin
  Result := FDelphiControl.SelLength;
end;

function TMemoX.Get_SelStart: Integer;
begin
  Result := FDelphiControl.SelStart;
end;

function TMemoX.Get_SelText: WideString;
begin
  Result := WideString(FDelphiControl.SelText);
end;

function TMemoX.Get_Text: WideString;
begin
  Result := WideString(FDelphiControl.Text);
end;

function TMemoX.Get_Visible: WordBool;
begin
  Result := FDelphiControl.Visible;
```

```
end;

function TMemoX.Get_WantReturns: WordBool;
begin
  Result := FDelphiControl.WantReturns;
end;

function TMemoX.Get_WantTabs: WordBool;
begin
  Result := FDelphiControl.WantTabs;
end;

function TMemoX.Get_WordWrap: WordBool;
begin
  Result := FDelphiControl.WordWrap;
end;

function TMemoX.GetControlsAlignment: TxAlignment;
begin
  Result := TxAlignment(FDelphiControl.GetControlsAlignment);
end;

function TMemoX.IsRightToLeft: WordBool;
begin
  Result := FDelphiControl.IsRightToLeft;
end;

function TMemoX.UseRightToLeftAlignment: WordBool;
begin
  Result := FDelphiControl.UseRightToLeftAlignment;
end;

function TMemoX.UseRightToLeftReading: WordBool;
begin
  Result := FDelphiControl.UseRightToLeftReading;
end;

function TMemoX.UseRightToLeftScrollBar: WordBool;
begin
  Result := FDelphiControl.UseRightToLeftScrollBar;
end;

procedure TMemoX._Set_Font(const Value: IFontDisp);
begin
  SetOleFont(FDelphiControl.Font, Value);
end;

procedure TMemoX.AboutBox;
begin
  ShowMemoXAbout;
```

```
end;

procedure TMemoX.Clear;
begin
  FDelphiControl.Clear;
end;

procedure TMemoX.ClearSelection;
begin
  FDelphiControl.ClearSelection;
end;

procedure TMemoX.ClearUndo;
begin
  FDelphiControl.ClearUndo;
end;

procedure TMemoX.CopyToClipboard;
begin
  FDelphiControl.CopyToClipboard;
end;

procedure TMemoX.CutToClipboard;
begin
  FDelphiControl.CutToClipboard;
end;

procedure TMemoX.FlipChildren(AllLevels: WordBool);
begin
  FDelphiControl.FlipChildren(AllLevels);
end;

procedure TMemoX.InitiateAction;
begin
  FDelphiControl.InitiateAction;
end;

procedure TMemoX.PasteFromClipboard;
begin
  FDelphiControl.PasteFromClipboard;
end;

procedure TMemoX.SelectAll;
begin
  FDelphiControl.SelectAll;
end;

procedure TMemoX.Set_Alignment(Value: TxAlignment);
begin
  FDelphiControl.Alignment := TAlignment(Value);
```

```
end;

procedure TMemoX.Set_BiDiMode(Value: TxBiDiMode);
begin
  FDelphiControl.BiDiMode := TBiDiMode(Value);
end;

procedure TMemoX.Set_BorderStyle(Value: TxBorderStyle);
begin
  FDelphiControl.BorderStyle := TBorderStyle(Value);
end;

procedure TMemoX.Set_Color(Value: OLE_COLOR);
begin
  FDelphiControl.Color := TColor(Value);
end;

procedure TMemoX.Set_Ctl3D(Value: WordBool);
begin
  FDelphiControl.Ctl3D := Value;
end;

procedure TMemoX.Set_Cursor(Value: Smallint);
begin
  FDelphiControl.Cursor := TCursor(Value);
end;

procedure TMemoX.Set_DoubleBuffered(Value: WordBool);
begin
  FDelphiControl.DoubleBuffered := Value;
end;

procedure TMemoX.Set_DragCursor(Value: Smallint);
begin
  FDelphiControl.DragCursor := TCursor(Value);
end;

procedure TMemoX.Set_DragMode(Value: TxDragMode);
begin
  FDelphiControl.DragMode := TDragMode(Value);
end;

procedure TMemoX.Set_Enabled(Value: WordBool);
begin
  FDelphiControl.Enabled := Value;
end;

procedure TMemoX.Set_Font(var Value: IFontDisp);
begin
  SetOleFont(FDelphiControl.Font, Value);
```

```
end;

procedure TMemoX.Set_HideSelection(Value: WordBool);
begin
  FDelphiControl.HideSelection := Value;
end;

procedure TMemoX.Set_ImeMode(Value: TxImeMode);
begin
  FDelphiControl.ImeMode := TImeMode(Value);
end;

procedure TMemoX.Set_ImeName(const Value: WideString);
begin
  FDelphiControl.ImeName := TImeName(Value);
end;

procedure TMemoX.Set_MaxLength(Value: Integer);
begin
  FDelphiControl.MaxLength := Value;
end;

procedure TMemoX.Set_Modified(Value: WordBool);
begin
  FDelphiControl.Modified := Value;
end;

procedure TMemoX.Set_OEMConvert(Value: WordBool);
begin
  FDelphiControl.OEMConvert := Value;
end;

procedure TMemoX.Set_ParentColor(Value: WordBool);
begin
  FDelphiControl.ParentColor := Value;
end;

procedure TMemoX.Set_ParentCtl3D(Value: WordBool);
begin
  FDelphiControl.ParentCtl3D := Value;
end;

procedure TMemoX.Set_ParentFont(Value: WordBool);
begin
  FDelphiControl.ParentFont := Value;
end;

procedure TMemoX.Set_ReadOnly(Value: WordBool);
begin
  FDelphiControl.ReadOnly := Value;
```

```
end;

procedure TMemoX.Set_ScrollBars(Value: TxScrollStyle);
begin
  FDelphiControl.ScrollBars := TScrollStyle(Value);
end;

procedure TMemoX.Set_SelLength(Value: Integer);
begin
  FDelphiControl.SelLength := Value;
end;

procedure TMemoX.Set_SelStart(Value: Integer);
begin
  FDelphiControl.SelStart := Value;
end;

procedure TMemoX.Set_SelText(const Value: WideString);
begin
  FDelphiControl.SelText := String(Value);
end;

procedure TMemoX.Set_Text(const Value: WideString);
begin
  FDelphiControl.Text := TCaption(Value);
end;

procedure TMemoX.Set_Visible(Value: WordBool);
begin
  FDelphiControl.Visible := Value;
end;

procedure TMemoX.Set_WantReturns(Value: WordBool);
begin
  FDelphiControl.WantReturns := Value;
end;

procedure TMemoX.Set_WantTabs(Value: WordBool);
begin
  FDelphiControl.WantTabs := Value;
end;

procedure TMemoX.Set_WordWrap(Value: WordBool);
begin
  FDelphiControl.WordWrap := Value;
end;

procedure TMemoX.Undo;
begin
  FDelphiControl.Undo;
```

```
end;

procedure TMemoX.ChangeEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnChange;
end;

procedure TMemoX.ClickEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnClick;
end;

procedure TMemoX.DblClickEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnDblClick;
end;

procedure TMemoX.KeyPressEvent(Sender: TObject; var Key: Char);
var
  TempKey: Smallint;
begin
  TempKey := Smallint(Key);
  if FEvents <> nil then FEvents.OnKeyPress(TempKey);
  Key := Char(TempKey);
end;

initialization
  TActiveXControlFactory.Create(ComServer, TMemoX, TMemo, Class_MemoX,
    1, '', 0, tmApartment);
end.
```

Вероятно, вы обратили внимание на необычайно большие размеры приведенных листингов. Хотя они и выглядят несколько устрашающе, однако в этом обилии кода нет ничего особенно сложного. Наоборот, здесь можно обнаружить нечто приятное: вы уже обладаете полнофункциональным элементом управления ActiveX на основе текстового редактора (включая интерфейс, библиотеку типов и события), и вам не нужно вводить ни одной строки кода!

Обратите внимание на вспомогательные функции, которые используются для двустороннего преобразования свойств IStrings и IFont в собственные типы Delphi TStrings и TFont. Все эти процедуры функционируют одинаково. Они являются мостом между классами Object Pascal и диспетчерскими интерфейсами автоматизации. В табл. 25.1 приведен список классов библиотеки VCL и эквивалентных им интерфейсов автоматизации.

Таблица 25.1. Классы библиотеки VCL и соответствующие интерфейсы автоматизации

Класс библиотеки VCL	Интерфейс автоматизации
TFont	IFont
TPicture	IPicture
TStrings	IStrings

На заметку

Технология ActiveX определяет интерфейсы `IFont` и `IPicture`, тогда как тип `IStrings` определен в библиотеке VCL. В Delphi имеется файл `StdVcl40.dll`, содержащий библиотеку типов, определяющую этот интерфейс. Эта библиотека должна быть установлена и зарегистрирована на компьютере клиента для того, чтобы приложения, использующие элемент управления ActiveX со свойством `IStrings`, функционировали правильно.

Среда разработки ActiveX

Описание среды разработки ActiveX в Delphi (*DAX*) содержится в модуле `AxCtrls`. Любой элемент управления ActiveX всегда описывается как объект автоматизации, поскольку в нем должен быть реализован интерфейс `IDispatch` (помимо многих других). По этой причине среда разработки DAX во многом подобна среде разработки объектов автоматизации, рассмотренной в главе 23, "COM-ориентированные технологии".

Компонент `TActiveXControl` является производным от класса `TAutoObject`, который реализует интерфейсы, необходимые элементу управления ActiveX. Среда разработки DAX функционирует как среда разработки двойного объекта, в котором вся его ActiveX-часть содержится в компоненте `TActiveXControl`, взаимодействующем с отдельным классом `TWinControl`, содержащим инкапсулированный элемент управления библиотеки VCL.

Как и все COM-объекты, элементы управления ActiveX создаются с помощью *фабрик класса*. Класс `TActiveXControlFactory` является фабрикой класса для объекта `TActiveXControl`. Экземпляры фабрик класса создаются в разделе `initialization` файла реализации любого элемента управления ActiveX. Конструктор этого класса определяется следующим образом:

```
constructor TActiveXControlFactory.Create(ComServer: TComServerObject;
  ActiveXControlClass: TActiveXControlClass;
  WinControlClass: TWinControlClass; const ClassID: TGUID;
  ToolboxBitmapID: Integer; const LicStr: string; MiscStatus: Integer;
  ThreadingModel: TThreadingModel = tmSingle);
```

Параметр `ComServer` содержит экземпляр класса `TComServer`. В общем случае в этом параметре передается указатель на объект `ComServer`, глобально объявляемый в модуле `ComServ`.

Параметр `ActiveXControlClass` содержит имя класса, производного от класса `TActiveXControl`, который объявлен в файле реализации.

Параметр `WinControlClass` содержит имя компонента VCL, производного от класса `TWinControl`, который требуется инкапсулировать в элемент управления ActiveX.

Параметр `ClassID` содержит идентификатор CLSID компонентного класса элемента управления, в том виде, в котором он представлен в редакторе библиотеки типов.

Параметр `ToolboxBitmapID` содержит идентификатор ресурса, определяющий растровое изображение, которое должно использоваться для представления элемента управления в палитре компонентов.

Параметр `LicStr` содержит строку, которая должна использоваться как строка ключа лицензии на элемент управления. Если эта строка пуста, значит, элемент управления является нелицензированным.

Параметр `MiscStatus` содержит флаги состояния `OLEMISC_`*`xxx`* элемента управления. Эти флаги определяются в модуле `ActiveX`. Они заносятся в системный реестр при регистрации элемента управления ActiveX. Флаги `OLEMISC` обеспечивают контейнеры элемента управления ActiveX информацией о различных атрибутах данного элемента управления ActiveX. Напри-

мер, они указывают, как элемент управления должен отображаться, может ли он содержать другие элементы управления и т.д. Эти флаги подробно описаны в разделе “OLEMISC” интерактивной системы Microsoft Developer’s Network.

И, наконец, параметр ThreadingModel определяет модель потоков, которую этот элемент управления зарегистрирует как поддерживаемую. Отметим еще одну важную особенность: установка какой-либо модели не гарантирует, что данный элемент управления окажется в безопасности при работе по этой модели. Этот параметр влияет лишь на регистрацию элемента управления. Разработчик должен самостоятельно обеспечивать безопасность элемента управления при его работе в выбранной модели потоков. Каждая из моделей потоков подробно рассматривалась в главе 23, “COM-ориентированные технологии”.

Контейнерные элементы управления

Один из флагов OLEMISC_*xxx* — OLEMISC_SIMPLEFRAME — автоматически устанавливается при добавлении стиля csAcceptsControls к множеству ControlStyle элемента управления VCL. При этом данный элемент управления ActiveX превращается в простую рамку, допускающую помещение в это приложение-контейнер других элементов управления ActiveX. Класс TActiveXControl содержит всю инфраструктуру обработки сообщений, необходимую для поддержания корректной работы простой рамки окна. Иногда мастер добавляет этот флаг к элементу управления, который не предназначен для использования в качестве простой рамки. В этом случае данный флаг следует удалить из вызова конструктора фабрики класса.

Окно отражения

Некоторым элементам управления VCL для правильного функционирования необходимы извещающие сообщения. В этом случае среда разработки DAX создает окно отображения, задача которого состоит в получении сообщений и их пересылке в элемент управления VCL. Стандартные элементы управления VCL, которым необходимо окно отображения, в свое множество ControlStyle всегда включают стиль csReflector. При создании пользовательских компонентов, производных от класса TWinControl, которые используют извещающие сообщения, не забудьте в их конструкторе включить стиль csReflector в множество ControlStyle.

Время разработки и время выполнения

Для определения текущего состояния элемента управления — режим разработки или режим выполнения — библиотека VCL предлагает простое средство. Для этого достаточно проверить состояние элемента csDesigning в множестве ComponentState. Однако определить состояние элементов управления ActiveX уже не так просто. Для этого сначала необходимо получить указатель на диспинтерфейс контейнера IAmbientDispatch, а затем проверить свойство UserMode этого интерфейса. Это можно осуществить следующим образом:

```
function IsControlRunning(Control: IUnknown): Boolean;
var
  OleObj: IOleObject;
  Site: IOleClientSite;
begin
  Result := True;
  { Получение указателя элемента управления IOleObject. После
    этого - получение контейнера IOleClientSite, а затем -
    получение IAmbientDispatch. }
```

```
  if (Control.QueryInterface(IOleObject, OleObj) = S_OK) and
    (OleObj.GetClientSite(Site) = S_OK) and (Site <> nil) then
    Result := (Site as IAmbientDispatch).UserMode;
end;
```

Лицензирование элемента управления

Ранее в этой главе уже упоминалось, что по умолчанию среда разработки DAX применяет для лицензирования `.lic`-файл, который распространяется вместе с `.ocx`-файлом элемента управления ActiveX. Как уже указывалось, строка лицензии — один из параметров конструктора фабрики класса элемента управления ActiveX. Если в окне мастера установлен флажок Make Control License, генерируется строка GUID, которая будет вставлена в вызовы конструктора и `.lic`-файла (позднее эту строку можно модифицировать). При использовании элемента управления во время разработки среда DAX будет сравнивать строку лицензии в фабрике класса со строкой в `.lic`-файле. Если эти строки окажутся идентичными, будет создан экземпляр элемента управления. При компиляции приложения, которое включает лицензированный элемент управления ActiveX, строка лицензии будет внедрена в приложение, и `.lic`-файл для запуска приложения не потребуется.

Использование `.lic`-файла — не единственный способ лицензирования элемента управления ActiveX. Например, некоторые разработчики считают использование дополнительного файла слишком обременительным и предпочитают хранить лицензионный ключ в системном реестре. К счастью, среда разработки DAX позволяет реализовывать альтернативные схемы лицензирования. Проверка лицензии выполняется в методе `HasMashineLicense()` класса `TActiveXControlFactory`. По умолчанию этот метод пытается просматривать строку лицензирования в `.lic`-файле, однако его можно заставить выполнять при проверке лицензии и любые другие действия. Например, в листинге 25.4 показан класс, производный от `TActiveXControlFactory`, способный выполнять поиск ключа лицензирования в системном реестре.

Листинг 25.4. Альтернативная схема лицензирования элемента управления

```
{ TRegLicAxControlFactory }

type
  TRegLicActiveXControlFactory = class(TActiveXControlFactory)
  protected
    function HasMachineLicense: Boolean; override;
  end;

function TRegLicActiveXControlFactory.HasMachineLicense: Boolean;
var
  Reg: TRegistry;
begin
  Result := True;
  if not SupportsLicensing then Exit;
  Reg := TRegistry.Create;
  try
    Reg.RootKey := HKEY_CLASSES_ROOT;
    { Элемент управления лицензирован, если в системном реестре найден ключ}
    Result := Reg.OpenKey('\Licenses\' + LicString, False);
```

```
  finally
    Reg.Free;
  end;
end;
```

Файл реестра (.reg-файл) может использоваться для помещения лицензионного ключа в системный реестр на лицензируемом компьютере. Текст этого файла приведен в листинге 25.5.

Листинг 25.5. REG-файл лицензии

```
REGEDIT4

[HKEY_CLASSES_ROOT\Licenses\{C06EFEA0-06B2-11D1-A9BF-B18A9F703311}]
@= "Licensing info for DDG demo ActiveX control"
```

Страницы свойств

Страницы свойств позволяют модифицировать свойства элемента управления ActiveX с помощью специального диалогового окна. Страницы свойств элемента управления добавляются как вкладки в многостраничное диалоговое окно, которое создается элементом управления ActiveX. Диалоговое окно со страницами свойств обычно вызывается из контекстного меню, создаваемого основным контейнером элемента управления. Доступ к этому контекстному меню можно получить с помощью щелчка правой кнопкой мыши.

Стандартные страницы свойств

Среда разработки DAX предлагает стандартные страницы свойств для свойств типов IStrings, IPicture, TColor и IFont. Идентификаторы CLSID этих вкладок свойств находятся в модуле AxCtrls. Они объявляются следующим образом:

```
const
  { Идентификаторы CLSID страниц свойств Delphi }
  Class_DColorPropPage: TGUID = '{5CFF5D59-5946-11D0-BDEF-00A024D1875C}';
  Class_DFontPropPage: TGUID = '{5CFF5D5B-5946-11D0-BDEF-00A024D1875C}';
  Class_DPicturePropPage: TGUID = '{5CFF5D5A-5946-11D0-BDEF-00A024D1875C}';
  Class_DStringPropPage: TGUID = '{F42D677E-754B-11D0-BDFB-00A024D1875C}';
```

Использовать любую из этих страниц свойств в элементе управления очень просто: достаточно передать один из этих идентификаторов CLSID в качестве параметра процедуре DefinePropertyPage() в методе DefinePropertyPages() элемента управления ActiveX, как показано в приведенном ниже фрагменте.

```
procedure TMemoX.DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage);
begin
  DefinePropertyPage(Class_DColorPropPage);
  DefinePropertyPage(Class_DFontPropPage);
  DefinePropertyPage(Class_DStringPropPage);
end;
```

На рис. 25.4–25.7 показаны стандартные страницы свойств среды разработки DAX.

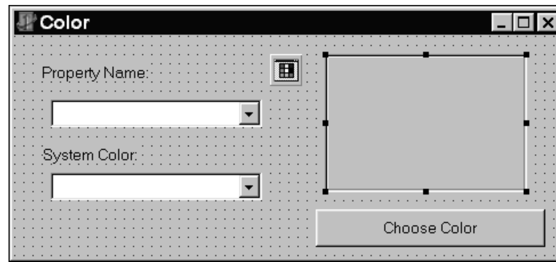


Рис. 25.4. Стандартная страница свойств Colors

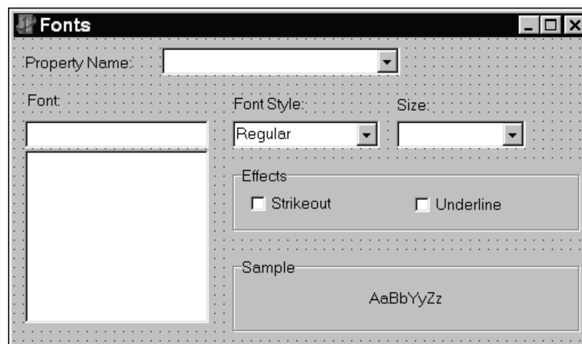


Рис. 25.5. Стандартная страница свойств Fonts

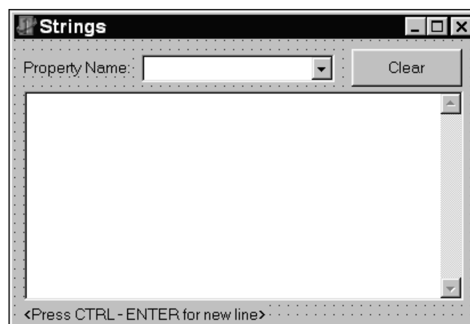


Рис. 25.6. Стандартная страница свойств Strings

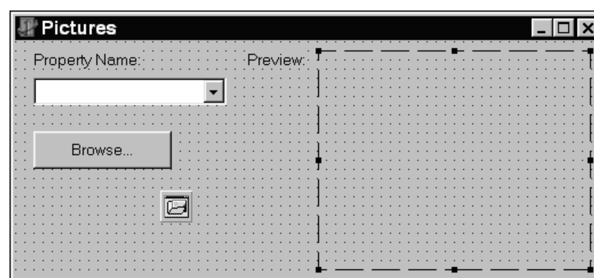


Рис. 25.7. Стандартная страница свойств Pictures

Каждая из этих страниц свойств работает одинаково: в раскрывающемся списке собраны имена каждого свойства определенного типа. Необходимо выбрать имя свойства, установить значение в диалоговом окне, а затем щелкнуть на кнопке OK для модификации выбранного свойства.

На заметку

Если вам необходимо использовать стандартные страницы свойств среды разработки DAX, то файл `StdVcl40.dll` следует передавать вместе с `.ocx`-файлом. Как уже отмечалось выше, в этом файле содержатся определения интерфейсов `IStrings`, `IProvider` и `IDataBroker`. Кроме того, в файле `StdVcl40.dll` содержится реализация каждой страницы свойств среды разработки DAX. Необходимо также зарегистрировать файл `StdVcl40.dll` и `.ocx`-файл на компьютере клиента.

Пользовательские страницы свойств

Для иллюстрации процесса создания пользовательских страниц свойств будет разработан элемент управления, более интересный, чем рассмотренный ранее в этой главе пример элемента управления `Memo`. В листинге 25.6 приведен файл реализации элемента управления ActiveX `TCardX`. Он инкапсулирует элемент управления VCL игральной карты, взятого из модуля `Cards`, который можно найти в подкаталоге `\Code\Comps` на прилагаемом компакт-диске.

Листинг 25.6. Файл `CardImpl.pas` — файл реализации элемента управления ActiveX `TCardX`

```
unit CardImpl;

interface

uses
  Windows, ActiveX, Classes, Controls, Graphics, Menus, Forms,
  StdCtrls, ComServ, StdVCL, AXCtrls, AxCard_TLB, Cards;

type
  TCardX = class(TActiveXControl, ICardX)
  private
    { Закрытые объявления }
    FDelphiControl: TCard;
    FEvents: ICardXEvents;
    procedure ClickEvent(Sender: TObject);
    procedure DblClickEvent(Sender: TObject);
    procedure KeyPressEvent(Sender: TObject; var Key: Char);
  protected
    { Защищенные объявления }
    procedure DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage);
      override;
    procedure EventSinkChanged(const EventSink: IUnknown); override;
    procedure InitializeControl; override;
    function ClassNameIs(const Name: WideString): WordBool; safecall;
    function DrawTextBiDiModeFlags(Flags: Integer): Integer; safecall;
    function DrawTextBiDiModeFlagsReadingOnly: Integer; safecall;
    function Get_BackColor: OLE_COLOR; safecall;
```

```
    function Get_BiDiMode: TxBiDiMode; safecall;
    function Get_Color: OLE_COLOR; safecall;
    function Get_Cursor: Smallint; safecall;
    function Get_DoubleBuffered: WordBool; safecall;
    function Get_DragCursor: Smallint; safecall;
    function Get_DragMode: TxDragMode; safecall;
    function Get_Enabled: WordBool; safecall;
    function Get_FaceUp: WordBool; safecall;
    function Get_ParentColor: WordBool; safecall;
    function Get_Suit: TxCardSuit; safecall;
    function Get_Value: TxCardValue; safecall;
    function Get_Visible: WordBool; safecall;
    function GetControlsAlignment: TxAlignment; safecall;
    function IsRightToLeft: WordBool; safecall;
    function UseRightToLeftAlignment: WordBool; safecall;
    function UseRightToLeftReading: WordBool; safecall;
    function UseRightToLeftScrollBar: WordBool; safecall;
    procedure FlipChildren(AllLevels: WordBool); safecall;
    procedure InitiateAction; safecall;
    procedure Set_BackColor(Value: OLE_COLOR); safecall;
    procedure Set_BiDiMode(Value: TxBiDiMode); safecall;
    procedure Set_Color(Value: OLE_COLOR); safecall;
    procedure Set_Cursor(Value: Smallint); safecall;
    procedure Set_DoubleBuffered(Value: WordBool); safecall;
    procedure Set_DragCursor(Value: Smallint); safecall;
    procedure Set_DragMode(Value: TxDragMode); safecall;
    procedure Set_Enabled(Value: WordBool); safecall;
    procedure Set_FaceUp(Value: WordBool); safecall;
    procedure Set_ParentColor(Value: WordBool); safecall;
    procedure Set_Suit(Value: TxCardSuit); safecall;
    procedure Set_Value(Value: TxCardValue); safecall;
    procedure Set_Visible(Value: WordBool); safecall;
    procedure AboutBox; safecall;
  end;

implementation

uses ComObj, About, CardPP;

{ TCardX }

procedure TCardX.DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage);
begin
  DefinePropertyPage(Class_DColorPropPage);
  DefinePropertyPage(Class_CardPropPage);
end;

procedure TCardX.EventSinkChanged(const EventSink: IUnknown);
begin
  FEvents := EventSink as ICardXEvents;
```

```
end;

procedure TCardX.InitializeControl;
begin
  FDelphiControl := Control as TCard;
  FDelphiControl.OnClick := ClickEvent;
  FDelphiControl.OnDblClick := DblClickEvent;
  FDelphiControl.OnKeyPress := KeyPressEvent;
end;

function TCardX.ClassNameIs(const Name: WideString): WordBool;
begin
  Result := FDelphiControl.ClassNameIs(Name);
end;

function TCardX.DrawTextBiDiModeFlags(Flags: Integer): Integer;
begin
  Result := FDelphiControl.DrawTextBiDiModeFlags(Flags);
end;

function TCardX.DrawTextBiDiModeFlagsReadingOnly: Integer;
begin
  Result := FDelphiControl.DrawTextBiDiModeFlagsReadingOnly;
end;

function TCardX.Get_BackColor: OLE_COLOR;
begin
  Result := OLE_COLOR(FDelphiControl.BackColor);
end;

function TCardX.Get_BiDiMode: TxBiDiMode;
begin
  Result := Ord(FDelphiControl.BiDiMode);
end;

function TCardX.Get_Color: OLE_COLOR;
begin
  Result := OLE_COLOR(FDelphiControl.Color);
end;

function TCardX.Get_Cursor: Smallint;
begin
  Result := Smallint(FDelphiControl.Cursor);
end;

function TCardX.Get_DoubleBuffered: WordBool;
begin
  Result := FDelphiControl.DoubleBuffered;
end;
```

```
function TCardX.Get_DragCursor: Smallint;
begin
  Result := Smallint(FDelphiControl.DragCursor);
end;

function TCardX.Get_DragMode: TxDragMode;
begin
  Result := Ord(FDelphiControl.DragMode);
end;

function TCardX.Get_Enabled: WordBool;
begin
  Result := FDelphiControl.Enabled;
end;

function TCardX.Get_FaceUp: WordBool;
begin
  Result := FDelphiControl.FaceUp;
end;

function TCardX.Get_ParentColor: WordBool;
begin
  Result := FDelphiControl.ParentColor;
end;

function TCardX.Get_Suit: TxCardSuit;
begin
  Result := Ord(FDelphiControl.Suit);
end;

function TCardX.Get_Value: TxCardValue;
begin
  Result := Ord(FDelphiControl.Value);
end;

function TCardX.Get_Visible: WordBool;
begin
  Result := FDelphiControl.Visible;
end;

function TCardX.GetControlsAlignment: TxAlignment;
begin
  Result := TxAlignment(FDelphiControl.GetControlsAlignment);
end;

function TCardX.IsRightToLeft: WordBool;
begin
  Result := FDelphiControl.IsRightToLeft;
end;
```

```
function TCardX.UseRightToLeftAlignment: WordBool;
begin
  Result := FDelphiControl.UseRightToLeftAlignment;
end;

function TCardX.UseRightToLeftReading: WordBool;
begin
  Result := FDelphiControl.UseRightToLeftReading;
end;

function TCardX.UseRightToLeftScrollBar: WordBool;
begin
  Result := FDelphiControl.UseRightToLeftScrollBar;
end;

procedure TCardX.FlipChildren(AllLevels: WordBool);
begin
  FDelphiControl.FlipChildren(AllLevels);
end;

procedure TCardX.InitiateAction;
begin
  FDelphiControl.InitiateAction;
end;

procedure TCardX.Set_BackColor(Value: OLE_COLOR);
begin
  FDelphiControl.BackColor := TColor(Value);
end;

procedure TCardX.Set_BiDiMode(Value: TxBiDiMode);
begin
  FDelphiControl.BiDiMode := TBiDiMode(Value);
end;

procedure TCardX.Set_Color(Value: OLE_COLOR);
begin
  FDelphiControl.Color := TColor(Value);
end;

procedure TCardX.Set_Cursor(Value: Smallint);
begin
  FDelphiControl.Cursor := TCursor(Value);
end;

procedure TCardX.Set_DoubleBuffered(Value: WordBool);
begin
  FDelphiControl.DoubleBuffered := Value;
end;
```

```
procedure TCardX.Set_DragCursor(Value: Smallint);
begin
  FDelphiControl.DragCursor := TCursor(Value);
end;

procedure TCardX.Set_DragMode(Value: TxDragMode);
begin
  FDelphiControl.DragMode := TDragMode(Value);
end;

procedure TCardX.Set_Enabled(Value: WordBool);
begin
  FDelphiControl.Enabled := Value;
end;

procedure TCardX.Set_FaceUp(Value: WordBool);
begin
  FDelphiControl.FaceUp := Value;
end;

procedure TCardX.Set_ParentColor(Value: WordBool);
begin
  FDelphiControl.ParentColor := Value;
end;

procedure TCardX.Set_Suit(Value: TxCardSuit);
begin
  FDelphiControl.Suit := TCardSuit(Value);
end;

procedure TCardX.Set_Value(Value: TxCardValue);
begin
  FDelphiControl.Value := TCardValue(Value);
end;

procedure TCardX.Set_Visible(Value: WordBool);
begin
  FDelphiControl.Visible := Value;
end;

procedure TCardX.ClickEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnClick;
end;

procedure TCardX.DblClickEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnDblClick;
end;
```

```
procedure TCardX.KeyPressEvent(Sender: TObject; var Key: Char);
var
  TempKey: Smallint;
begin
  TempKey := Smallint(Key);
  if FEvents <> nil then FEvents.OnKeyPress(TempKey);
  Key := Char(TempKey);
end;

procedure TCardX.AboutBox;
begin
  ShowCardXAbout;
end;

initialization
  TActiveXControlFactory.Create(ComServer, TCardX, TCard, Class_CardX, 1, '', 0,
tmApartment);
end.
```

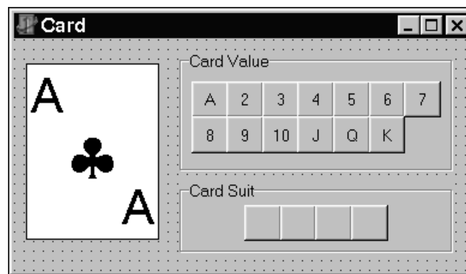


Рис. 25.8. Страница свойств в окне Form Designer

Этот модуль был полностью сгенерирован с помощью мастера, за исключением двух строк кода, содержащихся в методе `DefinePropertyPages()`. В этом методе стандартная страница свойств VCL Color дополнена пользовательской страницей свойств, идентификатор CLSID которой определен как `Class_CardPropPage`. Эта страница свойств была создана с помощью элемента Property Page, выбранного во вкладке ActiveX диалогового окна New Items. На рис. 25.8 показана страница свойств в окне Form Designer, а в листинге 25.7 приведен ее исходный код.

Листинг 25.7. Модуль страницы свойств `CardPP.pas`

```
unit CardPP;

interface

uses SysUtils, Windows, Messages, Classes, Graphics, Controls,
  StdCtrls, ExtCtrls, Forms, ComServ, ComObj, StdVcl, AxCtrls,
  Buttons, Cards, AxCard_TLB;

type
  TCardPropPage = class(TPropertyPage)
    Card1: TCard;
```

```
    ValueGroup: TGroupBox;
    SpeedButton1: TSpeedButton;
    SpeedButton2: TSpeedButton;
    SpeedButton3: TSpeedButton;
    SpeedButton4: TSpeedButton;
    SpeedButton5: TSpeedButton;
    SpeedButton6: TSpeedButton;
    SpeedButton7: TSpeedButton;
    SpeedButton8: TSpeedButton;
    SpeedButton9: TSpeedButton;
    SpeedButton10: TSpeedButton;
    SpeedButton11: TSpeedButton;
    SpeedButton12: TSpeedButton;
    SuitGroup: TGroupBox;
    SpeedButton13: TSpeedButton;
    SpeedButton14: TSpeedButton;
    SpeedButton15: TSpeedButton;
    SpeedButton16: TSpeedButton;
    SpeedButton17: TSpeedButton;
    procedure FormCreate(Sender: TObject);
    procedure SpeedButton1Click(Sender: TObject);
  protected
    procedure UpdatePropertyPage; override;
    procedure UpdateObject; override;
  end;

const
  Class_CardPropPage: TGUID = '{C06EFEA1-06B2-11D1-A9BF-B18A9F703311}';

implementation

{$R *.DFM}

procedure TCardPropPage.UpdatePropertyPage;
var
  i: Integer;
  AValue, ASuit: Integer;
begin
  // Получение масти и достоинства
  AValue := OleObject.Value;
  ASuit := OleObject.Suit;
  // Установка правильной карты
  Card1.Value := TCardValue(AValue);
  Card1.Suit := TCardSuit(ASuit);
  // Установка правильного значения
  with ValueGroup do
    for i := 0 to ControlCount - 1 do
      if (Controls[i] is TSpeedButton) and
        (TSpeedButton(Controls[i]).Tag = AValue) then
           TSpeedButton(Controls[i]).Down := True;
  // Установка правильной масти
```

```
  with SuitGroup do
    for i := 0 to ControlCount - 1 do
      if (Controls[i] is TSpeedButton) and
        (TSpeedButton(Controls[i]).Tag = ASuit) then
          TSpeedButton(Controls[i]).Down := True;
end;

procedure TCardPropPage.UpdateObject;
var
  i: Integer;
begin
  // Установка правильного значения
  with ValueGroup do
    for i := 0 to ControlCount - 1 do
      if (Controls[i] is TSpeedButton) and TSpeedButton(Controls[i]).Down then
      begin
        OleObject.Value := TSpeedButton(Controls[i]).Tag;
        Break;
      end;
  // Установка правильной масти
  with SuitGroup do
    for i := 0 to ControlCount - 1 do
      if (Controls[i] is TSpeedButton) and TSpeedButton(Controls[i]).Down then
      begin
        OleObject.Suit := TSpeedButton(Controls[i]).Tag;
        Break;
      end;
end;

procedure TCardPropPage.FormCreate(Sender: TObject);
const
  // Порядковое значение символа "масть" в шрифте Symbol:
  SSuits: PChar = #167#168#169#170;
var
  i: Integer;
begin
  // Установка достоинства карты на кнопке с помощью
  // прописной буквы шрифта Symbol
  with SuitGroup do
    for i := 0 to ControlCount - 1 do
      if Controls[i] is TSpeedButton then
        TSpeedButton(Controls[i]).Caption := SSuits[i];
end;

procedure TCardPropPage.SpeedButton1Click(Sender: TObject);
begin
  if Sender is TSpeedButton then
  begin
    with TSpeedButton(Sender) do
    begin
      if Parent = ValueGroup then
```

```
        Card1.Value := TCardValue(Tag)
      else if Parent = SuitGroup then
        Card1.Suit := TCardSuit(Tag);
    end;
    Modified;
  end;
end;

initialization
  TActiveXPropertyPageFactory.Create(
    ComServer,
    TCardPropPage,
    Class_CardPropPage);
end.
```

Для взаимодействия с элементом управления ActiveX со страницы свойств следует обратиться к ее полю `OleObject`. В этом поле, имеющем тип `Variant`, содержится ссылка на интерфейс `IDispatch` элемента управления. Методы `UpdatePropertyPage()` и `UpdateObject()` генерируются мастером. Метод `UpdatePropertyPage()` вызывается при активизации страницы свойств. В этом методе содержимое страницы необходимо установить в соответствии с текущими значениями элемента управления ActiveX, как указано в свойстве `OleObject`. Метод `UpdateObject()` вызывается по щелчку на кнопке OK или Apply в диалоговом окне страницы свойств. В этом методе для установки свойств элемента управления ActiveX, отображенных на странице свойств, необходимо использовать свойство `OleObject`.

Рис. 25.9. Страница свойств Card *в действии*

В рассматриваемом примере с помощью страницы свойств элемента управления ActiveX `TCardX` можно отредактировать масть карты или ее достоинство. При изменении масти карты или ее достоинства с помощью кнопок диалогового окна элемент управления VCL `TCard`, находящийся на странице свойств, отображает текущую карту. Обратите внимание также на то, что по щелчку на кнопке вызывается процедура `Modified()` страницы свойств, устанавливающая флаг модификации диалогового окна страницы свойств. При этом в диалоговом окне становится доступной кнопка Apply.

Рассматриваемая страница свойств показана на рис. 25.9.

Активные формы ActiveForm

Формы ActiveForm функционально подобны элементам управления ActiveX, которые рассматривались ранее в этой главе. Основным отличием является то, что элементы управления VCL, на которых основаны элементы управления ActiveX, после запуска мастера не могут быть изменены. А основное преимущество формы ActiveForm состоит в том, что ею может управлять разработчик. Поскольку мастер создания форм ActiveForm и среда разработки DAX очень похожи, то нет смысла повторять уже известные сведения. Вместо этого рассмотрим несколько интересных особенностей, которые свойственны формам ActiveForm.

Добавление свойств к формам ActiveForm

При использовании форм ActiveForm существует одна проблема: их представление в библиотеке типов состоит из "плоских" интерфейсов, а не из вложенных компонентов, уже знакомых по библиотеке VCL. Это означает, что к форме с несколькими кнопками нельзя обратиться привычным при работе с библиотекой VCL способом — *ActiveForm.Button.ButtonProperty*. Самый простой путь решения этой задачи предполагает рассмотрение свойств кнопок как свойств самой формы ActiveForm. Среда разработки DAX самостоятельно добавляет свойства к форме ActiveForm, а разработчику необходимо выполнить лишь несколько действий. Например, для размещения свойства кнопки Caption выполните следующие действия.

1. Добавьте новое размещаемое свойство к объявлению формы ActiveForm в файле реализации. Это свойство должно называться ButtonCaption и иметь методы чтения и записи, модифицирующие свойство Caption кнопки.
2. Добавьте новое свойство с тем же именем к интерфейсу формы ActiveForm, находящейся в библиотеке типов. Delphi должна автоматически создать каркас методов чтения и записи этого свойства, а вам необходимо реализовать чтение и запись свойства ButtonCaption формы ActiveForm.

Файл реализации этого компонента приведен в листинге 25.8.

Листинг 25.8. Добавление свойств к форме ActiveForm

```
unit AFImpl;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ActiveX, AxCtrls, AFrm_TLB, StdCtrls;

type
  TActiveFormX = class(TActiveForm, IActiveFormX)
    Button1: TButton;
  private
    { Закрытые объявления }
```

```
    FEvents: IActiveFormXEvents;
    procedure ActivateEvent(Sender: TObject);
    procedure ClickEvent(Sender: TObject);
    procedure CreateEvent(Sender: TObject);
    procedure DblClickEvent(Sender: TObject);
    procedure DeactivateEvent(Sender: TObject);
    procedure DestroyEvent(Sender: TObject);
    procedure KeyPressEvent(Sender: TObject; var Key: Char);
    procedure PaintEvent(Sender: TObject);
    function GetButtonCaption: string;
    procedure SetButtonCaption(const Value: string);
  protected
    { Защищенные объявления }
    procedure DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage);
      override;
    procedure EventSinkChanged(const EventSink: IUnknown); override;
    function Get_Active: WordBool; safecall;
    function Get_AutoScroll: WordBool; safecall;
    function Get_AutoSize: WordBool; safecall;
    function Get_AxBorderStyle: TxActiveFormBorderStyle; safecall;
    function Get_BiDiMode: TxBiDiMode; safecall;
    function Get_Caption: WideString; safecall;
    function Get_Color: OLE_COLOR; safecall;
    function Get_Cursor: Smallint; safecall;
    function Get_DoubleBuffered: WordBool; safecall;
    function Get_DropTarget: WordBool; safecall;
    function Get_Enabled: WordBool; safecall;
    function Get_Font: IFontDisp; safecall;
    function Get_HelpFile: WideString; safecall;
    function Get_KeyPreview: WordBool; safecall;
    function Get_PixelsPerInch: Integer; safecall;
    function Get_PrintScale: TxPrintScale; safecall;
    function Get_Scaled: WordBool; safecall;
    function Get_Visible: WordBool; safecall;
    procedure _Set_Font(const Value: IFontDisp); safecall;
    procedure AboutBox; safecall;
    procedure Set_AutoScroll(Value: WordBool); safecall;
    procedure Set_AutoSize(Value: WordBool); safecall;
    procedure Set_AxBorderStyle(Value: TxActiveFormBorderStyle); safecall;
    procedure Set_BiDiMode(Value: TxBiDiMode); safecall;
    procedure Set_Caption(const Value: WideString); safecall;
    procedure Set_Color(Value: OLE_COLOR); safecall;
    procedure Set_Cursor(Value: Smallint); safecall;
    procedure Set_DoubleBuffered(Value: WordBool); safecall;
    procedure Set_DropTarget(Value: WordBool); safecall;
    procedure Set_Enabled(Value: WordBool); safecall;
    procedure Set_Font(var Value: IFontDisp); safecall;
    procedure Set_HelpFile(const Value: WideString); safecall;
    procedure Set_KeyPreview(Value: WordBool); safecall;
    procedure Set_PixelsPerInch(Value: Integer); safecall;
    procedure Set_PrintScale(Value: TxPrintScale); safecall;
```

```
    procedure Set_Scaled(Value: WordBool); safecall;
    procedure Set_Visible(Value: WordBool); safecall;
    function Get_ButtonCaption: WideString; safecall;
    procedure Set_ButtonCaption(const Value: WideString); safecall;
  public
    { Открытые объявления }
    procedure Initialize; override;
  published
    property ButtonCaption: string read GetButtonCaption write
      SetButtonCaption;
  end;

implementation

uses ComObj, ComServ, About1;

{$R *.DFM}

{ TActiveFormX }

procedure TActiveFormX.DefinePropertyPages(DefinePropertyPage:
  TDefinePropertyPage);
begin
  { Здесь нужно определить страницы свойств. Страницы свойств
    определяются с помощью вызова DefinePropertyPage с идентификатором
    класса страницы. Например, DefinePropertyPage(Class_ActiveFormXPage); }
end;

procedure TActiveFormX.EventSinkChanged(const EventSink: IUnknown);
begin
  FEvents := EventSink as IActiveFormXEvents;
end;

procedure TActiveFormX.Initialize;
begin
  inherited Initialize;
  OnActivate := ActivateEvent;
  OnClick := ClickEvent;
  OnCreate := CreateEvent;
  OnDblClick := DblClickEvent;
  OnDeactivate := DeactivateEvent;
  OnDestroy := DestroyEvent;
  OnKeyPress := KeyPressEvent;
  OnPaint := PaintEvent;
end;

function TActiveFormX.Get_Active: WordBool;
begin
  Result := Active;
end;
```

```
function TActiveFormX.Get_AutoScroll: WordBool;
begin
  Result := AutoScroll;
end;

function TActiveFormX.Get_AutoSize: WordBool;
begin
  Result := AutoSize;
end;

function TActiveFormX.Get_AxBorderStyle: TxActiveFormBorderStyle;
begin
  Result := Ord(AxBorderStyle);
end;

function TActiveFormX.Get_BiDiMode: TxBiDiMode;
begin
  Result := Ord(BiDiMode);
end;

function TActiveFormX.Get_Caption: WideString;
begin
  Result := WideString(Caption);
end;

function TActiveFormX.Get_Color: OLE_COLOR;
begin
  Result := OLE_COLOR(Color);
end;

function TActiveFormX.Get_Cursor: Smallint;
begin
  Result := Smallint(Cursor);
end;

function TActiveFormX.Get_DoubleBuffered: WordBool;
begin
  Result := DoubleBuffered;
end;

function TActiveFormX.Get_DropTarget: WordBool;
begin
  Result := DropTarget;
end;

function TActiveFormX.Get_Enabled: WordBool;
begin
  Result := Enabled;
end;

function TActiveFormX.Get_Font: IFontDisp;
```

```
begin
  GetOleFont(Font, Result);
end;

function TActiveFormX.Get_HelpFile: WideString;
begin
  Result := WideString(HelpFile);
end;

function TActiveFormX.Get_KeyPreview: WordBool;
begin
  Result := KeyPreview;
end;

function TActiveFormX.Get_PixelsPerInch: Integer;
begin
  Result := PixelsPerInch;
end;

function TActiveFormX.Get_PrintScale: TxPrintScale;
begin
  Result := Ord(PrintScale);
end;

function TActiveFormX.Get_Scaled: WordBool;
begin
  Result := Scaled;
end;

function TActiveFormX.Get_Visible: WordBool;
begin
  Result := Visible;
end;

procedure TActiveFormX._Set_Font(const Value: IFontDisp);
begin
  SetOleFont(Font, Value);
end;

procedure TActiveFormX.AboutBox;
begin
  ShowActiveFormXAbout;
end;

procedure TActiveFormX.Set_AutoScroll(Value: WordBool);
begin
  AutoScroll := Value;
end;

procedure TActiveFormX.Set_AutoSize(Value: WordBool);
begin
```

```
  AutoSize := Value;
end;

procedure TActiveFormX.Set_AxBorderStyle(Value: TxActiveFormBorderStyle);
begin
  AxBorderStyle := TActiveFormBorderStyle(Value);
end;

procedure TActiveFormX.Set_BiDiMode(Value: TxBiDiMode);
begin
  BiDiMode := TBiDiMode(Value);
end;

procedure TActiveFormX.Set_Caption(const Value: WideString);
begin
  Caption := TCaption(Value);
end;

procedure TActiveFormX.Set_Color(Value: OLE_COLOR);
begin
  Color := TColor(Value);
end;

procedure TActiveFormX.Set_Cursor(Value: Smallint);
begin
  Cursor := TCursor(Value);
end;

procedure TActiveFormX.Set_DoubleBuffered(Value: WordBool);
begin
  DoubleBuffered := Value;
end;

procedure TActiveFormX.Set_DropTarget(Value: WordBool);
begin
  DropTarget := Value;
end;

procedure TActiveFormX.Set_Enabled(Value: WordBool);
begin
  Enabled := Value;
end;

procedure TActiveFormX.Set_Font(var Value: IFontDisp);
begin
  SetOleFont(Font, Value);
end;

procedure TActiveFormX.Set_HelpFile(const Value: WideString);
begin
  HelpFile := String(Value);
```

```
end;

procedure TActiveFormX.Set_KeyPreview(Value: WordBool);
begin
  KeyPreview := Value;
end;

procedure TActiveFormX.Set_PixelsPerInch(Value: Integer);
begin
  PixelsPerInch := Value;
end;

procedure TActiveFormX.Set_PrintScale(Value: TxPrintScale);
begin
  PrintScale := TPrintScale(Value);
end;

procedure TActiveFormX.Set_Scaled(Value: WordBool);
begin
  Scaled := Value;
end;

procedure TActiveFormX.Set_Visible(Value: WordBool);
begin
  Visible := Value;
end;

procedure TActiveFormX.ActivateEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnActivate;
end;

procedure TActiveFormX.ClickEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnClick;
end;

procedure TActiveFormX.CreateEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnCreate;
end;

procedure TActiveFormX.DblClickEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnDblClick;
end;

procedure TActiveFormX.DeactivateEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnDeactivate;
end;
```

```
procedure TActiveFormX.DestroyEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnDestroy;
end;

procedure TActiveFormX.KeyPressEvent(Sender: TObject; var Key: Char);
var
  TempKey: Smallint;
begin
  TempKey := Smallint(Key);
  if FEvents <> nil then FEvents.OnKeyPress(TempKey);
  Key := Char(TempKey);
end;

procedure TActiveFormX.PaintEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnPaint;
end;

function TActiveFormX.GetButtonCaption: string;
begin
  Result := Button1.Caption;
end;

procedure TActiveFormX.SetButtonCaption(const Value: string);
begin
  Button1.Caption := Value;
end;

function TActiveFormX.Get_ButtonCaption: WideString;
begin
  Result := ButtonCaption;
end;

procedure TActiveFormX.Set_ButtonCaption(const Value: WideString);
begin
  ButtonCaption := Value;
end;

initialization
  TActiveFormFactory.Create(ComServer, TActiveFormControl,
    TActiveFormX, Class_ActiveFormX, 1, '', OLEMISC_SIMPLEFRAME
    or OLEMISC_ACTSLIKELABEL, tmApartment);
end.
```

Элементы управления ActiveX в Web

Лучше всего формы ActiveForm использовать в качестве средства распространения небольших приложений через World Wide Web. Небольшие элементы управления ActiveX могут быть применены для улучшения внешнего вида и информативности Web-страниц. Однако

для получения наибольшей отдачи от размещения в Web созданных в Delphi элементов управления ActiveX необходимо разобраться с управлением потоками, вопросами безопасности и взаимодействия с броузером.

Связь с Web-броузером

Поскольку элементы управления ActiveX могут запускаться внутри Web-броузера, имеет смысл сделать так, чтобы Web-броузер предоставлял функции и интерфейсы, которые позволят элементам управления ActiveX им манипулировать. Большинство этих функций и интерфейсов находится в модуле UrlMon. Среди простейших — функции Hlink*XXX*(), позволяющие броузеру переходить к другому адресу. Например, с помощью функций HlinkGoForward() и HlinkGoBack() броузер может перемещаться вперед или назад по стеку ссылок. Использование функции HlinkNavigateString() приведет к тому, что броузер перейдет по заданному адресу URL. Эти функции определены в модуле UrlMon следующим образом:

```
function HlinkGoBack(pUnk: IUnknown): HResult; stdcall;
function HlinkGoForward(pUnk: IUnknown): HResult; stdcall;
function HlinkNavigateString(pUnk: IUnknown; szTarget: PWideChar): HResult;
  stdcall;
```

Для каждой из этих функций параметр pUnk определяет интерфейс IUnknown элемента управления ActiveX. Для элементов управления ActiveX в качестве этого параметра должно передаваться выражение *Control* as IUnknown, а при работе с формами ActiveForm — IUnknown(VclComObject). Параметр szTarget функции HlinkNavigateString() определяет адрес URL, к которому требуется перейти.

При решении более сложных задач можно использовать функцию URLDownloadToFile(), чтобы загрузить на локальный компьютер файл с сервера. Этот метод определяется в модуле UrlMon следующим образом:

```
function URLDownloadToFile(p1: IUnknown; p2: PChar; p3: PChar;
  p4: DWORD; p5: IBindStatusCallback): HResult; stdcall;
```

Понятные названия параметров, не правда ли? Параметр p1 представляет собой интерфейс IUnknown элемента управления ActiveX, подобно параметру pUnk функций Hlink*XXX*(). В качестве параметра p2 должен быть указан адрес URL загружаемого файла. Параметр p3 определяет имя локального файла, который должен быть заполнен данными файла, определяемого параметром p2. В качестве параметра p4 должен быть задан 0; параметр p5 является необязательным указателем на интерфейс IBindStatusCallback. Этот интерфейс можно использовать для получения информации о загрузке файла.

В листинге 25.9 приведен код реализации этих методов для формы ActiveForm, а также пример реализации интерфейса IBindStatusCallback.

Листинг 25.9. Форма ActiveForm, использующая функции модуля UrlMon

```
unit UrlTestMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ActiveX, AxCtrls, UrlTest_TLB, UrlMon,
```

```
  StdCtrls, MPlayer, ExtCtrls, ComCtrls;

type
  TUrlTestForm = class(TActiveForm, IUrlTestForm, IBindStatusCallback)
    GroupBox1: TGroupBox;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    MediaPlayer1: TMediaPlayer;
    Panel1: TPanel;
    Button1: TButton;
    StatusPanel: TPanel;
    ProgressBar1: TProgressBar;
    ServerName: TEdit;
    StaticText1: TStaticText;
    procedure Label1Click(Sender: TObject);
    procedure Label2Click(Sender: TObject);
    procedure Label3Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Закрытые объявления}
    FEvents: IUrlTestFormEvents;
    procedure ActivateEvent(Sender: TObject);
    procedure ClickEvent(Sender: TObject);
    procedure CreateEvent(Sender: TObject);
    procedure DblClickEvent(Sender: TObject);
    procedure DeactivateEvent(Sender: TObject);
    procedure DestroyEvent(Sender: TObject);
    procedure KeyPressEvent(Sender: TObject; var Key: Char);
    procedure PaintEvent(Sender: TObject);
  protected
    { IBindStatusCallback }
    function OnStartBinding(dwReserved: DWORD; pib: IBinding): HResult; stdcall;
    function GetPriority(out nPriority): HResult; stdcall;
    function OnLowResource(reserved: DWORD): HResult; stdcall;
    function OnProgress(ulProgress, ulProgressMax, ulStatusCode: ULONG;
      szStatusText: LPCWSTR): HResult; stdcall;
    function OnStopBinding( hRes: HResult; szError: PWideChar ): HResult;
      stdcall;
    function GetBindInfo(out grfBINDF: DWORD; var bindinfo: TBindInfo):
      HResult; stdcall;
    function OnDataAvailable(grfBSCF: DWORD; dwSize: DWORD; formatetc:
      PFormatEtc; stgmed: PStgMedium): HResult; stdcall;
    function OnObjectAvailable(const iid: TGUID; punk: IUnknown): HResult;
      stdcall;
    { UrlTestForm }
    procedure EventSinkChanged(const EventSink: IUnknown); override;
    procedure Initialize; override;
    function Get_Active: WordBool; safecall;
    function Get_AutoScroll: WordBool; safecall;
```

```
    function Get_AxBorderStyle: TxActiveFormBorderStyle; safecall;
    function Get_Caption: WideString; safecall;
    function Get_Color: OLE_COLOR; safecall;
    function Get_Cursor: Smallint; safecall;
    function Get_DropTarget: WordBool; safecall;
    function Get_Enabled: WordBool; safecall;
    function Get_Font: IFontDisp; safecall;
    function Get_HelpFile: WideString; safecall;
    function Get_KeyPreview: WordBool; safecall;
    function Get_PixelsPerInch: Integer; safecall;
    function Get_PrintScale: TxPrintScale; safecall;
    function Get_Scaled: WordBool; safecall;
    function Get_Visible: WordBool; safecall;
    function Get_WindowState: TxWindowState; safecall;
    procedure Set_AutoScroll(Value: WordBool); safecall;
    procedure Set_AxBorderStyle(Value: TxActiveFormBorderStyle); safecall;
    procedure Set_Caption(const Value: WideString); safecall;
    procedure Set_Color(Color: OLE_COLOR); safecall;
    procedure Set_Cursor(Value: Smallint); safecall;
    procedure Set_DropTarget(Value: WordBool); safecall;
    procedure Set_Enabled(Value: WordBool); safecall;
    procedure Set_Font(const Font: IFontDisp); safecall;
    procedure Set_HelpFile(const Value: WideString); safecall;
    procedure Set_KeyPreview(Value: WordBool); safecall;
    procedure Set_PixelsPerInch(Value: Integer); safecall;
    procedure Set_PrintScale(Value: TxPrintScale); safecall;
    procedure Set_Scaled(Value: WordBool); safecall;
    procedure Set_Visible(Value: WordBool); safecall;
    procedure Set_WindowState(Value: TxWindowState); safecall;
  public
    { Открытые объявления}
  end;

implementation

uses ComObj, ComServ;

{$R *.DFM}

{ TUrlTestForm.IBindStatusCallback }

function TUrlTestForm.OnStartBinding(dwReserved: DWORD; pib: IBinding):
  HResult;
begin
  Result := S_OK;
end;

function TUrlTestForm.GetPriority(out nPriority): HResult;
begin
  HRESULT(Result) := S_OK;
```

```
end;

function TUrlTestForm.OnLowResource(reserved: DWORD): HResult;
begin
  Result := S_OK;
end;

function TUrlTestForm.OnProgress(ulProgress, ulProgressMax,
  ulStatusCode: ULONG; szStatusText: LPCWSTR): HResult; stdcall;
begin
  Result := S_OK;
  ProgressBar1.Max := ulProgressMax;
  ProgressBar1.Position := ulProgress;
  StatusPanel.Caption := szStatusText;
end;

function TUrlTestForm.OnStopBinding(hRes: HResult; szError: PWideChar ):
  HResult;
begin
  Result := S_OK;
  if hRes = S_OK then
  begin
    MediaPlayer1.FileName := 'c:\temp\testavi.avi';
    MediaPlayer1.Open;
    MediaPlayer1.Play;
  end;
end;

function TUrlTestForm.GetBindInfo(out grfBINDF: DWORD; var bindinfo: TBindInfo):
HResult; stdcall;
begin
  Result := S_OK;
end;

function TUrlTestForm.OnDataAvailable(grfBSCF: DWORD; dwSize: DWORD;
  formatetc: PFormatEtc; stgmed: PStgMedium): HResult; stdcall;
begin
  Result := S_OK;
end;

function TUrlTestForm.OnObjectAvailable(const iid: TGUID; punk: IUnknown):
  HResult; stdcall;
begin
  Result := S_OK;
end;

{ TUrlTestForm }

procedure TUrlTestForm.EventSinkChanged(const EventSink: IUnknown);
begin
```

```
  FEvents := EventSink as IUrlTestFormEvents;
end;

procedure TUrlTestForm.Initialize;
begin
  OnActivate := ActivateEvent;
  OnClick := ClickEvent;
  OnCreate := CreateEvent;
  OnDblClick := DblClickEvent;
  OnDeactivate := DeactivateEvent;
  OnDestroy := DestroyEvent;
  OnKeyPress := KeyPressEvent;
  OnPaint := PaintEvent;
end;

function TUrlTestForm.Get_Active: WordBool;
begin
  Result := Active;
end;

function TUrlTestForm.Get_AutoScroll: WordBool;
begin
  Result := AutoScroll;
end;

function TUrlTestForm.Get_AxBorderStyle: TxActiveFormBorderStyle;
begin
  Result := Ord(AxBorderStyle);
end;

function TUrlTestForm.Get_Caption: WideString;
begin
  Result := WideString(Caption);
end;

function TUrlTestForm.Get_Color: OLE_COLOR;
begin
  Result := Color;
end;

function TUrlTestForm.Get_Cursor: Smallint;
begin
  Result := Smallint(Cursor);
end;

function TUrlTestForm.Get_DropTarget: WordBool;
begin
  Result := DropTarget;
end;
```

```
function TUrlTestForm.Get_Enabled: WordBool;
begin
  Result := Enabled;
end;

function TUrlTestForm.Get_Font: IFontDisp;
begin
  GetOleFont(Font, Result);
end;

function TUrlTestForm.Get_HelpFile: WideString;
begin
  Result := WideString(HelpFile);
end;

function TUrlTestForm.Get_KeyPreview: WordBool;
begin
  Result := KeyPreview;
end;

function TUrlTestForm.Get_PixelsPerInch: Integer;
begin
  Result := PixelsPerInch;
end;

function TUrlTestForm.Get_PrintScale: TxPrintScale;
begin
  Result := Ord(PrintScale);
end;

function TUrlTestForm.Get_Scaled: WordBool;
begin
  Result := Scaled;
end;

function TUrlTestForm.Get_Visible: WordBool;
begin
  Result := Visible;
end;

function TUrlTestForm.Get_WindowState: TxWindowState;
begin
  Result := Ord(WindowState);
end;

procedure TUrlTestForm.Set_AutoScroll(Value: WordBool);
begin
  AutoScroll := Value;
end;
```

```
procedure TUrlTestForm.Set_AxBorderStyle(Value: TxActiveFormBorderStyle);
begin
  AxBorderStyle := TActiveFormBorderStyle(Value);
end;

procedure TUrlTestForm.Set_Caption(const Value: WideString);
begin
  Caption := TCaption(Value);
end;

procedure TUrlTestForm.Set_Color(Color: OLE_COLOR);
begin
  Self.Color := Color;
end;

procedure TUrlTestForm.Set_Cursor(Value: Smallint);
begin
  Cursor := TCursor(Value);
end;

procedure TUrlTestForm.Set_DropTarget(Value: WordBool);
begin
  DropTarget := Value;
end;

procedure TUrlTestForm.Set_Enabled(Value: WordBool);
begin
  Enabled := Value;
end;

procedure TUrlTestForm.Set_Font(const Font: IFontDisp);
begin
  SetOleFont(Self.Font, Font);
end;

procedure TUrlTestForm.Set_HelpFile(const Value: WideString);
begin
  HelpFile := String(Value);
end;

procedure TUrlTestForm.Set_KeyPreview(Value: WordBool);
begin
  KeyPreview := Value;
end;

procedure TUrlTestForm.Set_PixelsPerInch(Value: Integer);
begin
  PixelsPerInch := Value;
end;
```

```
procedure TUrlTestForm.Set_PrintScale(Value: TxPrintScale);
begin
  PrintScale := TPrintScale(Value);
end;

procedure TUrlTestForm.Set_Scaled(Value: WordBool);
begin
  Scaled := Value;
end;

procedure TUrlTestForm.Set_Visible(Value: WordBool);
begin
  Visible := Value;
end;

procedure TUrlTestForm.Set_WindowState(Value: TxWindowState);
begin
  WindowState := TWindowState(Value);
end;

procedure TUrlTestForm.ActivateEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnActivate;
end;

procedure TUrlTestForm.ClickEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnClick;
end;

procedure TUrlTestForm.CreateEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnCreate;
end;

procedure TUrlTestForm.DblClickEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnDblClick;
end;

procedure TUrlTestForm.DeactivateEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnDeactivate;
end;

procedure TUrlTestForm.DestroyEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnDestroy;
end;
```

```
procedure TUrlTestForm.KeyPressEvent(Sender: TObject; var Key: Char);
var
  TempKey: Smallint;
begin
  TempKey := Smallint(Key);
  if FEvents <> nil then FEvents.OnKeyPress(TempKey);
  Key := Char(TempKey);
end;

procedure TUrlTestForm.PaintEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnPaint;
end;

procedure TUrlTestForm.Label1Click(Sender: TObject);
begin
  HLinkNavigateString(IUnknown(VCLComObject), 'http://www.inprise.com');
end;

procedure TUrlTestForm.Label2Click(Sender: TObject);
begin
  HLinkGoForward(IUnknown(VCLComObject));
end;

procedure TUrlTestForm.Label3Click(Sender: TObject);
begin
  HLinkGoBack(IUnknown(VCLComObject));
end;

procedure TUrlTestForm.Button1Click(Sender: TObject);
begin
  { Примечание. Имя .avi-файла, указанного в качестве первого параметра
    функции Format, можно заменить на имя другого .avi-файла,
    расположенного на вашем сервере. }
  URLDownloadToFile(IUnknown(VCLComObject),
    PChar(Format('http://%s/delphi3.avi', [ServerName.Text])),
    'c:\temp\testavi.avi', 0, Self);
end;

initialization
  TActiveFormFactory.Create(ComServer, TActiveFormControl, TUrlTestForm,
    Class_UrlTestForm, 1, '', OLEMISC_SIMPLEFRAME or OLEMISC_ACTSLIKELABEL,
    tmApartment);
end.
```

Метод `URLDownloadToFile()` загружает `.avi`-файл с сервера и воспроизводит его в `TMediaPlayer`. Обратите внимание: согласно этому примеру файл `delphi3.avi` должен располагаться в корневом каталоге диска сервера, так что вам, скорее всего, придется изменить расположение `.avi`-файла. На рис. 25.10 показано действие формы ActiveForm в броузере Internet Explorer.

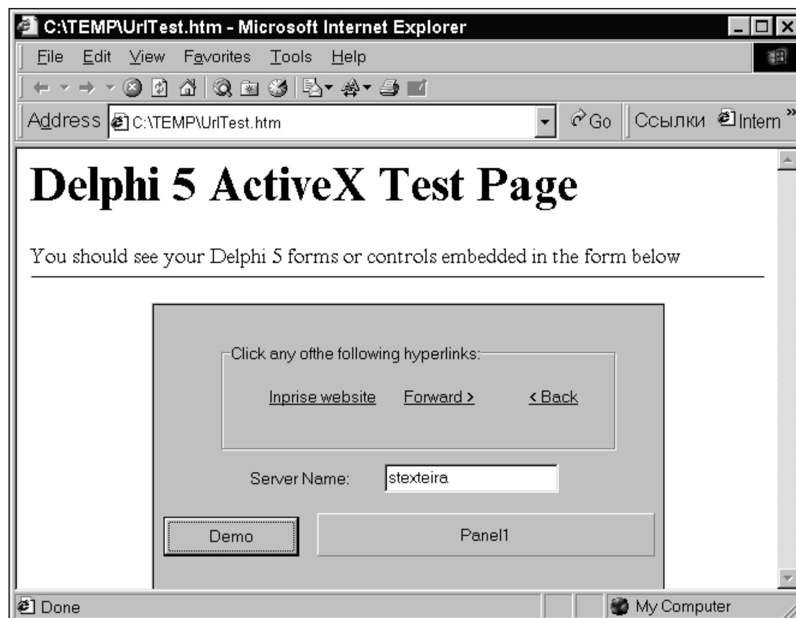


Рис. 25.10. Форма ActiveForm в окне броузера Internet Explorer

Распространение в Web

Интегрированная среда разработки Delphi включает удобные средства размещения проектов ActiveX в Web. Доступ к этим средствам можно получить с помощью команды Project⇨Web Deployment Options главного меню, при выборе которой появится диалоговое окно, показанное на рис. 25.11.

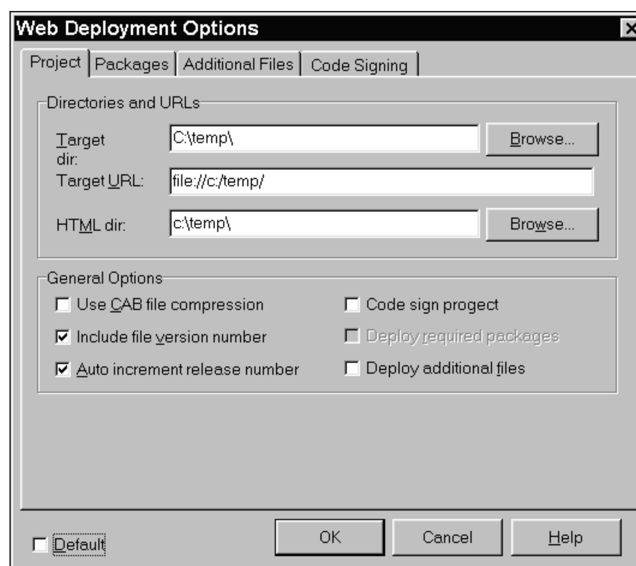


Рис. 25.11. Вкладка Project диалогового окна Web Deployment Options

Вкладка Project

В поле Target dir вкладки Project вводится путь к папке, в которой предполагается разместить проект ActiveX. Обратите внимание на то, что здесь можно использовать устройство на Web-сервере, но содержимое этого поля должно удовлетворять требованиям UNC. Заметьте также, что не нужно вводить имя файла после указания пути к нему.

В поле Target URL помещается адрес URL, который ссылается на папку, определенную в поле Target dir. Это должен быть реальный адрес, который использует стандартный префикс URL (`http://`, `file://`, `ftp://` и т.д.). Опять-таки, в это поле имя файла вводить не нужно.

В поле HTML dir задается еще один путь, указывающий место, куда должен быть скопирован сгенерированный HTML-файл. Чаще всего в этом поле задаются те же значения, что и в поле Target dir.

В этом диалоговом окне имеется также несколько флажков опций для определения дополнительных параметров распространения проекта.

- Use CAB file compression. Установка этого флажка опции приведет к тому, что `.ocx`-файл будет сжат с использованием формата Microsoft Cabinet (CAB). Рекомендуется использовать для элементов управления, которые будут распространяться среди клиентов, имеющих низкоскоростной доступ к Internet.
- Include file version number. Состояние этого флажка опции показывает, будет ли включаться номер версии в генерируемый HTML- или INF-файл. Рекомендуется всегда устанавливать этот флажок, поскольку пользователи могут не загружать элемент управления, если у них уже установлена более поздняя его версия.
- Auto increment release number. При установке этого флажка опции номер выпуска ресурса `VersionInfo` будет автоматически увеличиваться после размещения.
- Code sign project. Если у вас есть сертификат цифровой подписи кода, то с помощью этого флажка опции вы можете позволить интегрированной среде разработки подписывать файл при размещении проекта.

На заметку

Для того чтобы подписать файл, необходимо иметь программу Internet Explorer 3.02 или выше и утилиту Authenticode 2.0 в дополнение к сертификату такого распространителя, как VeriSign.

- Deploy required packages. Если ваш проект построен с применением пакетов, установка этого флажка опции приведет к тому, что в набор распространения автоматически будут включены пакеты, используемые в файле проекта.
- Deploy additional files. При установке этого флажка опции файлы, указанные во вкладке Additional Files, будут добавлены к набору распространения.

Вкладки Packages и Additional Files

Вкладки Packages и Additional Files показаны на рис. 25.12 и 25.13 соответственно. Различие между этими вкладками заключается в том, что поля во вкладке Packages заполняются автоматически на основе выбранных пакетов, используемых в проекте, а поля во вкладке Additional Files необходимо заполнять самостоятельно.

Если во вкладке Project установить флажок опции Use CAB file compression, станут доступными группы параметров во вкладках Packages и Additional Files, позволяющие выбрать способ сжатия (с использованием `.ocx`-файла или отдельного `.cab`-файла). В большин-

стве случаев более эффективно сжимать каждый файл в собственный .cab-файл, поскольку пользователь может не загружать файлы, уже установленные на его компьютере. Ниже приведено несколько параметров, с которыми необходимо ознакомиться.

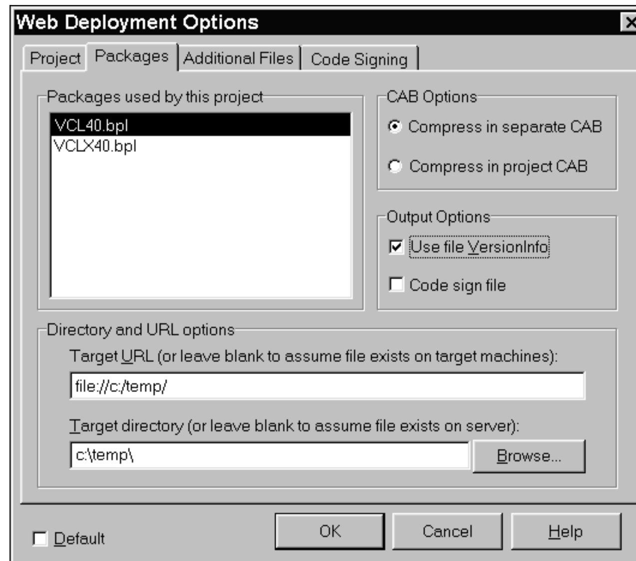


Рис. 25.12. Вкладка Packages

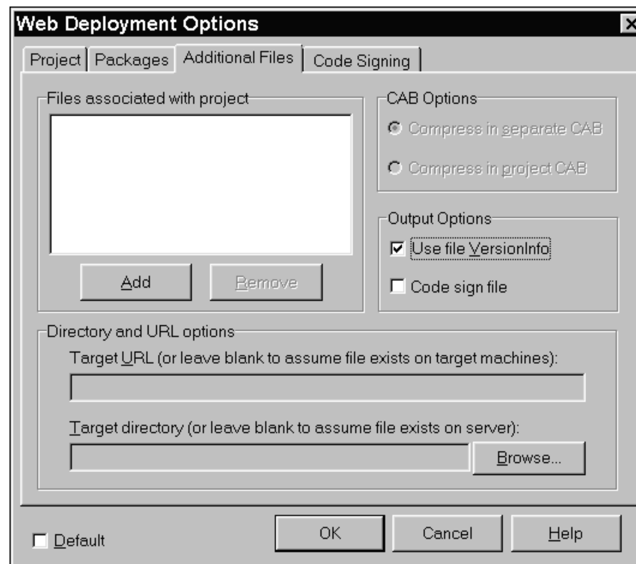


Рис. 25.13. Вкладка Additional Files

- Если установлен флажок опции Use file VersionInfo, средство распространения будет определять наличие параметра `VersionInfo` у выделенного файла. Если этот параметр имеется, номер версии будет заноситься в `.inf`-файл.

- Установка флажка опции Code sign file позволяет добавлять подпись аутентификации к выделенному файлу.
- В поле Target URL по умолчанию автоматически помещается содержимое поля Target URL вкладки Project. Это адрес, откуда может быть загружен файл. Если вы предполагаете, что у пользователя вашего элемента управления ActiveX выбранный файл уже установлен, оставьте поле пустым.
- Поле Target directory позволяет определить папку, в которую должен быть скопирован файл. Оставьте это поле пустым, если файл уже существует на сервере и не должен заново копироваться.

Вкладка Code Signing

Вкладка Code Signing, показанная на рис. 25.14, позволяет указать расположение файла сертификата и закрытого ключа, ассоциированного с этим сертификатом. Кроме того, здесь можно ввести заголовок приложения, URL приложения или организации, используемый тип шифрования, а также информацию об ограничении времени действия сертификата.

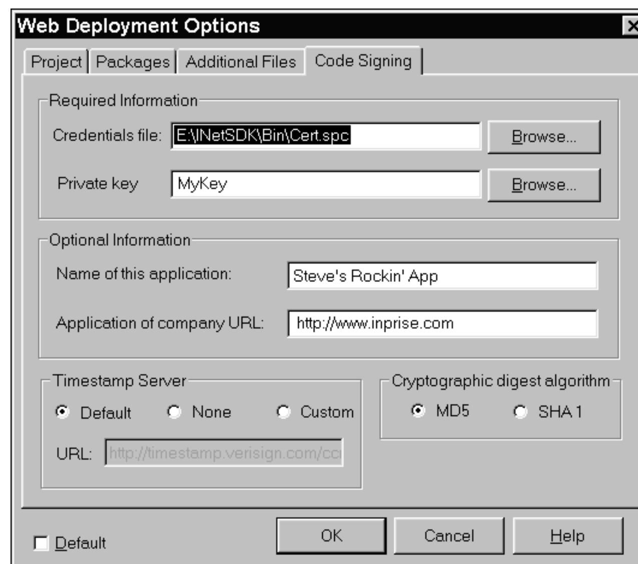


Рис. 25.14. Вкладка Code Signing

Общие советы

Если вы ошибетесь при установке параметров во вкладке Project, то ваш элемент управления появится на Web-странице в виде перечеркнутого прямоугольника. В этом случае необходимо проверить сгенерированные HTML- и INF-файлы (если вы распространяете несколько файлов) на наличие ошибок. Наиболее часто встречающаяся проблема — неверный адрес URL, определенный для данного элемента управления.

Резюме

На этом завершается обсуждение вопросов создания элементов управления ActiveX и форм ActiveForm в Delphi. В этой главе рассмотрены вопросы использования мастера создания элементов управления ActiveX — незаменимого помощника при работе со средой разработки ActiveX. В следующей главе вашему вниманию будет представлено использование программного интерфейса OpenTools API в Delphi. Это позволит еще глубже погрузиться в интегрированную среду разработки.

Глава

26 Использование интерфейса Open Tools API

Интерфейсы Open Tools 455

Использование интерфейса Open Tools API 457

Мастера форм 482

Резюме 489

Наверное, у вас не раз возникал вопрос: “Delphi — великолепный инструмент, но нельзя ли его интегрированную среду разработки дополнить еще одной небольшой функцией?”. Ответ: “Конечно же, можно!” Для этой цели следует использовать интерфейс Open Tools API. Данный интерфейс позволяет создавать дополнительные инструменты, которые можно будет применять в интегрированной среде разработки Delphi. В этой главе речь пойдет о различных интерфейсах, которые создаются с помощью Open Tools API, об их применении, а также о том, как полученными знаниями воспользоваться при создании полнофункционального мастера.

Интерфейсы Open Tools

Интерфейс Open Tools API состоит из восьми модулей, каждый из которых содержит один или несколько объектов, обеспечивающих интерфейс со многими средствами интегрированной среды разработки. С помощью этих интерфейсов можно создавать собственные мастера Delphi, диспетчеры управления версиями, а также компоненты и редакторы свойств. Благодаря таким надстройкам можно значительно повысить мощь среды разработки Delphi.

За исключением интерфейсов, разработанных для компонентов и редакторов свойств, объекты интерфейса Open Tools обеспечивают полностью виртуальный интерфейс с внешним миром. Это означает, что можно использовать лишь виртуальные функции этих объектов. Нельзя получить доступ к полям данных такого объекта, к его свойствам или статическим функциям, поскольку объекты интерфейса Open Tools создаются на основе стандарта COM (см. главу 23, “COM-ориентированные технологии”). После небольшой доработки эти интерфейсы могут использоваться с любым языком программирования, поддерживающим технологию COM. В этой главе рассматривается лишь Delphi, однако помните, что можно использовать и другие языки программирования.

На заметку

Предоставляемые интерфейсом Open Tools API возможности в полной мере можно использовать лишь в версиях Delphi Professional и Enterprise Edition. В версии Standard можно использовать надстройки, созданные с помощью интерфейса Open Tools API, однако их невозможно создавать, поскольку в этой версии содержатся лишь модули для разработки компонентов и редакторов свойств. Исходный код интерфейсов Open Tools можно найти в подкаталоге `\Delphi 5\Source\ToolsAPI`.

В табл. 26.1 приведены модули, которые, в сущности, и составляют интерфейс Open Tools API. Термин *интерфейс* в данном случае не означает встроенный в Delphi тип `interface`. Поскольку Open Tools API обеспечивает поддержку интерфейсов Delphi, в качестве замены “настоящих” интерфейсов он использует обычные классы Delphi с виртуальными абстрактными методами. Использование стандартных интерфейсов в Open Tools API в каждой новой версии Delphi увеличивалось, и текущая версия основывается практически только на них.

Таблица 26.1. Модули интерфейса Open Tools API

Имя модуля	Назначение
`ToolsAPI`	Содержит самые новейшие элементы интерфейса Open Tools API. Этот модуль, по существу, заместил абстрактные классы, применяемые в предыдущих версиях Delphi для управления надстройками меню, системы уведомления, файловой системы, редактора и мастеров. В нем содержатся также новые интерфейсы для управления отладчиком, клавиатурными эквивалентами интегрированной среды разработки, проектами, группами проектов, пакетами и списком To Do

Окончание табл. 26.1

Имя модуля	Назначение
VirtIntf*	Определяет базовый класс TInterface, от которого наследуются другие интерфейсы. В этом модуле определен также класс TIStream, который инкапсулирует VCL-класс TStream
IStreams*	Определяет классы TIMemoryStream, TIFileStream и TIVirtualStream, которые являются потомками класса TIStream. Эти интерфейсы могут использоваться для включения в интегрированную среду разработки своего собственного механизма работы с потоками
ToolIntf*	Определяет классы TIMenuItemIntf и TIMainMenuIntf, которые позволяют разработчику интерфейса Open Tools создавать и модифицировать меню интегрированной среды разработки Delphi. В этом модуле определен также класс TIAddInNotifier, который позволяет создаваемым инструментам-надстройкам получать уведомления о возникновении в среде разработки определенных событий. Следует также отметить, что в этом модуле имеется класс TIToolServices, который обеспечивает взаимодействие между различными элементами интегрированной среды разработки Delphi (такими как редактор, библиотека компонентов, редактор кода, конструктор форм и файловая система)
VCSIntf	Определяет класс TIVCSClient, обеспечивающий взаимодействие интегрированной среды разработки Delphi с программным обеспечением управления версиями
FileIntf*	Определяет класс TIVirtualFileSystem, используемый интегрированной средой разработки Delphi для работы с файлами. Мастера, диспетчеры управления версиями, а также редакторы свойств и компоненты могут использовать этот интерфейс для выполнения в Delphi специальных файловых операций
EditIntf*	Определяет классы, необходимые для управления редактором кода и конструктором форм. Класс TIEditReader предоставляет доступ к буферу редактора "для чтения", а класс TIEditWriter — "для записи". Класс TIEditView предоставляет возможность просмотра буфера редактирования. Класс TIEditInterface — это базовый интерфейс редактора, который может использоваться для доступа к уже упоминавшимся интерфейсам. Класс TIComponentInterface — это интерфейс с отдельным компонентом, помещенным в форму в режиме разработки. Класс TIFormInterface — основной интерфейс с формой или модулем данных в режиме разработки. Класс TIResourceEntry обеспечивает интерфейс с данными в файле ресурсов (*.res) проекта. Класс TIResourceFile определяет высокоуровневый интерфейс с файлом ресурсов проекта. Класс TIModuleNotifier определяет сообщения, используемые при появлении в конкретном модуле различных событий. И, наконец, класс TIModuleInterface обеспечивает интерфейс с любым файлом или модулем, открытым в интегрированной среде разработки
ExptIntf*	Определяет абстрактный класс TIExpert, из которого наследуются все мастера
DsgnIntf	Определяет интерфейс IFormDesigner, а также классы TPropertyEditor и TComponentEditor, которые используются для создания пользовательских свойств и редакторов компонентов

*Все функции этого модуля перенесены в модуль ToolsAPI. Данные модули сохранены лишь с целью обратной совместимости с предыдущими версиями Delphi.

На заметку

У вас может возникнуть закономерный вопрос: "А где в Delphi содержится документация по всем этим мастерам?" На этот счет можно сказать лишь следующее: все документировано, однако поиск необходимой информации представляет собой непростую задачу. В каждом из этих модулей содержится полная документация по определенным в нем классам и методам. Для получения исчерпывающей информации проанализируйте все модули самостоятельно, поскольку нельзя сказать наверняка, в каком модуле какие данные содержатся.

Использование интерфейса Open Tools API

Теперь, когда мы ознакомились с интерфейсом Open Tools API, можно приступить к созданию и анализу работы реальной программы. В этом разделе мы рассмотрим только процесс создания мастера с помощью интерфейса Open Tools API. Построение системы управления версиями не рассматривается, поскольку интерес к этой теме довольно ограничен. Примеры разработки компонентов и редакторов свойств можно найти в главах 21, "Создание пользовательских компонентов в Delphi", и 22, "Сложные методики работы с компонентами".

Мастер Dumb

Для начала создадим очень простой мастер, называемый "Dumb" ("бесполезный"). Для этого достаточно создать класс, реализующий интерфейс `IOTAWizard`. Напомним, что интерфейс `IOTAWizard` определен в модуле `ToolsAPI` следующим образом:

```
type
  IOTAWizard = interface(IOTANotifier)
    ['{B75C0CE0-EEA6-11D1-9504-00608CCBF153}']
    { Строки пользовательского интерфейса мастера }
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    { Запуск мастера }
    procedure Execute;
  end;
```

В основном этот интерфейс состоит из нескольких функций `Get`*`XXX`*`()`, которые предназначены для переопределения в классах-потомках с целью получения информации, специфической для каждого создаваемого мастера. Метод `IOTAWizard.Execute()` вызывается интегрированной средой разработки при выборе пользователем некоторого мастера в главном меню или диалоговом окне New Items. Поэтому в данном методе следует обеспечить создание и вызов соответствующего мастера.

Если провести углубленный анализ, то можно заметить, что интерфейс `IOTAWizard` является производным от другого интерфейса, `IOTANotifier`. Этот интерфейс также определен в модуле `ToolsAPI` и содержит методы, используемые интегрированной средой разработки для уведомления мастера о возникновении различных событий. Интерфейс `IOTANotifier` определен следующим образом:

```
type
  IOTANotifier = interface(IUnknown)
    ['{F17A7BCF-E07D-11D1-AB0B-00C04FB16FB3}']
```

```
    { Эта процедура вызывается сразу же после того, как элемент будет успешно
      сохранен. Для интерфейсов IOTAWizard такой вызов не выполняется }
    procedure AfterSave;
    { Эта процедура вызывается непосредственно перед сохранением элемента.
      Для интерфейсов IOTAWizard такой вызов не выполняется }
    procedure BeforeSave;
    { Разрушение указанного элемента. При этом следует аннулировать
      все ссылки на него. Исключения игнорируются. }
    procedure Destroyed;
    { Эта процедура вызывается при некоторой модификации указанного
      элемента. Для интерфейсов IOTAWizard такой вызов не выполняется }
    procedure Modified;
  end;
```

Как видно из содержащихся в коде комментариев, большинство методов для простых мастеров `IOTAWizard` не вызывается. Поэтому в модуле `ToolsAPI` имеется класс `TNotifierObject`, представляющий собой пустую реализацию методов интерфейса `IOTANotifier`. Если разрабатываемый мастер унаследовать от этого класса, то в распоряжении разработчика будут все его методы.

При создании мастера естественно предположить, что он будет запускаться. Проще всего это осуществить с помощью выбора команды меню. Если команду запуска мастера поместить в главное меню Delphi, то достаточно просто реализовать интерфейс `IOTAMenuWizard`, определенный в модуле `ToolsAPI` следующим образом:

```
type
  IOTAMenuWizard = interface(IOTAWizard)
    ['{B75C0CE2-EEA6-11D1-9504-00608CCBF153}']
    function GetMenuText: string;
  end;
```

Легко заметить, что интерфейс `IOTAMenuWizard` является потомком интерфейса `IOTAWizard`, добавляющим лишь один дополнительный метод, возвращающий текстовую строку меню.

Все вышесказанное обобщается в листинге 26.1. В нем приведен код модуля `DumbWiz.pas`, содержащего класс `TDumbWizard`.

Листинг 26.1. Модуль `Dumb Wiz.pas` — реализация простейшего мастера

```
unit DumbWiz;

interface

uses
  ShareMem, SysUtils, Windows, ToolsAPI;

type
  TDumbWizard = class(TNotifierObject, IOTAWizard, IOTAMenuWizard)
    // Методы интерфейса IOTAWizard
    function GetIDString: string;
    function GetName: string;
```

```
    function GetState: TWizardState;
    procedure Execute;
    // Методы интерфейса IOTAMenuWizard
    function GetMenuText: string;
  end;

procedure Register;

implementation

uses Dialogs;

function TDumbWizard.GetName: string;
begin
  Result := 'Dumb Wizard';
end;

function TDumbWizard.GetState: TWizardState;
begin
  Result := [wsEnabled];
end;

function TDumbWizard.GetIDString: String;
begin
  Result := 'DDG.DumbWizard';
end;

procedure TDumbWizard.Execute;
begin
  MessageDlg('This is a dumb wizard.', mtInformation, [mbOk], 0);
end;

function TDumbWizard.GetMenuText: string;
begin
  Result := 'Dumb Wizard';
end;

procedure Register;
begin
  RegisterPackageWizard(TDumbWizard.Create);
end;

end.
```

Функция `IOTAWizard.GetName()` должна возвращать уникальное имя мастера.

Функция `IOTAWizard.GetState()` возвращает сведения о состоянии мастера `wsStandard` в главном меню. Возвращаемое значение этой функции представляет собой множество, в котором могут содержаться значения `wsEnabled` и/или `wsChecked`, в зависимости от того, как элемент меню будет представляться в интегрированной среде разработки. Эта функция вызывается каждый раз при выводе мастера на экран для корректного отображения меню.

Функция `IOTAWizard.GetIDString()` возвращает глобальный идентификатор мастера, представляющий собой уникальную строку. В соответствии с используемыми соглашениями эта строка должна иметь следующий формат:

```
CompanyName.WizardName
```

Функция `IOTAWizard.Execute()` запускает мастер. Как видно из листинга 26.1, метод `Execute()` класса `TDumbWizard` не выполняет никаких действий. Однако ниже в данной главе будут показаны некоторые мастера, выполняющие реальные действия.

Функция `IOTAWizard.GetMenuText()` возвращает строку текста, которая должна появиться в главном меню. Эта функция вызывается каждый раз при выборе команды меню Help. Таким образом, при запуске мастера можно динамически изменять текст меню.

Обратите внимание на вызов функции `RegisterPackageWizard()` внутри процедуры `Register()`. Заметим, что такой вызов очень напоминает синтаксис вызовов, используемых для регистрации компонентов, редакторов компонентов или редакторов свойств при включении в библиотеку компонентов, как описывалось в главах 21, "Создание пользовательских компонентов в Delphi", и 22, "Сложные методики работы с компонентами". Это связано с тем, что данный тип мастеров хранится в пакете, который является частью библиотеки компонентов. Как будет показано в следующем примере, мастер можно сохранить также в отдельной библиотеке DLL.

Мастер устанавливается точно так же, как и компонент. Выберите в главном меню команду Components⇨Install Component, а затем добавьте модуль к новому или существующему пакету. После выполнения установки при выборе меню Help будет появляться команда меню для запуска мастера (рис. 26.1). Работа этого мастера показана на рис. 26.2.

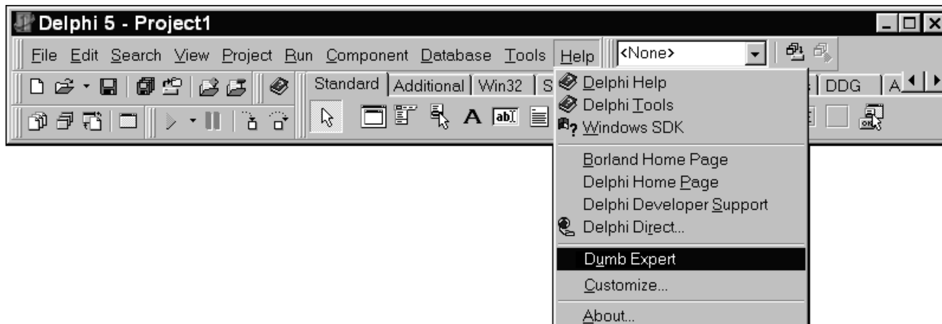


Рис. 26.1. Команда вызова нового мастера в главном меню

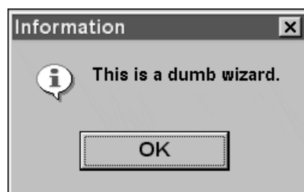


Рис. 26.2. Новый мастер в действии

Мастер Wizard

Процедура создания мастера в виде библиотеки DLL (далее мастер DLL) несколько более трудоемка, чем процедура создания мастера в библиотеке компонентов. Кроме создания мастера в виде библиотеки DLL, в примере мастера "Wizard" иллюстрируется нескольких менее

очевидных моментов. В частности, ниже будет показано, как мастер DLL связан с системным реестром и как один и тот же исходный код использовать для создания мастеров в виде библиотек DLL или выполняемых файлов EXE.

На заметку

Библиотеки DLL более подробно рассматривались в главе 9, "Динамически компонуемые библиотеки" (том I).

Совет

Не существует строго сформулированных правил, определяющих, где должен быть расположен мастер: в пакете библиотеки компонентов или в библиотеке DLL. С точки зрения пользователя, основное различие между этими двумя способами заключается в том, что для установки мастеров в библиотеке компонентов требуется выполнить простую процедуру установки пакета, тогда как для установки мастеров в библиотеке DLL необходимо ввести соответствующие записи в системный реестр. К тому же, для активизации внесенных изменений нужно выйти из среды Delphi, а затем запустить ее повторно. С точки зрения разработчика, также проще иметь дело с мастерами, размещаемыми в пакетах, причем, сразу по нескольким причинам. В частности, исключения будут автоматически передаваться между мастером и интегрированной средой разработки; для управления памятью нет необходимости использовать файл `sharemem.dll`; не нужно выполнять никаких специальных действий для инициализации переменных библиотеки DLL; будут правильно работать все всплывающие подсказки и сообщения мыши.

Принимая во внимание все вышесказанное, можно сделать вывод, что помещать мастер в библиотеку DLL целесообразно только в том случае, если усилия конечного пользователя по его установке должны быть минимальными.

Для того чтобы системой Delphi был распознан мастер, размещаемый в библиотеке DLL, соответствующая запись должна содержаться в следующей ветви системного реестра:

```
HKEY_CURRENT_USER\Software\Borland\Delphi\5.0\Experts
```

На рис. 26.3 эта ветвь показана в окне редактора реестра.

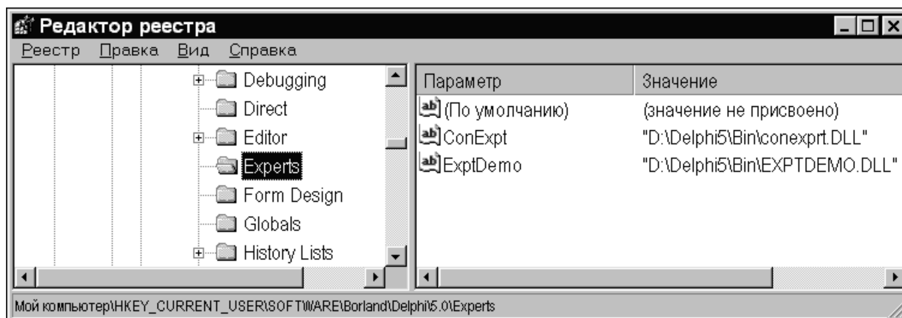


Рис. 26.3. Записи о мастерах Delphi в системном реестре

Интерфейс мастера

Мастер Wizard позволяет без использования редактора реестра добавлять, модифицировать и удалять из системного реестра записи о мастерах, размещенных в библиотеках DLL. Сначала рассмотрим модуль `InitWiz.pas`, содержащий класс мастера (листинг 26.2).

Листинг 26.2. Модуль `InitWiz.pas`, содержащий класс мастера DLL

```
unit InitWiz;

interface

uses Windows, ToolsAPI;

type
  TWizardWizard = class(TNotifierObject, IOTAWizard, IOTAMenuWizard)
    // Методы класса IOTAWizard
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    procedure Execute;
    // Метод класса IOTAMenuWizard
    function GetMenuText: string;
  end;

function InitWizard(const BorlandIDEServices: IBorlandIDEServices;
  RegisterProc: TWizardRegisterProc;
  var Terminate: TWizardTerminateProc): Boolean stdcall;

var
  { Параметры системного реестра, соответствующие мастерам Delphi 5.
    По умолчанию используется мастер EXE, тогда как мастер DLL
    получает параметры из метода ToolServices.GetBaseRegistryKey }
  SDelphiKey: string = '\Software\Borland\Delphi\5.0\Experts';

implementation

uses SysUtils, Forms, Controls, Main;

function TWizardWizard.GetName: string;
{ Возвращает имя мастера }
begin
  Result := 'WizardWizard';
end;

function TWizardWizard.GetState: TWizardState;
{ Этот мастер всегда доступен }
begin
  Result := [wsEnabled];
end;

function TWizardWizard.GetIDString: String;
{ "Vendor.AppName" строка идентификатора мастера }
begin
  Result := 'DDG.WizardWizard';
end;
```

```
function TWizardWizard.GetMenuText: string;
{ Строка меню мастера }
begin
  Result := 'Wizard Wizard';
end;

procedure TWizardWizard.Execute;
{ Вызывается, когда мастер выбран в главном меню. В этой процедуре
  создается, отображается и освобождается главная форма мастера. }
begin
  MainForm := TMainForm.Create(Application);
  try
    MainForm.ShowModal;
  finally
    MainForm.Free;
  end;
end;

function InitWizard(const BorlandIDEServices: IBorlandIDEServices;
  RegisterProc: TWizardRegisterProc;
  var Terminate: TWizardTerminateProc): Boolean stdcall;
var
  Svcs: IOTAServices;
begin
  Result := BorlandIDEServices <> nil;
  if Result then
  begin
    Svcs := BorlandIDEServices as IOTAServices;
    ToolsAPI.BorlandIDEServices := BorlandIDEServices;
    Application.Handle := Svcs.GetParentHandle;
    SDelphiKey := Svcs.GetBaseRegistryKey + '\Experts';
    RegisterProc(TWizardWizard.Create);
  end;
end;

end.
```

Между этим модулем и модулем мастера Dumb имеется несколько различий. Самым важным из них является то, что функция инициализации типа `TWizardInitProc` должна являться входной точкой, предоставляемой в библиотеке DLL для подпрограмм интегрированной среды разработки. В данном случае эта функция называется `InitWizard()`. Она выполняет задачи инициализации мастера, включая следующие.

- Получение через параметр `BorlandIDEServices` указателя на интерфейс `IOTAServices`.
- Сохранение указателя `BorlandIDEServices` для дальнейшего использования.
- Присваивание переменной `Application` библиотеки DLL дескриптора, возвращаемого методом `IOTAServices.GetParentHandle()`. Этот метод возвращает дескриптор окна, которое должно быть родительским по отношению ко всем окнам верхнего уровня, создаваемым мастером.

- Передача созданного экземпляра мастера процедуре `RegisterProc()` для его регистрации в интегрированной среде разработки. Процедура `RegisterProc()` вызывается один раз для каждого экземпляра мастера DLL, регистрируемого в интегрированной среде разработки.
- Дополнительно функция `InitWizard()` может назначать параметру `Terminate` процедуру типа `TWizardTerminateProc`, используемую в качестве процедуры выхода. Эта процедура будет вызвана перед выгрузкой мастера из среды разработки. В ней можно выполнять освобождение необходимых ресурсов. Первоначально этот параметр равен `nil`, и если при выходе из мастера выполнять какие-либо специфические действия не требуется, то это значение изменять не нужно.

В методе инициализации мастера должно использоваться соглашение о вызовах `stdcall`.

В разделе `uses` любого размещаемого в DLL мастера, в котором вызываются функции интерфейса Open Tools API со строковыми параметрами, должен быть указан модуль `ShareMem`. В противном случае при освобождении экземпляра мастера будет сгенерирована ошибка нарушения доступа.

Пользовательский интерфейс мастера

Метод `Execute()` немного сложнее методов, рассматривавшихся ранее в этой главе. В нем создается, а затем освобождается экземпляр мастера — форма `MainForm`, отображаемая в модальном режиме (рис. 26.4). В листинге 26.3 приведен модуль `Main.pas`, содержащий исходный текст формы `MainForm`.

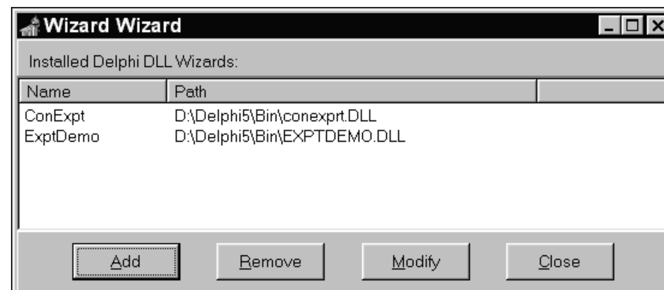


Рис. 26.4. Форма `MainForm` *мастера Wizard*

Листинг 26.3. `Main.pas` — главный модуль мастера Wizard

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, Registry, AddModU, ComCtrls, Menus;
```

```
type
  TMainForm = class(TForm)
    TopPanel: TPanel;
    Label1: TLabel;
    BottomPanel: TPanel;
    WizList: TListView;
    PopupMenu1: TPopupMenu;
    Add1: TMenuItem;
    Remove1: TMenuItem;
    Modify1: TMenuItem;
    AddBtn: TButton;
    RemoveBtn: TButton;
    ModifyBtn: TButton;
    CloseBtn: TButton;
    procedure RemoveBtnClick(Sender: TObject);
    procedure CloseBtnClick(Sender: TObject);
    procedure AddBtnClick(Sender: TObject);
    procedure ModifyBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    procedure DoAddMod(Action: TAddModAction);
    procedure RefreshReg;
  end;

var
  MainForm: TMainForm;

implementation

uses InitWiz;

{$R *.DFM}

var
  DelReg: TRegistry;

procedure TMainForm.RemoveBtnClick(Sender: TObject);
{ Обработчик щелчка на кнопке Remove. Выделенный элемент удаляется
  из системного реестра. }
var
  Item: TListItem;
begin
  Item := WizList.Selected;
  if Item <> nil then
  begin
    if MessageDlg(Format('Remove item "%s"', [Item.Caption]), mtConfirmation,
      [mbYes, mbNo], 0) = mrYes then
      DelReg.DeleteValue(Item.Caption);
    RefreshReg;
  end;
```

```
end;

procedure TMainForm.CloseBtnClick(Sender: TObject);
{ Обработчик щелчка на кнопке Close. Завершение приложения. }
begin
  Close;
end;

procedure TMainForm.DoAddMod(Action: TAddModAction);
{ Добавление в реестр записи для нового мастера
  или модификация уже существующей записи. }
var
  OrigName, ExpName, ExpPath: String;
  Item: TListItem;
begin
  if Action = amaModify then         // Если модификация...
  begin
    Item := WizList.Selected;
    if Item = nil then Exit;         // Проверка, есть ли выделенный элемент
    ExpName := Item.Caption;         // Инициализация переменных
    if Item.SubItems.Count > 0 then
      ExpPath := Item.SubItems[0];
    OrigName := ExpName;             // Сохранение первоначального имени
  end;
  { Активизация диалога, с помощью которого пользователь может
    добавить или отредактировать запись }
  if AddModWiz(Action, ExpName, ExpPath) then
  begin
    { Обработка ситуации, когда выбрано действие Modify
      и изменено имя параметра }
    if (Action = amaModify) and (OrigName <> ExpName) then
      DelReg.RenameValue(OrigName, ExpName);
    DelReg.WriteString(ExpName, ExpPath);  // Запись нового значения
  end;
  RefreshReg;                              // Обновление списка
end;

procedure TMainForm.AddBtnClick(Sender: TObject);
{ Обработчик щелчка на кнопке Add }
begin
  DoAddMod(amaAdd);
end;

procedure TMainForm.ModifyBtnClick(Sender: TObject);
{ Обработчик щелчка на кнопке Modify }
begin
  DoAddMod(amaModify);
end;

procedure TMainForm.RefreshReg;
```

```
{ Обновление списка содержимым системного реестра }
var
  i: integer;
  TempList: TStringList;
  Item: TListItem;
begin
  WizList.Items.Clear;
  TempList := TStringList.Create;
  try
    { Получение из реестра имен мастеров }
    DelReg.GetValueNames(TempList);
    { Получение пути для каждого имени мастера }
    for i := 0 to TempList.Count - 1 do
    begin
      Item := WizList.Items.Add;
      Item.Caption := TempList[i];
      Item.SubItems.Add(DelReg.ReadString(TempList[i]));
    end;
  finally
    TempList.Free;
  end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  RefreshReg;
end;

initialization
  DelReg := TRegistry.Create;          // Создание объекта реестра
  DelReg.RootKey := HKEY_CURRENT_USER; // Установка корневого параметра
  DelReg.OpenKey(SDelphiKey, True);   // Открытие/создание ключа мастера Delphi
finalization
  Delreg.Free;                         // Удаление объекта реестра
end.
```

Этот модуль обеспечивает пользовательский интерфейс помещенного в библиотеку DLL мастера, предназначенного для добавления, удаления и модификации записей в системном реестре. В разделе `initialization` этого модуля создается объект `DelReg`, имеющий тип `TRegistry`. Свойству `RootKey` объекта `DelReg` присваивается значение ключа `HKEY_CURRENT_USER`. Затем с помощью метода `OpenKey()` этот ключ верхнего уровня разворачивается к ключу `\Software\Borland\Delphi\5.0\Experts`, содержащему записи о мастерах Delphi, сохраняемых в библиотеках DLL.

При первом запуске мастера компонент `TListView` под именем `ExptList` заполняется параметрами и значениями из рассмотренного выше ключа системного реестра. Сначала вызывается метод `DelReg.GetValueNames()` с целью получения имен параметров и помещения их в объект `TStringList`. Затем для каждого элемента списка `TStringList` к компоненту `ExptList` добавляется компонент `TListItem`. Для получения значения каждого параметра объекта `TListItem`, помещаемого в список `SubItems`, используется метод `DelReg.ReadString()`.

Работа с реестром выполняется в методах RemoveBtnClick() и DoAddMod(). Метод RemoveBtnClick() отвечает за удаление из системного реестра записи, соответствующей текущему выделенному мастеру. Прежде всего в методе проверяется наличие некоторого выделенного элемента, а затем отображается диалоговое окно для подтверждения выполнения операции удаления. Затем вызывается метод DelReg.DeleteValue(), которому в качестве параметра передается значение CurrentItem.

Методу DoAddMod() может передаваться параметр типа TAddModAction. Этот тип определяется следующим образом:

```
type
  TAddModAction = (amaAdd, amaModify);
```

Как следует из приведенных значений, параметр указывает, будет ли в реестр добавлен новый элемент или модифицирован уже существующий. Сначала функция проверяет наличие выделенного в данный момент элемента. Если его нет, функция дополнительно проверяет, имеет ли переданный ей параметр Action значение amaAdd. Если в параметре Action имеется значение amaModify, то параметр и значение, соответствующие существующему мастеру, присваиваются локальным переменным ExpName и ExpPath. Затем эти значения передаются функции AddModExpert(), определенной в модуле AddModU (листинг 26.4). В этой функции создается диалоговое окно, в котором можно ввести новое либо изменить существующее имя или путь к мастеру (рис. 26.5). Функция AddModExpert() возвращает значение True, если ее диалоговое окно закрыто с помощью щелчка на кнопке OK. В этом случае существующий параметр системного реестра модифицируется с помощью метода DelReg.RenameValue(), а новое или модифицируемое значение записывается с помощью функции DelReg.WriteString().

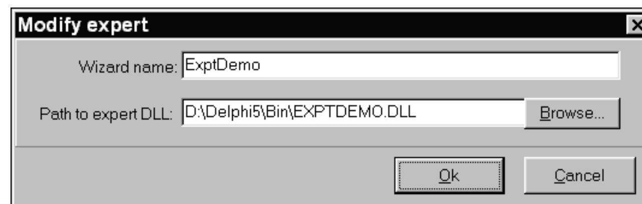


Рис. 26.5. Форма AddModForm *мастера Wizard*

Листинг 26.4. AddModU.pas — модуль с функцией добавления/модификации записи о мастере в системный реестр

```
unit AddModU;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls;

type
  TAddModAction = (amaAdd, amaModify);

  TAddModForm = class(TForm)
    OkBtn: TButton;
```

```
    CancelBtn: TButton;
    OpenDialog: TOpenDialog;
    Panel1: TPanel;
    Label1: TLabel;
    Label2: TLabel;
    PathEd: TEdit;
    NameEd: TEdit;
    BrowseBtn: TButton;
    procedure BrowseBtnClick(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

function AddModWiz(AAction: TAddModAction; var WizName, WizPath: String):
  Boolean;

implementation

{$R *.DFM}

function AddModWiz(AAction: TAddModAction; var WizName, WizPath: String):
Boolean;
{ Создание диалогового окна для добавления или модификации
  параметров системного реестра }
const
  CaptionArray: array[TAddModAction] of string[31] =
    ('Add new expert', 'Modify expert');
begin
  with TAddModForm.Create(Application) do // Создание диалогового окна
  begin
    Caption := CaptionArray[AAction];   // Задание заголовка
    if AAction = amaModify then         // Если модифицировать...
    begin
      NameEd.Text := WizName;         // Имя
      PathEd.Text := WizPath;         // и путь
    end;
    Result := ShowModal = mrOk;         // Отображение окна
    if Result then                      // Если Ok...
    begin
      WizName := NameEd.Text;   // Задание имени
      WizPath := PathEd.Text;   // и пути
    end;
    Free;
  end;
end;

procedure TAddModForm.BrowseBtnClick(Sender: TObject);
begin
```

```
  if OpenDialog.Execute then
    PathEd.Text := OpenDialog.FileName;
end;

end.
```

Одним выстрелом — двух зайцев: EXE и DLL

Как уже упоминалось, один набор модулей с исходным кодом можно применять для создания мастера в виде библиотеки DLL и в виде отдельного выполняемого EXE-файла. Это оказывается возможным, если в файле проекта использовались директивы компилятора. В листинге 26.5 содержится исходный код файла проекта `WizWiz.dpr`, применяемого для создания мастера Wizard.

Листинг 26.5. `WizWiz.dpr` — основной файл проекта создания мастера Wizard

```
{$ifdef BUILD_EXE}
program WizWiz;     // Построение мастера в виде EXE-файла
{$else}
library WizWiz;     // Построение мастера DLL
{$endif}

uses
{$ifndef BUILD_EXE}
  ShareMem,         // Для мастера DLL необходимо использовать модуль ShareMem
  InitWiz in 'InitWiz.pas',
{$endif}
  ToolsAPI,
  Forms,
  Main in 'Main.pas' {MainForm},
  AddModU in 'AddModU.pas' {AddModForm};

{$ifdef BUILD_EXE}
{$R *.RES}                          // Требуется для EXE
{$else}
exports                             // Требуется для DLL
  InitWizard name WizardEntryPoint;  // Требуется точка входа
{$endif}

begin
{$ifdef BUILD_EXE}                  // Требуется для EXE...
  Application.Initialize;
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
{$endif}
end.
```

Как видно из приведенного кода, при построении этого проекта будет создан выполняемый файл, если определено условие компиляции `BUILD_EXE`. В противном случае будет построен мастер DLL. Условие компиляции задается в списке Conditional Defines, расположенном во вкладке Directories/Conditionals диалогового окна Project Options (рис. 26.6).

Относительно рассмотренного проекта можно сделать еще одно заключительное замечание: функция `InitWizard()` из модуля `IntWiz` экспортируется в разделе `exports` файла проекта. Эту функцию необходимо экспортировать под именем `WizardEntryPoint`, определенным в модуле `ToolsAPI`.

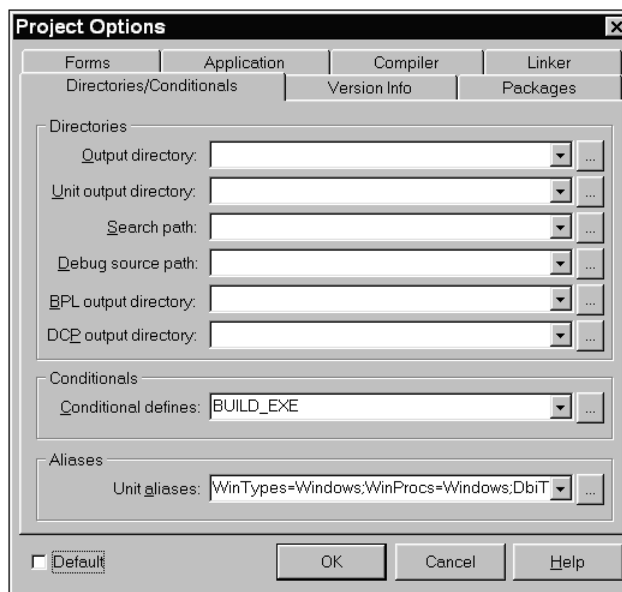


Рис. 26.6. Диалоговое окно Project Options

Компанией Borland файл `ToolsAPI.dcu` не предоставляется. Это означает, что мастера DLL и EXE, содержащие в операторе `uses` ссылки на модуль `ToolsAPI`, могут быть построены лишь с помощью *пакетов*. В настоящее время без пакетов мастер построить нельзя.

Мастер DDG Search

Помните небольшое изящное приложение для поиска, разработка которого была описана в главе 11, "Создание многопоточных приложений" (том I)? В этом разделе вы узнаете, как это полезное приложение преобразовать в еще более полезный мастер Delphi, минимально изменив код. Этот мастер называется DDG Search.

Сначала рассмотрим модуль `InitWiz.pas` (листинг 26.6), который обеспечивает взаимодействие мастера DDG Search с интегрированной средой разработки. Легко заметить, что этот модуль очень похож на модуль из предыдущего примера с тем же именем. И это неслучайно. Модуль `InitWiz.pas` является всего лишь копией модуля из предыдущего примера, в которую внесены некоторые необходимые изменения в имя мастера и метод `Execute()`. Копирование и вставка — это именно то, что можно назвать наследованием в своем изначальном виде. Зачем вводить больше кода, чем требуется?

Листинг 26.6. Модуль `InitWiz.pas`, содержащий логику мастера `DDGSrch`

```
unit InitWiz;

interface

uses
  Windows, ToolsAPI;

type
  TSearchWizard = class(TNotifierObject, IOTAWizard, IOTAMenuWizard)
    // Методы класса IOTAWizard
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    procedure Execute;
    // Метод класса IOTAMenuWizard
    function GetMenuText: string;
  end;

function InitWizard(const BorlandIDEServices: IBorlandIDEServices;
  RegisterProc: TWizardRegisterProc;
  var Terminate: TWizardTerminateProc): Boolean stdcall;

var
  ActionSvc: IOTAActionServices;

implementation

uses SysUtils, Dialogs, Forms, Controls, Main, PriU;

function TSearchWizard.GetName: string;
{ Возвращает имя мастера }

begin
  Result := 'DDG Search';
end;

function TSearchWizard.GetState: TWizardState;
{ Этот мастер в меню всегда доступен }
begin
  Result := [wsEnabled];
end;

function TSearchWizard.GetIDString: String;
{ Возвращает уникальное имя мастера в формате Vendor.Product }
begin
  Result := 'DDG.DDGSearch';
end;
```

```
function TSearchWizard.GetMenuText: string;
{ Возвращает текстовую строку, предназначенную для помещения в меню Help }
begin
  Result := 'DDG Search Expert';
end;

procedure TSearchWizard.Execute;
{ Вызывается, если имя мастера выбрано в меню Help среды разработки. }
{ В этой функции запускается мастер }
begin
  // Если форма не создана, то она создается и отображается
  if MainForm = nil then
  begin
    MainForm := TMainForm.Create(Application);
    ThreadPriWin := TThreadPriWin.Create(Application);
    MainForm.Show;
  end
  else
  // Если форма уже создана, то она восстанавливается и отображается
    with MainForm do
    begin
      if not Visible then Show;
      if WindowState = wsMinimized then WindowState := wsNormal;
      SetFocus;
    end;
end;

function InitWizard(const BorlandIDEServices: IBorlandIDEServices;
  RegisterProc: TWizardRegisterProc;
  var Terminate: TWizardTerminateProc): Boolean stdcall;
var
  Svcs: IOTAServices;
begin
  Result := BorlandIDEServices <> nil;
  if Result then
  begin
    Svcs := BorlandIDEServices as IOTAServices;
    ActionSvc := BorlandIDEServices as IOTAActionServices;
    ToolsAPI.BorlandIDEServices := BorlandIDEServices;
    Application.Handle := Svcs.GetParentHandle;
    RegisterProc(TSearchWizard.Create);
  end;
end;

end.
```

Функция Execute() мастера немного отличается от аналогичной функции, рассматриваемой выше. Главная форма мастера MainForm отображается в немодальном режиме. Конечно, это требует написания дополнительного кода, поскольку заранее нельзя определить, когда форма действительно создана, а когда переменная формы содержит некорректное значение.

Для этого необходимо обеспечить, чтобы переменная MainForm имела значение nil, когда мастер неактивен. Более подробно это будет обсуждаться немного позже.

Еще одним существенным отличием этого проекта от примера, приведенного в главе 11, “Создание многопоточных приложений” (том I), состоит в том, что файл проекта теперь называется DDGSrch.dpr. Он приведен в листинге 26.7.

Листинг 26.7. Файл проекта DDGSrch.dpr для создания мастера DDG Search

```
library DDGSrch;

uses
  ShareMem,
  ToolsAPI,
  Main in 'MAIN.PAS' {MainForm},
  SrchIni in 'SrchIni.pas',
  SrchU in 'SrchU.pas',
  PriU in 'PriU.pas' {ThreadPriWin},
  InitWiz in 'InitWiz.pas',
  MemMap in '..\..\Utils\MemMap.pas',
  StrUtils in '..\..\Utils\StrUtils.pas';

{$R *.RES}

exports
  { Точка входа, используемая средой разработки Delphi }
  InitWizard name WizardEntryPoint;

begin
end.
```

Как видно из кода, этот файл имеет небольшой размер. Обратите внимание на два важных момента. Во-первых, заголовок library показывает, что будет создан мастер DLL. А, во-вторых, для инициализации мастера интегрированной средой разработки Delphi экспортируется функция InitExpert().

В модуль Main этого проекта также было внесено несколько изменений. Как упоминалось ранее, если мастер неактивен, то переменная MainForm должна содержать значение nil. Как уже указывалось в главе 2, “Язык программирования Object Pascal” (том I), при запуске приложения переменная MainForm автоматически принимает значение nil. Кроме того, в обработчике события OnClick экземпляр формы уничтожается, а глобальная переменная MainForm принимает значение nil. Этот метод имеет следующий вид:

```
procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
  Application.OnShowHint := FOldShowHint;
  MainForm := nil;
end;
```

И, наконец, дважды щелкнув в списке главной формы, вы сможете с помощью рассматриваемого мастера переносить файлы в редактор кода среды разработки. Эта возможность реализована в методе FileLBDblClick() следующим образом:

```
procedure TMainForm.FileLBDblClick(Sender: TObject);
{ Вызывается, если пользователь дважды щелкнул в списке. Файл загружается в
интегрированную среду разработки. }
var
  FileName: string;
  Len: Integer;
begin
  FileName := FileLB.Items[FileLB.ItemIndex];
  {Нужно удостовериться, что пользователь щелкнул на файле... }
  if (FileName <> '') and (Pos('File ', FileName) = 1) then
  begin
    { Вырезание из строки фрагментов "File " и ":". }
    FileName := Copy(FileName, 6, Length(FileName));
    Len := Length(FileName);
    if FileName[Len] = ':' then SetLength(FileName, Len - 1);
    { Открытие проекта или файла. }
    if CompareText(ExtractFileExt(FileName), '.DPR') = 0 then
      ActionSvc.OpenProject(FileName, True)
    else
      ActionSvc.OpenFile(FileName);
    { Получение интерфейса модуля. }
  end;
end;
```

В этом методе для открытия определенного файла используются методы `OpenFile()` и `OpenProject()` класса `IOTAActionServices`.

В листинге 26.8 приведен полный код модуля `Main` проекта `DDGSrch`, а на рис. 26.7 показан мастер DDG Search, работающий в интегрированной среде.

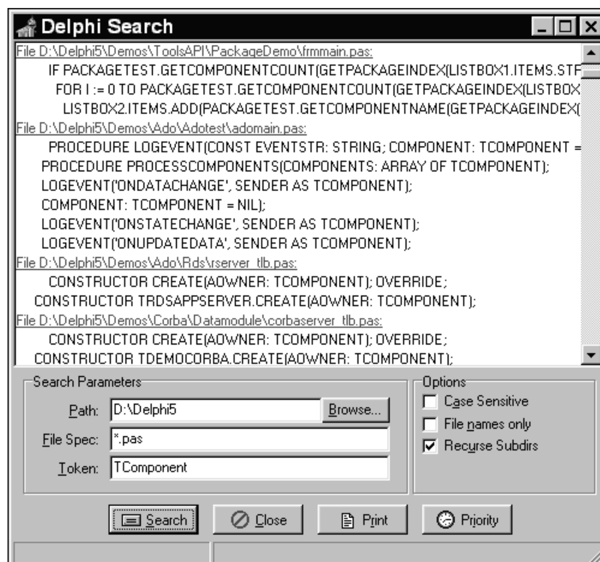


Рис. 26.7. Мастер DDG Search в действии

Листинг 26.8. `Main.pas` **— главный модуль проекта** `DDGSrch`

```
unit Main;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons, ExtCtrls, Menus, SrchIni,
  SrchU, ComCtrls, InitWiz;

type
  TMainForm = class(TForm)
    FileLB: TListBox;
    PopupMenu1: TPopupMenu;
    Font1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    FontDialog1: TFontDialog;
    StatusBar: TStatusBar;
    AlignPanel: TPanel;
    ControlPanel: TPanel;
    ParamsGB: TGroupBox;
    LFileSpec: TLabel;
    LToken: TLabel;
    lPathName: TLabel;
    EFileSpec: TEdit;
    EToken: TEdit;
    PathButton: TButton;
    OptionsGB: TGroupBox;
    cbCaseSensitive: TCheckBox;
    cbFileNamesOnly: TCheckBox;
    cbRecurse: TCheckBox;
    SearchButton: TBitBtn;
    CloseButton: TBitBtn;
    PrintButton: TBitBtn;
    PriorityButton: TBitBtn;
    View1: TMenuItem;
    EPathName: TEdit;
    procedure SearchButtonClick(Sender: TObject);
    procedure PathButtonClick(Sender: TObject);
    procedure FileLBDrawItem(Control: TWinControl; Index: Integer;
      Rect: TRect; State: TOwnerDrawState);
    procedure Font1Click(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure PrintButtonClick(Sender: TObject);
    procedure CloseButtonClick(Sender: TObject);
    procedure FileLBDblClick(Sender: TObject);
    procedure FormResize(Sender: TObject);
```

```
    procedure PriorityButtonClick(Sender: TObject);
    procedure ETokenChange(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    FOldShowHint: TShowHintEvent;
    procedure ReadIni;
    procedure WriteIni;
    procedure DoShowHint(var HintStr: string; var CanShow: Boolean;
      var HintInfo: THintInfo);
    procedure WMGetMinMaxInfo(var M: TWMGetMinMaxInfo);
      message WM_GETMINMAXINFO;
  public
    Running: Boolean;
    SearchPri: integer;
    SearchThread: TSearchThread;
    procedure EnableSearchControls(Enable: Boolean);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses Printers, ShellAPI, MemMap, FileCtrl, PriU;

procedure PrintStrings(Strings: TStrings);
{ Эта процедура печатает все строки из параметра Strings }
var
  Prn: TextFile;
  i: word;
begin
  if Strings.Count = 0 then // Существуют ли строки?
  begin
    MessageDlg('No text to print!', mtInformation, [mbOk], 0);
    Exit;
  end;
  AssignPrn(Prn);          // Связывание с принтером переменной Prn
  try
    Rewrite(Prn);         // Открытие принтера
    try
      for i := 0 to Strings.Count - 1 do  // Повторить для всех строк
        writeln(Prn, Strings.Strings[i]); // Запись в принтер
    finally
      CloseFile(Prn);                     // Закрытие принтера
    end;
  except
    on EInOutError do
      MessageDlg('Error Printing text.', mtError, [mbOk], 0);
```

```
  end;
end;

procedure TMainForm.WMGetMinMaxInfo(var M: TWMGetMinMaxInfo);
begin
  inherited;
  // Предотвращение чрезмерного уменьшения размеров формы
  with M.MinMaxInfo^ do
  begin
    ptMinTrackSize.x := OptionsGB.Left + OptionsGB.Width - ParamsGB.Left + 10;
    ptMinTrackSize.y := 200;
  end;
end;

procedure TMainForm.EnableSearchControls(Enable: Boolean);
{ Активизация или деактивизация определенных элементов
  управления, поскольку параметры не могут быть
  модифицированы во время выполнения поиска. }
begin
  // Разрешение/запрещение соответствующих элементов управления
  SearchButton.Enabled := Enable;
  cbRecurse.Enabled := Enable;
  cbFileNamesOnly.Enabled := Enable;
  cbCaseSensitive.Enabled := Enable;
  PathButton.Enabled := Enable;
  EPathName.Enabled := Enable;
  EFileSpec.Enabled := Enable;
  EToken.Enabled := Enable;
  Running := not Enable         // Установка флага Running
  ETokenChange(nil);
  with CloseButton do
  begin
    if Enable then
    begin                // Установка свойств кнопки Close/Stop
      Caption := '&Close';
      Hint := 'Close Application';
    end
    else begin
      Caption := '&Stop';
      Hint := 'Stop Searching';
    end;
  end;
end;

procedure TMainForm.SearchButtonClick(Sender: TObject);
{Вызывается по щелчку на кнопке Search. Создание потока поиска.}
begin
  EnableSearchControls(False);          // Блокирование элементов управления
  FileLB.Clear;                         // Очистка списка
  { запуск потока }
```

```
  SearchThread := TSearchThread.Create(cbCaseSensitive.Checked,
    cbFileNamesOnly.Checked, cbRecurse.Checked, EToken.Text,
    EPathName.Text, EFileSpec.Text);
end;

procedure TMainForm.ETokenChange(Sender: TObject);
begin
  SearchButton.Enabled := not Running and (EToken.Text <> '');
end;

procedure TMainForm.PathButtonClick(Sender: TObject);
{ Вызывается по щелчку на кнопке Path. Позволяет выбрать новый путь. }
var
  ShowDir: string;
begin
  ShowDir := EPathName.Text;
  if SelectDirectory(ShowDir, [], 0) then
    EPathName.Text := ShowDir;
end;

procedure TMainForm.FileLBDrawItem(Control: TWinControl;
  Index: Integer; Rect: TRect; State: TOwnerDrawState);
{ Вызывается для отображения списка. }
var
  CurStr: string;
begin
  with FileLB do
  begin
    CurStr := Items.Strings[Index];
    Canvas.FillRect(Rect);                // Очистка прямоугольника
    if not cbFileNamesOnly.Checked then   // Если не только имя файла...
      { если текущая строка - имя файла... }
      if (Pos('File ', CurStr) = 1) and
        (CurStr[Length(CurStr)] = ':') then
      begin
        Canvas.Font.Style := [fsUnderline]; // Шрифт с подчеркиванием
        Canvas.Font.Color := clRed;         // Установка красного цвета
      end
    else
      Rect.Left := Rect.Left + 15;          // Иначе отступ
    DrawText(Canvas.Handle, PChar(CurStr), Length(CurStr), Rect, dt_SingleLine);
  end;
end;

procedure TMainForm.Font1Click(Sender: TObject);

begin
  { Выбор нового шрифта }
  if FontDialog1.Execute then
    FileLB.Font := FontDialog1.Font;
```

```
end;

procedure TMainForm.FormDestroy(Sender: TObject);
{ Обработчик события OnDestroy формы }
begin
  WriteIni;
end;

procedure TMainForm.FormCreate(Sender: TObject);
{ Обработчик события OnCreate формы }
begin
  Application.HintPause := 0;       // Подсказки отображаются без задержки
  FOldShowHint := Application.OnShowHint; // Установка подсказок
  Application.OnShowHint := DoShowHint;
  ReadIni;                               // Чтение INI-файла
end;

procedure TMainForm.DoShowHint(var HintStr: string; var CanShow: Boolean;
 var HintInfo: THintInfo);
{ Обработчик события OnHint приложения }
begin
  { Отображение подсказок в строке состояния }
  StatusBar.Panels[0].Text := HintStr;
  { Не отображать всплывающие подсказки, если курсор расположен
    над пользовательским элементом управления }
  if (HintInfo.HintControl <> nil) and
    (HintInfo.HintControl.Parent <> nil) and
    ((HintInfo.HintControl.Parent = ParamsGB) or
    (HintInfo.HintControl.Parent = OptionsGB) or
    (HintInfo.HintControl.Parent = ControlPanel)) then
    CanShow := False;
  FOldShowHint(HintStr, CanSHow, HintInfo);
end;

procedure TMainForm.PrintButtonClick(Sender: TObject);
{ Вызывается по щелчку на кнопке Print. }
begin
  if MessageDlg('Send search results to printer?', mtConfirmation,
    [mbYes, mbNo], 0) = mrYes then
    PrintStrings(FileLB.Items);
end;

procedure TMainForm.CloseButtonClick(Sender: TObject);
{ Вызывается для остановки потока или для завершения приложения }
begin
  // Если поток запущен, то он завершается
  if Running then SearchThread.Terminate
  // В противном случае завершение приложения
  else Close;
end;
```

```
procedure TMainForm.FormResize(Sender: TObject);
{ Обработчик события OnResize. Центрирование элементов управления в форме. }
begin
 { Разделение строки состояния на две панели в отношении 1/3 - 2/3 }
  with StatusBar do
  begin
    Panels[0].Width := Width div 3;
    Panels[1].Width := Width * 2 div 3;
  end;
  { Центрирование элементов управления в середине формы }
  ControlPanel.Left := (AlignPanel.Width div 2) - (ControlPanel.Width div 2);
end;

procedure TMainForm.PriorityButtonClick(Sender: TObject);
{ Отображение формы приоритета потока }
begin
  ThreadPriWin.Show;
end;

procedure TMainForm.ReadIni;
{ Считывание из системного реестра значений по умолчанию }
begin
  with SrchIniFile do
  begin
    EPathName.Text := ReadString('Defaults', 'LastPath', 'C:\');
    EFileSpec.Text := ReadString('Defaults', 'LastFileSpec', '*.*');
    EToken.Text := ReadString('Defaults', 'LastToken', '');
    cbFileNamesOnly.Checked := ReadBool('Defaults', 'FNamesOnly', False);
    cbCaseSensitive.Checked := ReadBool('Defaults', 'CaseSens', False);
    cbRecurse.Checked := ReadBool('Defaults', 'Recurse', False);
    Left := ReadInteger('Position', 'Left', 100);
    Top := ReadInteger('Position', 'Top', 50);
    Width := ReadInteger('Position', 'Width', 510);
    Height := ReadInteger('Position', 'Height', 370);
  end;
end;

procedure TMainForm.WriteIni;
{ Запись текущих значений обратно в системный реестр }
begin
  with SrchIniFile do
  begin
    WriteString('Defaults', 'LastPath', EPathName.Text);
    WriteString('Defaults', 'LastFileSpec', EFileSpec.Text);
    WriteString('Defaults', 'LastToken', EToken.Text);
    WriteBool('Defaults', 'CaseSens', cbCaseSensitive.Checked);
    WriteBool('Defaults', 'FNamesOnly', cbFileNamesOnly.Checked);
    WriteBool('Defaults', 'Recurse', cbRecurse.Checked);
    WriteInteger('Position', 'Left', Left);
    WriteInteger('Position', 'Top', Top);
```

```
    WriteInteger('Position', 'Width', Width);
    WriteInteger('Position', 'Height', Height);
  end;
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
  Application.OnShowHint := FOldShowHint;
  MainForm := nil;
end;

end.
```

Мастера форм

Интерфейсом Open Tools API поддерживается еще один тип мастера — мастер форм. После установки мастер этого типа можно выбрать в диалоговом окне New Items. С их помощью можно генерировать новые формы и модули. В главе 24, “Расширение оболочки Windows”, этот тип мастера использовался для генерации новых форм AppBar. Однако тогда не был показан текст, который, собственно, и “оживлял” мастера.

Мастер форм создать очень просто, хотя при этом и требуется реализовать множество интерфейсных методов. Процесс создания мастера форм можно разделить на пять основных этапов.

1. Создайте класс, производный от класса TCustomForm, TDataModule или любого класса, производного от класса TWinControl, — он будет использоваться в качестве базового класса формы. Обычно этот класс находится в отдельном модуле. В данном случае в качестве базового будет использоваться класс TAppBar.
2. Создайте потомок класса TNotifierObject, реализующий следующие интерфейсы: IOTAWizard, IOTARepositoryWizard, IOTAFormWizard, IOTACreator и IOTAModuleCreator.
3. Обычно для получения нового имени модуля и класса мастера в методе IOTAWizard.Execute() необходимо вызвать метод IOTAModuleServices.GetNewModuleAndClassName(). Для того чтобы сообщить среде разработки о необходимости создания нового модуля вызовите метод IOTAModuleServices.CreateModule().
4. Реализация большинства методов вышеупомянутых интерфейсов занимает лишь одну строку. К менее тривиальным относятся методы NewFormFile() и NewImplFile() интерфейса IOTAModuleCreator, возвращающие код для формы и модуля соответственно. Более сложным может оказаться также метод IOTACreator.GetOwner(), однако следующий пример является хорошей иллюстрацией того, как модуль можно добавить в текущий проект.
5. Завершите процедуру Register() мастера регистрацией обработчика класса новой формы, что осуществляется с помощью процедуры RegisterCustomModule() модуля DsgnIntf. Создайте мастер, вызвав процедуру RegisterPackageWizard() модуля ToolsAPI.

В листинге 26.9 приведен исходный код модуля ABWizard.pas, в котором реализован мастер AppBar.

Листинг 26.9. Модуль ABWizard.pas, содержащий реализацию мастера AppBar

```
unit ABWizard;

interface

uses Windows, Classes, ToolsAPI;

type
  TAppBarWizard = class(TNotifierObject, IOTAWizard, IOTARepositoryWizard,
    IOTAFormWizard, IOTACreator, IOTAModuleCreator)
  private
    FUnitIdent: string;
    FClassName: string;
    FFileName: string;
  protected
    // Методы класса IOTAWizard
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    procedure Execute;
    // Методы классов IOTARepositoryWizard / IOTAFormWizard
    function GetAuthor: string;
    function GetComment: string;
    function GetPage: string;
    function GetGlyph: HICON;
    // Методы класса IOTACreator
    function GetCreatorType: string;
    function GetExisting: Boolean;
    function GetFileSystem: string;
    function GetOwner: IOTAModule;
    function GetUnnamed: Boolean;
    // Методы класса IOTAModuleCreator
    function GetAncestorName: string;
    function GetImplFileName: string;
    function GetIntfFileName: string;
    function GetFormName: string;
    function GetMainForm: Boolean;
    function GetShowForm: Boolean;
    function GetShowSource: Boolean;
    function NewFormFile(const FormIdent, AncestorIdent: string): IOTAFile;
    function NewImplSource(const ModuleIdent, FormIdent,
      AncestorIdent: string): IOTAFile;
    function NewIntfSource(const ModuleIdent, FormIdent,
      AncestorIdent: string): IOTAFile;
    procedure FormCreated(const FormEditor: IOTAFormEditor);
  end;
```

```
implementation

uses Forms, AppBars, SysUtils, DsgnIntf;

{$R CodeGen.res}

type
  TBaseFile = class(TInterfacedObject)
  private
    FModuleName: string;
    FFormName: string;
    FAncestorName: string;
  public
    constructor Create(const ModuleName, FormName, AncestorName: string);
  end;

  TUnitFile = class(TBaseFile, IOTAFile)
  protected
    function GetSource: string;
    function GetAge: TDateTime;
  end;

  TFormFile = class(TBaseFile, IOTAFile)
  protected
    function GetSource: string;
    function GetAge: TDateTime;
  end;

{ TBaseFile }

constructor TBaseFile.Create(const ModuleName, FormName,
  AncestorName: string);
begin
  inherited Create;
  FModuleName := ModuleName;
  FFormName := FormName;
  FAncestorName := AncestorName;
end;

{ TUnitFile }

function TUnitFile.GetSource: string;
var
  Text: string;
  ResInstance: THandle;
  HRes: HRSRC;
begin
  ResInstance := FindResourceHInstance(HInstance);
  HRes := FindResource(ResInstance, 'CODEGEN', RT_RCDATA);
  Text := PChar(LockResource(LoadResource(ResInstance, HRes)));
```

```
  SetLength(Text, SizeOfResource(ResInstance, HRes));
  Result := Format(Text, [FModuleName, FFormName, FAncestorName]);
end;

function TUnitFile.GetAge: TDateTime;
begin
  Result := -1;
end;

{ TFormFile }

function TFormFile.GetSource: string;
const
  FormText = 'object %0:s: T%0:s'#13#10'end';
begin
  Result := Format(FormText, [FFormName]);
end;

function TFormFile.GetAge: TDateTime;
begin
  Result := -1;
end;

{ TAppBarWizard }

{ TAppBarWizard.IOTAWizard }

function TAppBarWizard.GetIDString: string;
begin
  Result := 'DDG.AppBarWizard';
end;

function TAppBarWizard.GetName: string;
begin
  Result := 'DDG AppBar Wizard';
end;

function TAppBarWizard.GetState: TWizardState;
begin
  Result := [wsEnabled];
end;

procedure TAppBarWizard.Execute;
begin
  (BorlandIDEServices as IOTAModuleServices).GetNewModuleAndClassName(
    'AppBar', FUnitIdent, FClassName, FFileName);
  (BorlandIDEServices as IOTAModuleServices).CreateModule(Self);
end;

{ TAppBarWizard.IOTARepositoryWizard / TAppBarWizard.IOTAFormWizard }
```

```
function TAppBarWizard.GetGlyph: HICON;
begin
  Result := 0;  // Использование стандартной пиктограммы
end;

function TAppBarWizard.GetPage: string;
begin
  Result := 'DDG';
end;

function TAppBarWizard.GetAuthor: string;
begin
  Result := 'Delphi 5 Developer''s Guide';
end;

function TAppBarWizard.GetComment: string;
begin
  Result := 'Creates a new AppBar form.'
end;

{ TAppBarWizard.IOTACreator }

function TAppBarWizard.GetCreatorType: string;
begin
  Result := '';
end;

function TAppBarWizard.GetExisting: Boolean;
begin
  Result := False;
end;

function TAppBarWizard.GetFileSystem: string;
begin
  Result := '';
end;

function TAppBarWizard.GetOwner: IOTAModule;
var
  I: Integer;
  ModServ: IOTAModuleServices;
  Module: IOTAModule;
  ProjGrp: IOTAProjectGroup;
begin
  Result := nil;
  ModServ := BorlandIDEServices as IOTAModuleServices;
  for I := 0 to ModServ.ModuleCount - 1 do
  begin
    Module := ModSErv.Modules[I];
    // Поиск текущей группы проекта
```

```
    if CompareText(ExtractFileExt(Module.FileName), '.bpg') = 0 then
      if Module.QueryInterface(IOTAProjectGroup, ProjGrp) = S_OK then
      begin
        // Возврат активного проекта группы
        Result := ProjGrp.GetActiveProject;
        Exit;
      end;
  end;
end;

function TAppBarWizard.GetUnnamed: Boolean;
begin
  Result := True;
end;

{ TAppBarWizard.IOTAModuleCreator }

function TAppBarWizard.GetAncestorName: string;
begin
  Result := 'TAppBar';
end;

function TAppBarWizard.GetImplFileName: string;
var
  CurrDir: array[0..MAX_PATH] of char;
begin
  // Примечание: требуется полный путь!
  GetCurrentDirectory(SizeOf(CurrDir), CurrDir);
  Result := Format('%s\%s.pas', [CurrDir, FUnitIdent, '.pas']);
end;

function TAppBarWizard.GetIntfFileName: string;
begin
  Result := '';
end;

function TAppBarWizard.GetFormName: string;
begin
  Result := FClassName;
end;

function TAppBarWizard.GetMainForm: Boolean;
begin
  Result := False;
end;

function TAppBarWizard.GetShowForm: Boolean;
begin
  Result := True;
end;
```

```
function TAppBarWizard.GetShowSource: Boolean;
begin
  Result := True;
end;

function TAppBarWizard.NewFormFile(const FormIdent,
  AncestorIdent: string): IOTAFile;
begin
  Result := TFormFile.Create('', FormIdent, AncestorIdent);
end;

function TAppBarWizard.NewImplSource(const ModuleIdent, FormIdent,
  AncestorIdent: string): IOTAFile;
begin
  Result := TUnitFile.Create(ModuleIdent, FormIdent, AncestorIdent);
end;

function TAppBarWizard.NewIntfSource(const ModuleIdent, FormIdent,
  AncestorIdent: string): IOTAFile;
begin
  Result := nil;
end;

procedure TAppBarWizard.FormCreated(const FormEditor: IOTAFormEditor);
begin
  // Ничего не выполняется
end;

end.
```

В этом модуле для генерации исходного кода используется интересный прием: неформатированный исходный код сохраняется в `.res`-файле, который связывается с помощью директивы `$R`. Это очень гибкий способ хранения исходного кода мастера, позволяющий легко его модифицировать. Файл ресурсов (`.res`-файл) создан путем включения текстового файла и ресурсов `RCDATA` в `.RC`-файл, который затем был скомпилирован с помощью компилятора `BRCC32`. В листингах 26.10 и 26.11 приведено содержимое файлов `CodeGen.txt` и `CodeGen.rc`.

Листинг 26.10. Файл `CodeGen.txt` — шаблон ресурсов мастера `AppBar`

```
unit %0:s;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, AppBars;

type
  T%1:s = class(%2:s)
  private
```

```
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  %1:s: T%1:s;

implementation

{$R *.DFM}

end.
```

Листинг 26.11. Файл `CODEGEN.RC`

```
CODEGEN RCDATA CODEGEN.TXT
```

Регистрация пользовательского модуля и мастера производится внутри процедуры `Register()` в содержащем модуль пакете разработки с помощью следующих двух строк:

```
  RegisterCustomModule(TAppBar, TCustomModule);
  RegisterPackageWizard(TAppBarWizard.Create);
```

Резюме

Прочитав данную главу, вы должны гораздо лучше понимать принципы работы различных модулей и интерфейсов, предоставляемых интерфейсом Open Tools API в Delphi. В частности, здесь рассказано, как создавать мастера, встраиваемые в интегрированную среду разработки. Следующая глава, “Разработка приложений CORBA в Delphi”, завершает эту часть книги. Она посвящена технологии CORBA и ее реализации в Delphi.

Разработка приложений CORBA в Delphi

Глава

27

Брокеры запросов объектов 542

Интерфейсы 542

Заглушки и каркасы 543

ORB-брокер VisiBroker 544

Поддержка архитектуры CORBA в среде Delphi 546

Создание CORBA-решений в среде Delphi 5 561

Развертывание ORB-брокера VisiBroker 598

Резюме 599

Аббревиатура CORBA означает Common Object Request Broker Architecture (Универсальная архитектура посредничества при запросе объектов). CORBA — это спецификация, разработанная группой поддержки объектной технологии (Object Management Group, или OMG), которая с помощью разрабатываемых ею стандартов определяет архитектуру решений, принимаемых в качестве основы при реализации объектов, независимо от используемого языка программирования и вычислительной платформы. Группа OMG — это независимый консорциум компаний и отраслевых экспертов, цель которых — разработка стандартов открытых и независящих от платформы архитектур распределенных объектов. В отличие от некоторых конкурирующих стандартов (например, таких как технологии COM/DCOM компании Microsoft), группа OMG не предлагает никаких реализаций определяемых ею стандартов.

Брокеры запросов объектов

Ядром архитектуры CORBA является *брокер запросов объектов* (Object Request Broker — ORB). ORB-брокер обеспечивает реализацию спецификации CORBA и является своего рода связующим звеном (на самом деле это программные средства промежуточного слоя), которое соединяет воедино все составные части, предназначенные для решения некоторой проблемы. Если вы уже знакомы с технологией COM/DCOM компании Microsoft, то наверняка согласитесь, что работу ORB-брокера можно сравнить с работой библиотек COM/DCOM, обеспечивающих функционирование уровней времени выполнения, безопасности и передачи данных. Все взаимодействие между клиентом и сервером осуществляется через ORB-брокер, причем таким образом, чтобы вызовы методов и их параметры можно было разрешить в адресном пространстве вызывающей или вызываемой процедуры (маршалинг). Брокер ORB также предоставляет много вспомогательных процедур, которые могут быть вызваны прямо из клиента или сервера, аналогично тому, как это делается в библиотеке `oleaut32.dll` для технологии COM/DCOM. Как упоминалось выше, спецификация CORBA не предоставляет никакой стандартной реализации библиотеки ORB. Поскольку построение ORB-брокера не является тривиальной задачей, разработчики CORBA зависят от сторонних производителей в части поддержки CORBA-совместимых реализаций ORB. Следует считать положительным тот факт, что на данный момент реализацией ORB успешно занимаются многие производители, причем не забыта ни одна из ведущих платформ (например, такая как Windows или UNIX), а также некоторые не столь известные операционные системы. В настоящее время двумя наиболее популярными реализациями спецификации CORBA являются ORB-брокер VisiBroker компании Inprise и ORB-брокер Orbix фирмы IONA.

Интерфейсы

Одно решение CORBA может включать разнообразные объекты, разработанные в среде различных языков программирования и выполняющиеся на различных платформах. Следовательно, нужно иметь некоторый стандартный способ представления одних объектов другим объектам, а также клиентам и ORB-брокеру. Такое представление выполняется с помощью интерфейса. Интерфейс определяет список доступных методов и их параметров, но не предназначен для реализации этих процедур. Когда объект CORBA реализует некоторый интерфейс, он должен гарантировать реализацию всех определенных этим интерфейсом методов. На самом низком уровне интерфейс представляет собой просто таблицу функций или список точек входа в конкретные методы. Поскольку такая конструкция может быть представлена на любой аппаратной платформе и посредством любых инструментов разработки, интерфейсы являются универсальным языком мира CORBA. Но так как синтаксис разных языков разра-

ботки может сильно различаться друг от друга, группа OMG создала так называемый язык определения интерфейсов (Interface Definition Language, или IDL). Язык IDL — это стандартный язык, предназначенный для определения CORBA-интерфейсов. Многие инструменты разработки способны транслировать синтаксис языка IDL в их родной синтаксис, чтобы позволить разработчикам легко конструировать CORBA-совместимые интерфейсы. Что касается Delphi, то здесь не нужно вручную писать IDL-код, поскольку редактор библиотек типов позволяет визуально определять создаваемые интерфейсы и при необходимости автоматически экспортировать соответствующий код IDL.

Заглушки и каркасы

Механизм архитектуры CORBA работает на основе передачи полномочий. Использование полномочий в настоящее время является ведущей моделью разработки для решения сложных проблем, связанных с передачей данных между распределенными объектами. Уполномоченный "*представитель*" (proxy) находится как на стороне клиента, так и на стороне сервера и создает у каждой из сторон впечатление общения с локальным процессом. Затем ORB-брокер обрабатывает все разнородные детали, имеющие отношение к связям между представителями (например, маршалинг, сетевые взаимодействия и пр.). Эта архитектура, как показано на рис. 27.1, позволяет разработчикам CORBA-клиента или CORBA-сервера в определенной степени абстрагироваться от необходимости вникать в детали передачи данных нижнего уровня и сосредоточиться на оптимальной реализации конкретного решения своих бизнес-задач. В терминах архитектуры CORBA уполномоченный представитель сервера, с которым связывается клиент, называется *заглушкой* (*stub*), а представитель клиента на стороне сервера — *каркасом* (*skeleton*). При создании объекта CORBA-сервера с помощью мастера Delphi модуль, содержащий определения интерфейса для заглушки и каркаса, будет сгенерирован автоматически.

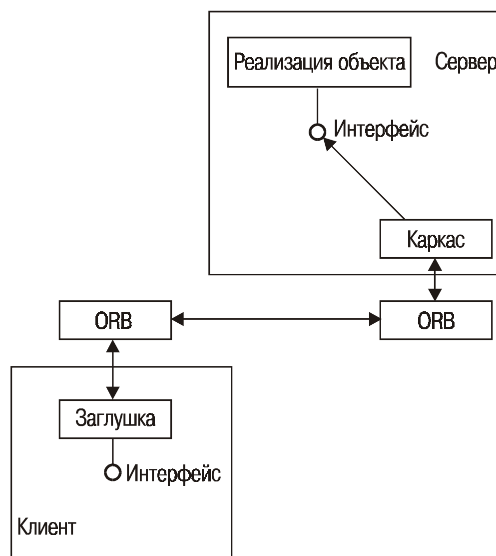


Рис. 27.1. Упрощенная схема архитектуры CORBA

ORB-брокер VisiBroker

Как упоминалось выше, CORBA — это стандарт, который предусматривает наличие некоторого стороннего производителя, осуществляющего практическую реализацию ORB-сервиса. Поддержка архитектуры CORBA, предлагаемая в Delphi 4 и 5, реализуется с помощью ORB-брокера "VisiBroker" фирмы Inprise. Продукт VisiBroker обеспечивает полную поддержку спецификации CORBA, а также предоставляет многочисленные дополнительные VisiBroker-расширения, такие как служба имен или служба генерации событий. Поскольку полное освещение продукта VisiBroker выходит за рамки этой главы, мы остановимся на тех особенностях этого продукта, которые имеют только непосредственное отношение к реализации архитектуры CORBA в среде Delphi. Более подробную информацию по ORB-брокеру VisiBroker, включая документацию на продукт, можно найти по адресу: `www.borland.com/visibroker`.

VisiBroker: службы поддержки времени выполнения

В ORB-брокер VisiBroker включены библиотеки, которые представляют собой различные службы поддержки времени выполнения. Их задача — обеспечить совместную работу отдельных частей распределенной архитектуры CORBA/VisiBroker. Каждая из служб будет рассмотрена в последующих разделах.

Служба Smart Agent (инструмент `osagent`)

Одна из составных частей продукта VisiBroker — служба Smart Agent — предоставляет для CORBA-приложений услуги, связанные с определением местоположения объектов. Использование службы Smart Agent придает CORBA-среде дислокационную прозрачность. То есть, сами клиенты не интересуются расположением серверов: клиентам достаточно "уметь" определить местоположение агента Smart Agent, и он сам справится со всеми деталями поиска соответствующего сервера. Служба Smart Agent должна функционировать на одном из узлов данной локальной сети. В одной сети можно использовать несколько агентов Smart Agent, каждый из которых настраивается для прослушивания различных портов, что, по сути, обеспечивает функционирование в сети нескольких независимых `ORB`-доменов. Это может оказаться весьма полезным с точки зрения как производительности ORB-среды, так и расширения возможностей ORB-среды разработки. Служба Smart Agent может быть также настроена для взаимодействия с другими агентами Smart Agent, расположенными в других локальных сетях, что позволит расширить рамки создаваемой CORBA-инфраструктуры.

Демон активизации объектов

Еще одна служба, входящая в состав продукта VisiBroker, — демон активизации объектов (Object Activation Daemon, или OAD) — предоставляет услуги, связанные с динамическим запуском серверов, когда в этом появляется необходимость. Служба Smart Agent может лишь связывать клиентов с уже работающими реализациями объектов. Однако, если реализация объекта CORBA зарегистрирована с помощью демона OAD, то служба Smart Agent и демон OAD смогут сотрудничать с целью запуска серверных процессов, которые в данный момент являются недоступными.

Хранилище интерфейсов

Хранилище интерфейсов (Interface Repository, или IREP) — это интерактивная база данных с информацией о типах объектов. Без этого хранилища не обойтись клиентам, которым требуется динамически связываться (режим позднего связывания) с CORBA-интерфейсами.

ORB-брокер может использовать помещенную в хранилище интерфейсов информацию о типах для выполнения корректного маршалинга вызовов с поздним связыванием. Чтобы поддерживать динамическое связывание, хранилище интерфейсов должно функционировать на одном из узлов сети, доступном для всех клиентов, и в этом хранилище должен быть зарегистрирован каждый используемый интерфейс.

VisiBroker: инструменты администрирования

Чтобы настраивать и управлять вышеупомянутыми средствами поддержки времени выполнения, в Delphi пакет VisiBroker поставляется с набором утилит администрирования, либо имеющих графический интерфейс пользователя (GUI), либо вызываемых из командной строки. Все эти утилиты перечислены в табл. 27.1, а подробности их применения будут описаны ниже в этой главе.

Таблица 27.1. Инструменты администрирования пакета VisiBroker

Инструмент	Назначение
`osagent`	Используется для управления службой Smart Agent
`osfind`	Перечисляет реализации объектов, доступные в сети
`oad`	Используется для управления демоном активизации объектов (OAD)
`oadutil`	Применяется для регистрации, отмены регистрации и создания списка интерфейсов с помощью демона OAD
`irep`	Используется для управления хранилищем интерфейсов
`idl2ir`	Утилита для регистрации языка определения интерфейсов IDL с использованием хранилища интерфейсов
`vregedit`	Позволяет внесение изменений в стандартные параметры агента Smart Agent в записях системного реестра Windows
`vbver`	Сообщает номера версий служб, входящих в состав продукта VisiBroker

Поддержка архитектуры CORBA в среде Delphi

Введенная в Delphi (начиная с версии 4) поддержка архитектуры CORBA часто подвергалась незаслуженной критике. Несмотря на наличие определенных ограничений, многие слухи оказались сильно преувеличенными или попросту неверными. Начнем с того, что поддержка в Delphi действительно является "настоящей" реализацией архитектуры CORBA. ORB-брокер VisiBroker для языка C++ (в виде библиотеки `orb_br.dll`) функционирует на нижнем уровне. В качестве оболочки для него используется библиотека `orbpas50.dll`, что позволяет определениям интерфейсов и типов данных в среде Pascal или Delphi взаимодействовать с ORB-брокером VisiBroker.

Однако существует одна деталь, которая у "борцов за чистоту" спецификации CORBA вызывает неописуемый ужас при первом же взгляде на сгенерированный средствами Delphi код заглушки и каркаса. Подобное категорическое неприятие вызывают ссылки на уникальные идентификаторы интерфейсов GUID и интерфейсы `IUnknown` и `IDispatch`. Эти конструкции являются базовыми элементами архитектуры COM/DCOM, и большинство программи-

стов, занимающихся поддержкой архитектуры CORBA, стремятся избежать их применения в своих любимых CORBA-реализациях. Вокруг существования этих COM-"пугал" распространяется множество слухов, включая разговоры о том, что CORBA-вызовы осуществляются с помощью механизмов COM или что маршалинг параметров выполняется дважды (один раз — по принципам COM, а другой — по принципам CORBA). Но чем безосновательно терять голову от всех этих нелепых предположений, лучше разобраться, почему эти COM-определения существуют в CORBA-сервере, сгенерированном в среде Delphi.

- Во-первых, первоначально добавление интерфейсов к Delphi происходило с ориентацией на технологию COM. Все интерфейсы Delphi являются производными от базового COM-интерфейса (`IUnknown`). Это значит, что при определении в Delphi любого интерфейса, в том числе и предназначенного для использования в архитектуре CORBA, должны быть реализованы три дополнительных метода, определенных в интерфейсе `IUnknown` (`QueryInterface`, `AddRef` и `Release`). Это справедливо в отношении любого CORBA-интерфейса; в том числе и для интерфейсов базового класса `TCorbaImplementation`, обеспечивающего реализацию этих трех методов для Delphi-разработчиков.
- Во-вторых, при создании объекта CORBA с помощью мастера Delphi нетрудно заметить, что по умолчанию создается двойной COM-интерфейс. Изучив сгенерированный Delphi модуль заглушки и каркаса, можно увидеть, что CORBA-интерфейсы являются производными от интерфейса `IDispatch` и, следовательно, могут классифицироваться как диспинтерфейсы. И хотя для CORBA-реализации такое решение не является обязательным (в частности, можно изменить созданные определения, взяв в качестве базового интерфейс `IUnknown`), в реализации объекта должны быть дополнительно определены методы интерфейса `IDispatch`, иначе эти объекты не будут откомпилированы должным образом. В объявлениях классов `TCorbaDispatchStub` и `TCorbaImplementation` реализованы четыре дополнительных метода, определенных в интерфейсе `IDispatch`. Внимательно изучив исходный текст, можно заметить, что на самом деле эти реализации ничего не выполняют; они *присутствуют* лишь для того, чтобы при работе с CORBA-объектами можно было использовать редактор библиотеки типов.
- Наконец, интерфейсы, которые генерируются мастером Delphi, содержат уникальные идентификаторы GUID (или идентификаторы интерфейсов IID), которые в COM-технологии используются для уникальной идентификации интерфейсов. И хотя в самой архитектуре CORBA идентификаторы GUID для идентификации объектов или интерфейсов не используются, некоторые внутренние процедуры библиотеки VCL обращаются к ним с целью уникальной идентификации CORBA-интерфейсов. Поэтому идентификаторы GUID не следует удалять из интерфейсов, сгенерированных мастером CORBA Object Wizard.

Из вышесказанного следует, что COM-конструкции, генерируемые мастером объектов CORBA в Delphi, не так страшны, как представлялось некоторым разработчикам. Но зато сколько от них пользы (именно в этом проявляется уникальность Delphi)! Оказывается, можно с минимальными усилиями создавать классы, которые одновременно могут работать как по технологии COM/DCOM, так и в среде с архитектурой CORBA!

На момент написания этой книги самым бросающимся в глаза недостатком реализации в Delphi архитектуры CORBA было отсутствие утилиты для преобразования языка IDL в язык Pascal (`Idl2Pas`). Подобный инструмент в данный момент разработан компанией Inprise для языков Java и C++. Обычным недоразумением является то, что в Delphi не предусмотрена возможность раннего связывания с CORBA-сервером, написанным на другом языке. Правильнее было бы сказать, что разработчик Delphi не может с минимальными усилиями реали-

зовать раннее связывание с CORBA-сервером, написанным на другом языке. Delphi-клиенты могут выполнять статическое (раннее) или динамическое (позднее) связывание с CORBA-серверами, написанными на языке Delphi или на любом другом языке. Однако неспособность Delphi импортировать IDL-файл и генерировать Pascal-код, который был бы в состоянии понимать компилятор, очень усложняет процесс раннего связывания с CORBA-серверами, написанными на других языках. Поэтому разработчик должен вручную писать программный код классов CORBA-заглушек при проектировании раннего связывания Delphi-клиента с CORBA-объектом, реализованным на языках C++ или Java. В компании Inprise сейчас ведутся работы над созданием преобразователя `Idl2Pas`, который должен значительно упростить разработку приложений Delphi/CORBA, и, возможно, скоро он станет доступен как дополнение (надстройка) для Delphi 5. Общий обзор этой новой технологии приводится ниже в данной главе.

Классы поддержки архитектуры CORBA

В качестве структуры поддержки архитектуры CORBA в Delphi используется некоторая цепочка наследования интерфейсов и их реализаций, позволяющая разработчикам создавать CORBA-клиенты и CORBA-серверы. Разработка приложений по технологии CORBA осуществляется в основном посредством реализации интерфейсов для объектов, заглушек и каркасов. Поскольку интерфейсы не поддерживают концепцию наследования кода реализации, решение этой задачи могло бы оказаться довольно утомительным, поскольку для всех интерфейсов понадобилась бы повторная реализация обращений к ORB-брокеру CORBA. Для исключения этой необходимости Delphi предоставляет группу базовых VCL-классов, реализующих методы основных CORBA-интерфейсов (например, таких как `ICorbaObject`, `ISkeletonObject` и `IStubObject`). Основные базовые классы показаны на рис. 27.2 и описаны в приведенном ниже списке.

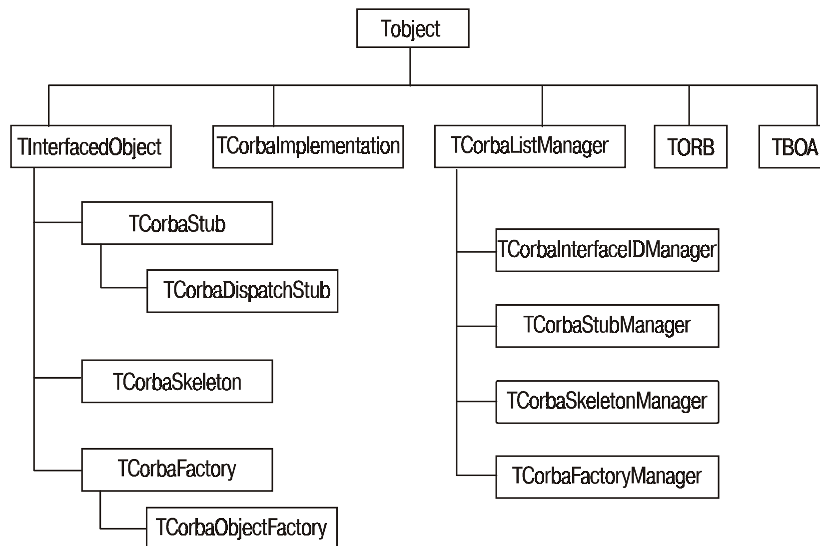


Рис. 27.2. Иерархическая структура поддержки архитектуры CORBA средствами библиотеки VCL

- **Класс TCorbaImplementation**. Этот класс поддерживает интерфейс IUnknown и обеспечивает средства запроса интерфейса и подсчета ссылок. В данном классе также *присутствуют* методы поддержки дуальных интерфейсов интерфейса IDispatch, необходимые для поддержки работы с редактором библиотеки типов Delphi. CORBA-объекты в Delphi-исполнении являются производными именно от этого класса.
- **Класс TCorbaStub**. В этом классе реализованы интерфейсы ICorbaObject и IStubObject. Класс TCorbaStub — это базовый класс для всех заглушек, сгенерированных редактором библиотеки типов Delphi. Заглушка используется для маршалинга вызовов интерфейсов из CORBA-клиента. При создании классов, производных от класса TCorbaStub, потребуется предоставлять собственные средства маршалинга.
- **Класс TCorbaDispatchStub**. Этот класс является производным от класса TCorbaStub и реализует (чтобы обеспечить присутствие) методы COM-интерфейса IDispatch. Это именно те интерфейсы, которые создаются для поддержки редактора библиотеки типов Delphi, являющегося потомком интерфейса IDispatch, благодаря чему становится возможным его взаимодействие с элементами CORBA-архитектуры.
- **Класс TCorbaSkeleton**. Этот класс реализует интерфейс ISkeletonObject и отвечает за подключение к ORB-брокеру и передачу вызовов на объект сервера. В отличие от заглушки, класс каркаса на самом деле не реализует интерфейс сервера. Вместо этого каркас содержит ссылку на сервер и вызывает методы по этой ссылке.
- **Классы TCorbaFactory и TCorbaObjectFactory**. Класс TCorbaFactory — это базовый класс для объектов, способных создавать экземпляры CORBA-объектов. Класс TCorbaObjectFactory способен создавать экземпляры любых потомков класса TCorbaImplementation.
- **Класс TCorbaListManager (и его подклассы)**. Структура поддержки архитектуры CORBA в Delphi во время выполнения должна отслеживать поведение таких абсолютно разных образований, как каркасы, заглушки, фабрики классов и идентификаторы интерфейсов ID. Класс TCorbaListManager — это базовый класс, который обеспечивает поддержку синхронизации потоков. Это позволяет библиотеке VCL организовать внутренний механизм поддержки безопасности потоков. Обычно разработчику не приходится отдавать много сил классам управления списками, за исключением тех редких случаев, когда нужна регистрация пользовательского объекта заглушки.
- **Класс TBOA**. Это класс Delphi, который реализует адаптер базисных объектов (Basic Object Adapter, или BOA), представляющих в технологии CORBA механизм связи между ORB-брокером и каркасом. Класс TBOA является специфическим объектом, который всегда используется в единственном числе и не нуждается в непосредственном создании экземпляров.
- **Класс TORB**. Класс TORB позволяет подпрограммам библиотеки VCL Delphi взаимодействовать с ORB-брокером VisiBroker. Подобно классу TBOA, класс TORB также является объектом, который всегда используется в единственном числе и не нуждается в непосредственном создании экземпляров. В реализациях многочисленных методов класса TORB используются обращения к функциям библиотеки orbpas50.dll, которая, в свою очередь, вызывает процедуры, определенные в C++-варианте ORB-брокера VisiBroker (библиотеке orb_br.dll).

Мастер CORBA-объектов

Перечисленные выше классы относительно просты и представляют практически весь набор CORBA-классов библиотеки VCL, с которыми разработчику приходится иметь дело в среде Delphi. Но, возможно, вам будет интересно узнать, что в Delphi существует мастер, который поможет вам должным образом реализовать CORBA-объекты. Для вызова мастера выберите команду File⇨New. При этом откроется окно хранилища объектов Delphi. Находясь в этом окне, перейдите во вкладку Multitier, показанную на рис. 27.3.

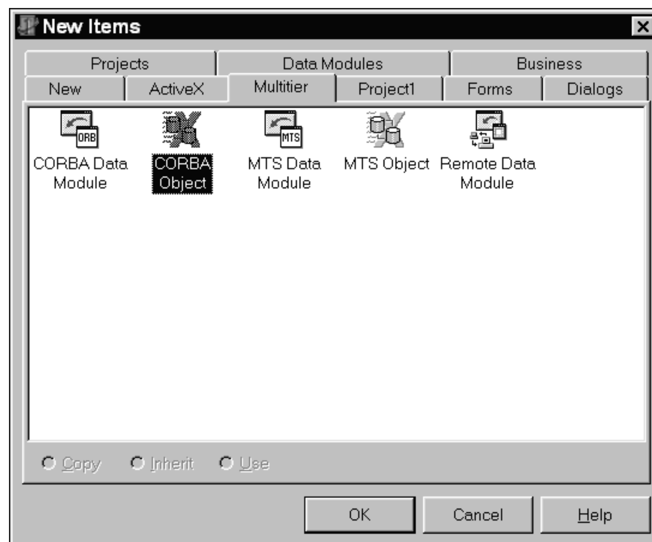


Рис. 27.3. Выбор мастера CORBA-объектов в окне хранилища объектов Delphi

Теперь щелкните на пиктограмме CORBA Object — на экран будет выведено окно мастера CORBA Object Wizard, показанное на рис. 27.4.

Рис. 27.4. Окно мастера CORBA Object Wizard

Введите в текстовое поле Class Name имя, выбранное для создаваемого CORBA-объекта и интерфейса. Обратите внимание на то, что при этом не следует действовать в соответствии со стандартным соглашением Delphi, рекомендующим начинать имена классов с буквы *T*, поскольку эта буква будет автоматически добавлена мастером. Например, если ввести имя *MyObject*, будет сгенерирован класс Delphi под именем `TMyObject` с реализованным в нем интерфейсом `IMyObject`.

С помощью раскрывающегося списка Instancing необходимо определить, каким образом экземпляры объектов будут предоставляться клиентам. При этом можно выбрать один из двух вариантов.

- *Shared Instance* (Разделяемый экземпляр). Именно эта модель обычно применяется в большинстве CORBA-разработок. Каждый клиент использует один разделяемый экземпляр реализации этого объекта. CORBA-серверы, использующие эту модель, должны быть построены как серверы, не запоминающие своего состояния. Поскольку один и тот же экземпляр объекта могут совместно использовать несколько клиентов, невозможно гарантировать, что любой отдельно взятый клиент найдет сервер точно в таком же состоянии, в котором он находился после последнего вызова из этого клиента.
- *Instance-per-client* (Каждому клиенту — по экземпляру). Эта модель предполагает создание уникального экземпляра объекта для каждого клиента, который запрашивает связь с объектом. Эта модель позволяет использовать объекты, способные сохранять свое состояние между последовательными вызовами клиентов. Однако данная модель связана с б*о*льшим уровнем потребления системных ресурсов (по сравнению с предыдущим вариантом), поскольку в этом случае необходимо, чтобы серверы отслеживали состояние соединений с клиентами и освобождали объекты после завершения клиентами работы с этими ресурсами.

С помощью раскрывающегося списка Threading Model (Модель потоков) можно указать, как будут вызываться создаваемые CORBA-объекты. Здесь также можно выбирать один из двух вариантов.

- *Single-threaded* (Однопоточная модель). Каждый экземпляр объекта будет вызываться из единственного потока; следовательно, сам объект не нуждается в специальных мерах по обеспечению безопасности потока. Отметим, что приложение CORBA-сервера может содержать несколько объектов или их экземпляров, поэтому доступ к глобальным или совместно используемым данным должен быть организован с соблюдением мер безопасности при доступе из отдельных потоков.
- *Multithreaded* (Многопоточная модель). Несмотря на то что каждое соединение клиента будет создавать вызовы в потоке, выделенном для данного клиента, объекты могут получать совпадающие по времени вызовы от нескольких потоков. В этом случае должны быть предусмотрены меры по обеспечению безопасности работы потоков как с глобальными данными, так и с данными экземпляров объектов. Самый трудный для реализации сценарий (с точки зрения работы с потоками) возникает при разделении экземпляров объектов в сочетании с многопоточной моделью их использования. Простейший же сценарий получается при выборе однопоточной модели с созданием уникального экземпляра объекта для каждого клиента.

Следует иметь в виду, что простым выбором потоковой модели еще не обеспечивается безопасность работы с потоками в реализациях создаваемых серверов или объектов. Варианты установки опции Threading Model служат просто для указания потоковой модели, которую поддерживает создаваемый объект. Вся ответственность за реализацию CORBA-серверов в смысле безопасности работы с потоками ложится на плечи разработчика, выбравшего ту или иную потоковую модель.

После успешного завершения работы мастера создания CORBA-объектов будут сгенерированы два модуля на языке Pascal. При этом имя модуля заглушки/каркаса соответствует шаблону `Проект_TLB.pas`. Этот файл будет содержать определения главного интерфейса создаваемого объекта, классов заглушки и каркаса, класса фабрики CORBA-классов и код, предназначенный для регистрации классов заглушки, каркаса и интерфейса с помощью соответствующих механизмов Delphi. В листинге 27.1 представлен код, сгенерированный для класса под именем "`MyFirstCORBAServer`".

Листинг 27.1. Модуль заглушки и каркаса, сгенерированный мастером Delphi

```
unit FirstCorbaServer_TLB;

// ************************************************************************
// Внимание
// --------
// Типы, объявленные в этом файле, были сгенерированы на основании
// данных, считанных из библиотеки типов. Если эта библиотека типов
// явно или косвенно (через другую библиотеку типов, ссылающуюся на данную)
// будет реимпортирована либо во время редактирования библиотеки типов
// будет выбрана команда редактора библиотеки типов 'Refresh' (Обновить),
// содержимое этого файла будет сгенерировано заново и все сделанные
// вручную модификации будут утеряны.
// ************************************************************************

// PASTLWTR : $Revision:   1.88  $
// Файл сгенерирован 11/02/1999 4:01:10 PM из библиотеки типов, описанной ниже.

// ************************************************************************
// Type Lib: C:\ICON99\FirstCORBAServer\FirstCorbaServer.tlb (1)
// IID\LCID: {CE8DB340-913A-11D3-9706-0000861F6726}\0
// Helpfile:
// DepndLst:
//   (1) v2.0 stdole, (C:\WINDOWS\SYSTEM\STDOLE2.TLB)
//   (2) v4.0 StdVCL, (C:\WINDOWS\SYSTEM\STDVCL40.DLL)
// ************************************************************************
{$TYPEDADDRESS OFF} // Модуль должен быть скомпилирован без
                    // выполнения контроля типов указателей
interface

uses Windows, ActiveX, Classes, Graphics, OleServer, OleCtrls, StdVCL, 
SysUtils, CORBAObj, OrbPas, CorbaStd;

// ************************************************************************
// Уникальные идентификаторы GUID объявлены в библиотеке типов.
//. Здесь используются следующие префиксы:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// ************************************************************************
const
  // Главная и второстепенная версии библиотеки типов
  FirstCorbaServerMajorVersion = 1;
  FirstCorbaServerMinorVersion = 0;

  LIBID_FirstCorbaServer: TGUID = '{CE8DB340-913A-11D3-9706-0000861F6726}';

  IID_IMyFirstCorbaServer: TGUID = '{CE8DB341-913A-11D3-9706-0000861F6726}';
```

```
  CLASS_MyFirstCorbaServer: TGUID = '{CE8DB343-913A-11D3-9706-0000861F6726}';
type
// *********************************************************************
// Объявление типов, определенных в библиотеке типов
// *********************************************************************
  IMyFirstCorbaServer = interface;
  IMyFirstCorbaServerDisp = dispinterface;

// *********************************************************************
// Объявление компонентных классов (CoClasses), определенных в библиотеке типов
// (ПРИМЕЧАНИЕ. Здесь мы устанавливаем соответствие каждого
// компонентного класса его стандартному интерфейсу)
// *********************************************************************
  MyFirstCorbaServer = IMyFirstCorbaServer;

// *********************************************************************
// Interface: IMyFirstCorbaServer
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {CE8DB341-913A-11D3-9706-0000861F6726}
// *********************************************************************
  IMyFirstCorbaServer = interface(IDispatch)
    ['{CE8DB341-913A-11D3-9706-0000861F6726}']
    procedure SayHelloWorld; safecall;
  end;

// *********************************************************************
// DispIntf:  IMyFirstCorbaServerDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {CE8DB341-913A-11D3-9706-0000861F6726}
// *********************************************************************
  IMyFirstCorbaServerDisp = dispinterface
    ['{CE8DB341-913A-11D3-9706-0000861F6726}']
    procedure SayHelloWorld; dispid 1;
  end;

  TMyFirstCorbaServerStub = class(TCorbaDispatchStub, IMyFirstCorbaServer)
  public
    procedure SayHelloWorld; safecall;
  end;

  TMyFirstCorbaServerSkeleton = class(TCorbaSkeleton)
  private
    FIntf: IMyFirstCorbaServer;
  public
    constructor Create(const InstanceName: string; const Impl: IUnknown);
      override;
    procedure GetImplementation(out Impl: IUnknown); override; stdcall;
  published
    procedure SayHelloWorld(const InBuf: IMarshalInBuffer; Cookie: Pointer);
  end;
```

```
// *********************************************************************
// В классе CoMyFirstCorbaServer предусмотрены методы Create и CreateRemote
// для создания экземпляров стандартного интерфейса IMyFirstCorbaServer,
// предоставляемого компонентным классом MyFirstCorbaServer. Эти функции
// предназначены для использования клиентами, желающими автоматизировать
// CoClass-объекты, предоставляемые сервером данной библиотеки типов.
// *********************************************************************
  CoMyFirstCorbaServer = class
    class function Create: IMyFirstCorbaServer;
    class function CreateRemote(const MachineName: string):
      BIMyFirstCorbaServer;
  end;

  TMyFirstCorbaServerCorbaFactory = class
    class function CreateInstance(const InstanceName: string):
      IMyFirstCorbaServer;
  end;

implementation

uses ComObj;

{ TMyFirstCorbaServerStub }

procedure TMyFirstCorbaServerStub.SayHelloWorld;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('SayHelloWorld', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
end;

{ TMyFirstCorbaServerSkeleton }

constructor TMyFirstCorbaServerSkeleton.Create(const InstanceName: string;
  const Impl: IUnknown);
begin
  inherited;
  inherited InitSkeleton('MyFirstCorbaServer', InstanceName,
    'IDL:FirstCorbaServer/IMyFirstCorbaServer:1.0', tmMultiThreaded, True);
  FIntf := Impl as IMyFirstCorbaServer;
end;

procedure TMyFirstCorbaServerSkeleton.GetImplementation(out Impl: IUnknown);
begin
  Impl := FIntf;
end;
procedure TMyFirstCorbaServerSkeleton.SayHelloWorld(
  const InBuf: IMarshalInBuffer; Cookie: Pointer);
```

```
var
  OutBuf: IMarshalOutBuffer;
begin
  FIntf.SayHelloWorld;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
end;

class function CoMyFirstCorbaServer.Create: IMyFirstCorbaServer;
begin
  Result := CreateComObject(CLASS_MyFirstCorbaServer) as IMyFirstCorbaServer;
end;

class function CoMyFirstCorbaServer.CreateRemote(const MachineName: string):
  IMyFirstCorbaServer;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_MyFirstCorbaServer) as
    IMyFirstCorbaServer;
end;

class function TMyFirstCorbaServerCorbaFactory.CreateInstance(
  const InstanceName: string): IMyFirstCorbaServer;
begin
  Result := CorbaFactoryCreateStub(
    'IDL:FirstCorbaServer/MyFirstCorbaServerFactory:1.0', 'MyFirstCorbaServer',
    InstanceName, '', IMyFirstCorbaServer) as IMyFirstCorbaServer;
end;

initialization
  CorbaStubManager.RegisterStub(IMyFirstCorbaServer, TMyFirstCorbaServerStub);
  CorbaInterfaceIDManager.RegisterInterface(IMyFirstCorbaServer,
    'IDL:FirstCorbaServer/IMyFirstCorbaServer:1.0');
  CorbaSkeletonManager.RegisterSkeleton(IMyFirstCorbaServer,
    TMyFirstCorbaServerSkeleton);

end.
```

Внимательно изучив содержимое модуля заглушки и каркаса, можно заметить то, что класс каркаса на самом деле не реализует интерфейс IMyFirstCorbaServer. Каркас будет иметь те же самые методы, что и поддерживаемый им интерфейс, но при этом нельзя не увидеть, что параметры у них различны. Методам каркаса будет передаваться необработанная, подвергнутая маршалингу информация, которую они должны подвергнуть демаршалингу и передать в качестве параметров соответствующему интерфейсу. По этой причине каркас не реализует интерфейс IMyFirstCorbaServer напрямую. Вместо этого он содержит внутреннюю ссылку на поддерживаемый интерфейс и передает вызовы к нему по этой внутренней ссылке.

Второй модуль, сгенерированный мастером, будет содержать исходную структуру для реализации создаваемого объекта. В нем генерируется класс Delphi, который является потомком класса TCorbaImplementation и реализует главный интерфейс объекта. В этом модуле также создается экземпляр фабрики класса, которая отвечает за создание данного CORBA-объекта. Модуль реализации типичного CORBA-объекта представлен в листинге 27.2.

Листинг 27.2. Реализация CORBA-объекта, сгенерированного мастером Delphi

```
unit uMyFirstCorbaServer;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, ComObj, StdVcl,
  CorbaObj, FirstCorbaServer_TLB;

type

  TMyFirstCorbaServer = class(TCorbaImplementation, IMyFirstCorbaServer)
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  protected
    procedure SayHelloWorld; safecall;
  end;

implementation

uses CorbInit;

procedure TMyFirstCorbaServer.SayHelloWorld;
begin
  //Здесь реализуем метод.
end;

initialization
  TCorbaObjectFactory.Create('MyFirstCorbaServerFactory', 'MyFirstCorbaServer',
    'IDL:FirstCorbaServer/MyFirstCorbaServerFactory:1.0', IMyFirstCorbaServer,
    TMyFirstCorbaServer, iMultiInstance, tmSingleThread);
end.
```

В конечном итоге этот модуль будет содержать код реализации всех методов интерфейса IMyFirstCORBAServer, а также всех внутренних функций, определяющих внутреннее поведение класса TMyFirstCORBAServer. Благодаря механизмам классического наследования от класса TCorbaImplementation, эта реализация способна автоматически превратиться в CORBA-объект. Поддерживая интерфейс IMyFirstCorbaServer, данный объект гарантирует, что он удовлетворяет контракту работы с этим интерфейсом. Но вместо того чтобы вручную объявлять интерфейс и выполнять реализацию этого объекта, обратимся за помощью к редактору библиотек типов визуальной среды разработки Delphi.

Редактор библиотек типов IDE Delphi

Чтобы полностью реализовать этот пользовательский CORBA-объект, как в модуль заглушки и скелета, так и в модуль реализации объекта необходимо добавить некоторый код. Поначалу эта задача может показаться очень трудной, но, к счастью, у нас есть помощник в

лице редактора библиотек типов. Выберите в главном меню Delphi команду View⇨Type Library. Откроется диалоговое окно, показанное на рис. 27.5, которое визуально представляет интерфейсы и другие конструкции, определенные в модуле заглушки и каркаса.

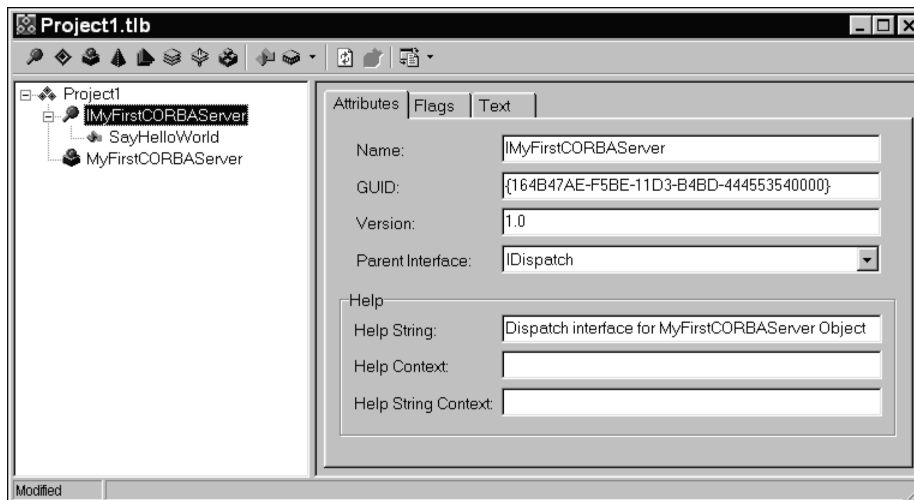


Рис. 27.5. Редактор библиотек типов визуальной среды разработки Delphi

На данном этапе можно выделить в редакторе интерфейс `IMyFirstCorbaServer` и щелкнуть на кнопке панели инструментов New Method, что позволит добавить новый метод. Добавив метод, можно использовать визуальный интерфейс редактора для определения параметров, типов возвращаемых значений и пр. Обратите внимание на то, что не все типы данных, показанные в качестве возможных типов параметров в окне редактора библиотек типов, допустимы для CORBA-объектов. Поскольку в данный момент редактор библиотек типов является инструментом двойного назначения (для COM- и CORBA-объектов), многие типы данных допустимы только для объектов COM/Automation (автоматизации). За исчерпывающими списками допустимых для спецификации CORBA (языка IDL) типов данных лучше всего обратиться к справочным файлам Delphi. Добавив с помощью редактора библиотек типов методы интерфейса, щелкните на кнопке Refresh Implementation, чтобы регенерировать код в проекте. При этом будет обновлен модуль заглушки и каркаса, а в модуль реализации будут добавлены пустые методы реализации. Теперь вам останется лишь ввести нужный код и реализовать пустые методы, сгенерированные редактором библиотеки типов.

На заметку

В состав Delphi 5 включено новое средство, которое способно генерировать оболочки компонентов для компонентных классов (CoClasses), содержащихся в библиотеке типов. К сожалению, эти оболочки генерируются в любом случае: при импортировании существующей библиотеки типов и при создании собственной библиотеки. Данные оболочки компонентов не подходят для CORBA-объектов, поэтому для предотвращения генерации ненужного кода необходимо выполнить следующие действия. В меню Delphi выберите команду Project⇨Import Type Library. В открывшемся диалоговом окне Import Type Library сбросьте флажок Generate Component Wrapper (Генерировать компонентную оболочку) и затем закройте диалоговое окно, щелкнув на кнопке Close, расположенной в верхнем правом углу. Наконец, щелкните на кнопке Refresh Implementation, расположенной на панели инструментов в окне редактора Type Library Editor. Этого достаточно для того, чтобы ненужный код был удален из создаваемого приложения.

Создание CORBA-решений в среде Delphi 5

Рассмотрев базовую структуру приложения архитектуры CORBA и имеющие к ней отношение инструментальные средства IDE Delphi, попробуем применить полученные знания к созданию CORBA-сервера. Затем займемся построением клиента, который будет использовать наш собственный CORBA-сервер.

Построение CORBA-сервера

Изучив основы создания CORBA-сервера, самое время вникнуть в некоторые детали и построить CORBA-сервер от начала и до конца. Наша задача — создать CORBA-объект промежуточного слоя, который может принимать SQL-запросы от клиента, опрашивать базу данных и отсылать результаты клиенту, пославшему этот вызов. В нашей реализации для упрощения получения данных от сервера базы данных мы используем систему доступа к базам данных фирмы Borland (Borland Database Engine, или BDE). При этом следует иметь в виду, что связь с BDE существует только с точки зрения объекта сервера, тогда как приложение клиента не обязано “знать” о существовании механизма BDE. С таким же успехом сервер можно было бы настроить на получение данных, используя другой механизм, — например, реализованный в Delphi 5 новый набор данных ADO или даже пользовательский тип, описываемый объектом класса `TDataset`.

Вызов мастера CORBA-объектов

Создайте новое приложение Delphi, а затем, как описано выше, вызовите мастер CORBA-объектов (CORBA Object Wizard). Присвойте объекту имя `QueryServer`, и мастер создаст интерфейс с именем `IQueryServer` и класс реализации с именем `TQueryServer`. Выберите в раскрывающемся списке Instancing вариант Instance-Per-Client, поскольку наш объект должен поддерживать перемещения в наборе данных (например, команды First, Next и пр.) и, следовательно, помнить о своем текущем состоянии. Чтобы избежать на данном этапе сложностей, связанных с написанием кода, гарантирующего безопасное взаимодействие потоков, выберите в раскрывающемся списке Threading Model вариант Single-Threaded. По щелчку на кнопке OK в проект будут добавлены модуль заглушки/каркаса и модуль реализации объекта.

Нетрудно заметить, что стандартное приложение Delphi содержит форму, добавленную по умолчанию. Чтобы оставаться в главном цикле сообщений Windows, созданное в Delphi GUI-приложение обязательно должно иметь форму. Но большинство приложений CORBA-серверов не нуждается в визуальной форме, поэтому эту небольшую проблему можно решить путем ввода в файл проекта следующей программной строки:

```
Application.ShowMainForm := False;
```

Но в данном случае хотелось бы иметь уверенность в работоспособности сервера, поэтому оставим пока форму видимой и добавим метку (компонент `TLabel`) для уведомления о том, что наш CORBA-сервер активен. Полученная форма показана на рис. 27.6.

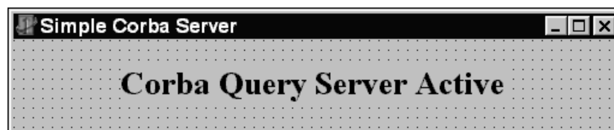


Рис. 27.6. Главная форма создаваемого CORBA-сервера

Нужно отметить, что эту форму следует рассматривать как глобальные данные. Несмотря на то что мы создали CORBA-объект с однопотоковой моделью, приложение CORBA-сервера может содержать и другие объекты, обслуживающие вызовы в других потоках. Следовательно, прямой доступ к этой форме из кода объекта нельзя расценивать безопасным с точки зрения работы с потоками.

Использование редактора библиотек типов

После генерации кода, предназначенного для реализации CORBA-объекта, воспользуемся услугами редактора библиотек типов для добавления в создаваемый интерфейс методов поддержки. В данном примере предполагается добавить в интерфейс `IQueryServer` функции, позволяющие клиентам входить в базу данных и посылать SQL-операторы, переходить при желании от одних данных к другим и считывать одновременно по одной записи результирующего набора. Все это выполняется с помощью выделения интерфейса `IQueryServer` и щелчка на кнопке New Method панели инструментов. При добавлении каждого нового метода присваиваемое ему имя указывается в текстовом поле Name вкладки Attributes. Кроме того, для каждого нового метода может понадобиться воспользоваться сеткой, расположенной во вкладке Parameters окна Type Library Editor. Этот инструмент позволяет избежать ошибок при указании типов параметров и возвращаемых значений. После добавления нескольких методов, обеспечивающих желаемую функциональность, окно редактора библиотек типов примет вид, показанный на рис. 27.7.

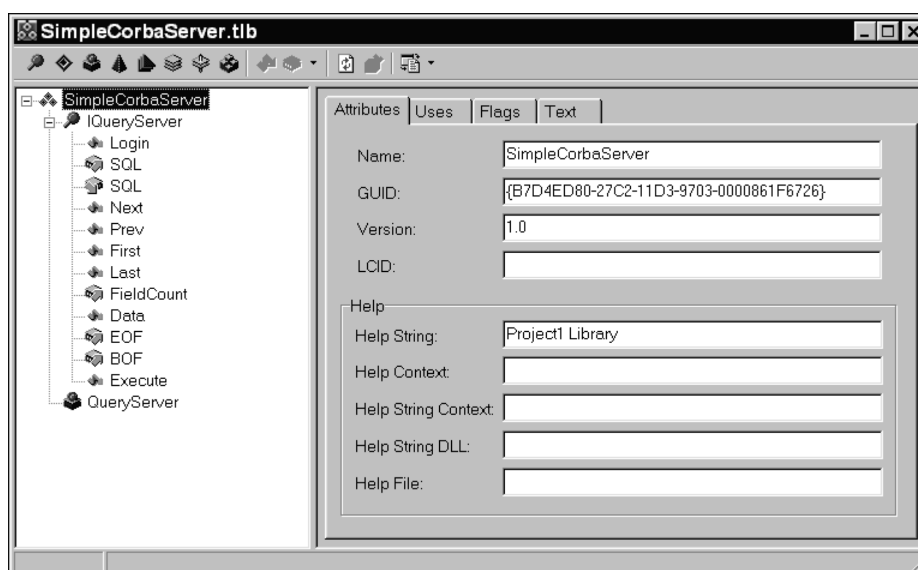


Рис. 27.7. Методы интерфейса `IQueryServer` в окне редактора библиотек типов

Реализация методов интерфейса `IQueryServer`

После определения интерфейса нового CORBA-объекта осталось реализовать код, который заставил бы добавленные методы работать. Создаваемый класс реализации будет инкапсулировать компоненты `TDatabase` и `TQuery`, что обеспечит доступ к системе BDE и данным сервера. Поэтому остальная часть работы не содержит никаких неожиданностей: методы интерфейса будут просто вызывать функции, использующие возможности компонентов `TDatabase` и `TQuery` библиотеки VCL.

Единственным методом, который может потребовать к себе чуть большего внимания в процессе реализации, является метод Data (функция). Этот метод будет считывать всю строку данных, которая в данный момент выбрана в результатах выполнения запроса. Поскольку предполагается обеспечить возврат нескольких значений, нам потребуется использовать для этого структуру определенного типа, способную соответствующим образом представить возвращаемые значения. В языке IDL в подобных случаях обычно используется *последовательность*, которая представляет собой массив переменной длины некоторого типа данных. В редакторе библиотек типов пока не предусмотрена возможность определения IDL-последовательностей, поэтому в качестве типа возвращаемого значения метода Data можно использовать тип OLEVariant. Тип OLEVariant в действительности будет представлен массивом, содержащим значения отдельных полей для текущей строки данных. Тип OLEVariant вполне пригоден для этой задачи, поскольку язык IDL имеет подобную конструкцию, именуемую Any, которая может хранить любой допустимый в языке IDL тип данных. IDL-код, генерируемый средствами Delphi (он приведен ниже), будет воспринимать тип OLEVariant как IDL-конструкцию Any, и сгенерированная в Delphi структура CORBA разрешит преобразование этого значения в конструкцию Any, а также его корректный маршалинг в ORB-брокер и обратно. В действительности в библиотеке VCL Delphi существует объявленный тип, именуемый TAny, который напрямую переводится в тип Variant. Поэтому от нас требуется лишь создать массив данных типа Variant и передать его в качестве возвращаемого значения функции Data, как показано ниже.

```
function TQueryServer.Data: OleVariant;
var
  i : integer;
begin
  //Упаковка и отправка данных.
  Result := VarArrayCreate([0,FQuery.FieldCount-1],varOLEStr);
  for i := 0 to FQuery.FieldCount - 1 do
  begin
    Result[i] := FQuery.Fields[i].AsString;
  end;
end;
```

После реализации остальных методов мы получим модуль заглушки/каркаса, текст которого представлен в листинге 27.3.

Листинг 27.3. Модуль заглушки и каркаса для реализации интерфейса IQueryServer

```
unit SimpleCorbaServer_TLB;

// ************************************************************************
// ВНИМАНИЕ
// --------
// Типы, объявленные в этом файле, были сгенерированы из данных, считанных из
// библиотеки типов. Если эта библиотека типов явно или косвенно (через
// другую библиотеку типов, ссылающуюся на данную) будет реимпортирована,
// либо во время редактирования библиотеки типов будет выбрана
// команда редактора библиотеки типов 'Refresh' (Обновить), то
// содержимое этого файла будет регенерировано, а
// все внесенные вручную модификации будут утеряны.
```

```
// *********************************************************************

// PASTLWTR : $Revision:   1.88  $
// Файл сгенерирован 11/02/1999 6:01:08 PM из библиотеки типов, описанной ниже.

// *********************************************************************
// Type Lib: C:\ICON99\CORBA Server\SimpleCorbaServer.tlb (1)
// IID\LCID: {B7D4ED80-27C2-11D3-9703-0000861F6726}\0
// Helpfile:
// DepndLst:
//   (1) v2.0 stdole, (C:\WINDOWS\SYSTEM\STDOLE2.TLB)
//   (2) v4.0 StdVCL, (C:\WINDOWS\SYSTEM\STDVCL40.DLL)
// ********************************************************************* //
{$TYPEDADDRESS OFF} // Модуль должен быть скомпилирован без контроля
                    // типа используемых указателей,
interface

uses Windows, ActiveX, Classes, Graphics, OleServer, OleCtrls, StdVCL,
  SysUtils, CORBAObj, OrbPas, CorbaStd;

// *********************************************************************
// Уникальные идентификаторы GUID объявлены в библиотеке типов.
// Используются следующие префиксы:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// *********************************************************************
const
  // Главная и вспомогательная версии библиотек типов
  SimpleCorbaServerMajorVersion = 1;
  SimpleCorbaServerMinorVersion = 0;

  LIBID_SimpleCorbaServer: TGUID = '{B7D4ED80-27C2-11D3-9703-0000861F6726}';

  IID_IQueryServer: TGUID = '{B7D4ED81-27C2-11D3-9703-0000861F6726}';
  CLASS_QueryServer: TGUID = '{B7D4ED83-27C2-11D3-9703-0000861F6726}';
type

// *********************************************************************
// Объявление типов, определенных в библиотеке типов
// *********************************************************************
  IQueryServer = interface;
  IQueryServerDisp = dispinterface;

// *********************************************************************
// Объявление компонентных классов (CoClass), определенных в библиотеке типов
// (ПРИМЕЧАНИЕ. Здесь мы ставим каждый компонентный класс
// в соответствие его стандартному интерфейсу)
// *********************************************************************
  QueryServer = IQueryServer;
```

```
// *********************************************************************
// Interface: IQueryServer
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {B7D4ED81-27C2-11D3-9703-0000861F6726}
// *********************************************************************
  IQueryServer = interface(IDispatch)
    ['{B7D4ED81-27C2-11D3-9703-0000861F6726}']
    function  Login(const Db: WideString; const User: WideString;
      const Password: WideString): WordBool; safecall;
    function  Get_SQL: WideString; safecall;
    procedure Set_SQL(const Value: WideString); safecall;
    procedure Next; safecall;
    procedure Prev; safecall;
    procedure First; safecall;
    procedure Last; safecall;
    function  Get_FieldCount: Integer; safecall;
    function  Data: OleVariant; safecall;
    function  Get_EOF: WordBool; safecall;
    function  Get_BOF: WordBool; safecall;
    function  Execute: WordBool; safecall;
    property SQL: WideString read Get_SQL write Set_SQL;
    property FieldCount: Integer read Get_FieldCount;
    property EOF: WordBool read Get_EOF;
    property BOF: WordBool read Get_BOF;
  end;

// *********************************************************************
// DispIntf:  IQueryServerDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {B7D4ED81-27C2-11D3-9703-0000861F6726}
// *********************************************************************
  IQueryServerDisp = dispinterface
    ['{B7D4ED81-27C2-11D3-9703-0000861F6726}']
    function  Login(const Db: WideString; const User: WideString;
      const Password: WideString): WordBool; dispid 1;
    property SQL: WideString dispid 2;
    procedure Next; dispid 3;
    procedure Prev; dispid 4;
    procedure First; dispid 5;
    procedure Last; dispid 6;
    property FieldCount: Integer readonly dispid 7;
    function  Data: OleVariant; dispid 8;
    property EOF: WordBool readonly dispid 9;
    property BOF: WordBool readonly dispid 11;
    function  Execute: WordBool; dispid 12;
  end;

  TQueryServerStub = class(TCorbaDispatchStub, IQueryServer)
  public
    function Login(const Db: WideString; const User: WideString;
      const Password: WideString): WordBool; safecall;
```

```
    function Get_SQL: WideString; safecall;
    procedure Set_SQL(const Value: WideString); safecall;
    procedure Next; safecall;
    procedure Prev; safecall;
    procedure First; safecall;
    procedure Last; safecall;
    function Get_FieldCount: Integer; safecall;
    function Data: OleVariant; safecall;
    function Get_EOF: WordBool; safecall;
    function Get_BOF: WordBool; safecall;
    function Execute: WordBool; safecall;
  end;

  TQueryServerSkeleton = class(TCorbaSkeleton)
  private
    FIntf: IQueryServer;
  public
    constructor Create(const InstanceName: string; const Impl: IUnknown);
      override;
    procedure GetImplementation(out Impl: IUnknown); override; stdcall;
  published
    procedure Login(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Get_SQL(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Set_SQL(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Next(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Prev(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure First(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Last(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Get_FieldCount(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Data(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Get_EOF(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Get_BOF(const InBuf: IMarshalInBuffer; Cookie: Pointer);
    procedure Execute(const InBuf: IMarshalInBuffer; Cookie: Pointer);
  end;

// *********************************************************************
// В классе CoQueryServer предусмотрены методы Create и CreateRemote
// для создания экземпляров стандартного интерфейса IQueryServer,
// открываемого компонентным классом QueryServer. Эти функции
// предназначены для использования клиентами, желающими автоматизировать
// CoClass-объекты, открываемые сервером данной библиотеки типов.
// *********************************************************************
  CoQueryServer = class
    class function Create: IQueryServer;
    class function CreateRemote(const MachineName: string): IQueryServer;
  end;

  TQueryServerCorbaFactory = class
    class function CreateInstance(const InstanceName: string): IQueryServer;
  end;
```

```
implementation

uses ComObj;

{ TQueryServerStub }

function TQueryServerStub.Login(const Db: WideString; const User: WideString;
  const Password: WideString): WordBool;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Login', True, OutBuf);
  OutBuf.PutWideText(PWideChar(Pointer(Db)));
  OutBuf.PutWideText(PWideChar(Pointer(User)));
  OutBuf.PutWideText(PWideChar(Pointer(Password)));
  FStub.Invoke(OutBuf, InBuf);
  Result := UnmarshalWordBool(InBuf);
end;

function TQueryServerStub.Get_SQL: WideString;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Get_SQL', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
  Result := UnmarshalWideText(InBuf);
end;

procedure TQueryServerStub.Set_SQL(const Value: WideString);
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Set_SQL', True, OutBuf);
  OutBuf.PutWideText(PWideChar(Pointer(Value)));
  FStub.Invoke(OutBuf, InBuf);
end;

procedure TQueryServerStub.Next;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Next', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
end;

procedure TQueryServerStub.Prev;
var
```

```
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Prev', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
end;

procedure TQueryServerStub.First;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('First', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
end;

procedure TQueryServerStub.Last;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Last', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
end;

function TQueryServerStub.Get_FieldCount: Integer;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Get_FieldCount', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
  Result := InBuf.GetLong;
end;

function TQueryServerStub.Data: OleVariant;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Data', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
  Result := UnmarshalAny(InBuf);
end;

function TQueryServerStub.Get_EOF: WordBool;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Get_EOF', True, OutBuf);
```

```
  FStub.Invoke(OutBuf, InBuf);
  Result := UnmarshalWordBool(InBuf);
end;

function TQueryServerStub.Get_BOF: WordBool;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Get_BOF', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
  Result := UnmarshalWordBool(InBuf);
end;

function TQueryServerStub.Execute: WordBool;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Execute', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
  Result := UnmarshalWordBool(InBuf);
end;

{ TQueryServerSkeleton }

constructor TQueryServerSkeleton.Create(const InstanceName: string;
  const Impl: IUnknown);
begin
  inherited;
  inherited InitSkeleton('QueryServer', InstanceName,
    'IDL:SimpleCorbaServer/IQueryServer:1.0', tmMultiThreaded, True);
  FIntf := Impl as IQueryServer;
end;

procedure TQueryServerSkeleton.GetImplementation(out Impl: IUnknown);
begin
  Impl := FIntf;
end;

procedure TQueryServerSkeleton.Login(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
  Retval: WordBool;
  Db: WideString;
  User: WideString;
  Password: WideString;
begin
  Db := UnmarshalWideText(InBuf);
  User := UnmarshalWideText(InBuf);
```

```
  Password := UnmarshalWideText(InBuf);
  Retval := FIntf.Login(Db, User, Password);
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
  MarshalWordBool(OutBuf, Retval);
end;

procedure TQueryServerSkeleton.Get_SQL(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
  Retval: WideString;
begin
  Retval := FIntf.Get_SQL;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
  OutBuf.PutWideText(PWideChar(Pointer(Retval)));
end;

procedure TQueryServerSkeleton.Set_SQL(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
  Value: WideString;
begin
  Value := UnmarshalWideText(InBuf);
  FIntf.Set_SQL(Value);
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
end;

procedure TQueryServerSkeleton.Next(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
begin
  FIntf.Next;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
end;

procedure TQueryServerSkeleton.Prev(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
begin
  FIntf.Prev;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
end;

procedure TQueryServerSkeleton.First(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
begin
```

```
  FIntf.First;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
end;

procedure TQueryServerSkeleton.Last(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
begin
  FIntf.Last;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
end;

procedure TQueryServerSkeleton.Get_FieldCount(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
  Retval: Integer;
begin
  Retval := FIntf.Get_FieldCount;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
  OutBuf.PutLong(Retval);
end;

procedure TQueryServerSkeleton.Data(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
  Retval: OleVariant;
begin
  Retval := FIntf.Data;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
  MarshalAny(OutBuf, Retval);
end;

procedure TQueryServerSkeleton.Get_EOF(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
  Retval: WordBool;
begin
  Retval := FIntf.Get_EOF;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
  MarshalWordBool(OutBuf, Retval);
end;

procedure TQueryServerSkeleton.Get_BOF(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
  Retval: WordBool;
```

```
begin
  Retval := FIntf.Get_BOF;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
  MarshalWordBool(OutBuf, Retval);
end;

procedure TQueryServerSkeleton.Execute(const InBuf: IMarshalInBuffer;
  Cookie: Pointer);
var
  OutBuf: IMarshalOutBuffer;
  Retval: WordBool;
begin
  Retval := FIntf.Execute;
  FSkeleton.GetReplyBuffer(Cookie, OutBuf);
  MarshalWordBool(OutBuf, Retval);
end;

class function CoQueryServer.Create: IQueryServer;
begin
  Result := CreateComObject(CLASS_QueryServer) as IQueryServer;
end;

class function CoQueryServer.CreateRemote(const MachineName: string):
  IQueryServer;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_QueryServer) as
    IQueryServer;
end;

class function TQueryServerCorbaFactory.CreateInstance(
  const InstanceName: string): IQueryServer;
begin
  Result := CorbaFactoryCreateStub(
    'IDL:SimpleCorbaServer/QueryServerFactory:1.0', 'QueryServer',
    InstanceName, '', IQueryServer) as IQueryServer;
end;

initialization
  CorbaStubManager.RegisterStub(IQueryServer, TQueryServerStub);
  CorbaInterfaceIDManager.RegisterInterface(IQueryServer,
    'IDL:SimpleCorbaServer/IQueryServer:1.0');
  CorbaSkeletonManager.RegisterSkeleton(IQueryServer, TQueryServerSkeleton);

end.
```

Обратите внимание на то, что редактор библиотек типов в содружестве с мастерами Delphi сгенерировал весь код, необходимый для корректного маршалинга параметров. Сначала выполняется маршалинг параметров от заглушки к ORB-брокеру, а затем их демаршалинг от каркаса к реализации реального объекта.

Программный текст, который придется написать собственными силами, приведен в листинге 27.4. Рассмотрев его, нетрудно заметить, что в нем потребовалось позаботиться только о корректной реализации поведения создаваемого объекта, без какой-либо необходимости вникать в детали спецификации CORBA и маршалинга параметров.

Листинг 27.4. Модуль реализации интерфейса `TQueryServer`

```
unit uQueryServer;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, ComObj, StdVcl,
  CorbaObj, db, dbtables, orbpas, SimpleCorbaServer_TLB, frmqueryserver;

type

  TQueryServer = class(TCorbaImplementation, IQueryServer)
  private
    { Закрытые объявления }
    FDatabase: TDatabase;
    FQuery: TQuery;
  public
    { Открытые объявления }
    constructor Create(Controller: IObject; AFactory: TCorbaFactory); override;
    destructor Destroy; override;
  protected
    function Data: OleVariant; safecall;
    function Get_BOF: WordBool; safecall;
    function Get_EOF: WordBool; safecall;
    function Get_FieldCount: Integer; safecall;
    function Get_SQL: WideString; safecall;
    function Login(const Db, User, Password: WideString): WordBool; safecall;
    procedure First; safecall;
    procedure Last; safecall;
    procedure Next; safecall;
    procedure Prev; safecall;
    procedure Set_SQL(const Value: WideString); safecall;
    function Execute: WordBool; safecall;
  end;

implementation

uses CorbInit;

function TQueryServer.Data: OleVariant;
var
  i : integer;
begin
  // Упаковка и отправка данных.
```

```
  Result := VarArrayCreate([0,FQuery.FieldCount-1],varOLEStr);
  for i := 0 to FQuery.FieldCount - 1 do
  begin
    Result[i] := FQuery.Fields[i].AsString;
  end;
end;

function TQueryServer.Get_BOF: WordBool;
begin
  Result := FQuery.BOF;
end;

function TQueryServer.Get_EOF: WordBool;
begin
  Result := FQuery.EOF;
end;

function TQueryServer.Get_FieldCount: Integer;
begin
  Result := FQuery.FieldCount;
end;

function TQueryServer.Get_SQL: WideString;
begin
  Result := FQuery.SQL.Text;
end;

function TQueryServer.Login(const Db, User,
  Password: WideString): WordBool;
begin
  if FDatabase.Connected then FDatabase.Close;
  FDatabase.AliasName := Db;
  FDatabase.Params.Clear;
  FDatabase.Params.Add('USER NAME=' + User);
  FDatabase.Params.Add('PASSWORD=' + Password);
  FDatabase.Open;
end;

procedure TQueryServer.First;
begin
  FQuery.First;
end;

procedure TQueryServer.Last;
begin
  FQuery.Last;
end;

procedure TQueryServer.Next;
begin
```

```
  FQuery.Next;
end;

procedure TQueryServer.Prev;
begin
  FQuery.Prior;
end;

procedure TQueryServer.Set_SQL(const Value: WideString);
begin
  FQuery.SQL.Clear;
  FQuery.SQL.Add(Value);
end;

constructor TQueryServer.Create(Controller: IObject;
  AFactory: TCorbaFactory);
begin
  inherited Create(Controller,AFactory);
  FDatabase := TDatabase.Create(nil);
  FDatabase.LoginPrompt := false;
  FDatabase.DatabaseName := 'CorbaDb';
  FDatabase.HandleShared := true;
  FQuery := TQuery.Create(nil);
  FQuery.DatabaseName := 'CorbaDb';
end;

destructor TQueryServer.Destroy;
begin
  FQuery.Free;
  FDatabase.Free;
  inherited Destroy;
end;

function TQueryServer.Execute: WordBool;
begin
  FQuery.Close;
  FQuery.Open;
end;

initialization
  TCorbaObjectFactory.Create('QueryServerFactory', 'QueryServer',
    'IDL:SimpleCorbaServer/QueryServerFactory:1.0', IQueryServer,
    TQueryServer, iMultiInstance, tmSingleThread);
end.
```

В коде из листинга 27.4 следует обратить внимание на одну деталь, имеющую отношение к библиотеке VCL. Речь идет о корректной работе с объектом TDatabase. Пространство имен BDE предусматривает взаимодействие внутри одного и того же сеанса работы только с одной базой данных с уникальным именем. Поскольку внутри нашего CORBA-сервера могло бы существовать несколько объектов TQueryServer, которые совместно используют один и тот

же объект TSession, необходимо установить свойство HandleShared компонента TDatabase равным значению True. Если этого не сделать, следующий клиент, который захочет создать новый экземпляр объекта TQueryServer, не сможет соединиться с сервером.

В окне редактора библиотек типов можно просмотреть код IDL, который представляет собой наш интерфейс. Щелкните на направленной вниз стрелке, изображенной на кнопке Export to IDL панели инструментов окна этого редактора, и выберите в раскрывающемся списке вариант Export to CORBA IDL (обратите внимание, что этот вариант подобен, но все-таки отличается от варианта Microsoft IDL (или MIDL)). Код IDL можно просмотреть в самом редакторе Delphi или в листинге 27.5.

Листинг 27.5. IDL-вариант интерфейса IQueryServer CORBA-сервера

```
module SimpleCorbaServer
{
  interface IQueryServer;


  interface IQueryServer
  {
    boolean Login(in wstring Db, in wstring User, in wstring Password);
    wstring Get_SQL();
    wstring Set_SQL(in wstring Value);
    void Next();
    void Prev();
    void First();
    void Last();
    long Get_FieldCount();
    any Data();
    boolean Get_EOF();
    boolean Get_BOF();
    boolean Execute();
  };

  interface QueryServerFactory
  {
    IQueryServer CreateInstance(in string InstanceName);
  };

};
```

Обратите внимание на то, что все COM-типы данных, которые мы выбрали в редакторе библиотек типов, должным образом преобразованы в их IDL-эквиваленты. Этот код на языке IDL может быть импортирован любым другим инструментом, поддерживающим спецификацию CORBA. Такие инструментальные средства разработки, как CBuilder и JBuilder, генерируют классы оболочек, чтобы написанные на этих языках клиенты могли без затруднений использовать функциональность нашего CORBA-объекта, созданного в Delphi.

На заметку

Сгенерированный средствами Delphi код на языке IDL (см. листинг 27.5) на самом деле слегка некорректен. Функция `Set_SQL` не должна возвращать никакого значения. Хотя Delphi и справится с этим недоразумением, проблема вытекает из того факта, что в редакторе библиотек типов мы добавили свойство (`SQL`). Свойства распознаются средствами COM, но они не являются теми конструкциями, которые обычно присутствуют в CORBA-объектах. В среде Delphi для этого свойства созданы методы чтения и записи, но метод записи не экспортирован корректно в код IDL. Этой проблемы можно легко избежать, объявив в CORBA-интерфейсах только методы или вручную отредактировав объявление в сгенерированном редактором IDL-коде так, как показано ниже.

```
void Set_SQL(in wstring Value);
```

Запуск CORBA-сервера

Создание сервера запросов завершено. Теперь пора запустить приложение CORBA-сервера и дать знать ORB-брокеру VisiBroker о том, что данный объект доступен для клиентов. Чтобы клиенты могли отыскать реализацию любого CORBA-объекта и подключиться к ней с помощью ORB-брокера VisiBroker, на одном из узлов локальной сети должен функционировать агент Smart Agent. Этому агенту вовсе не обязательно работать на одном компьютере с клиентом или сервером. Службу Smart Agent можно запустить из командной строки (в среде Windows NT агент Smart Agent может выполняться как сервисная служба), введя такую команду:

`OSAGENT [-`*`опции`*`]`

где возможными опциями являются следующие:

- `-p` — задает порт, который будет прослушиваться агентом;
- `-v` — активизирует вывод отладочной информации в файл `osagent.log`;
- `-?` — активизирует вывод в файл `osagent.log` информации об использовании;
- `-C` — запускает службу `osagent` в консольном режиме (только в среде Windows NT; принимается по умолчанию в среде Windows 95/98).

При запуске агента Smart Agent в среде Windows NT вручную важно ввести команду `osagent` с использованием ключа `-c`. Это позволит службе `osagent`, которая была инсталлирована как служба Windows NT, работать в качестве консольного приложения. Пример запуска агента Smart Agent в среде Windows NT как консольного приложения для прослушивания запросов через порт 14005 будет выглядеть следующим образом:

```
osagent -c  -p 14005.
```

Если агент Smart Agent уже работает в сети, можно запустить проект, который мы только что завершили — он зарегистрирует себя с использованием агента Smart Agent, в результате чего станет доступным для соединений с клиентами. Заметьте, что на данном этапе необходимо действительно запустить серверное приложение. Для запуска сервера не предусмотрено встроенного средства (как в модели DCOM), за исключением тех случаев, когда используется служба OAD.

Построение CORBA-клиента с использованием раннего связывания

Теперь, когда у нас есть доступный CORBA-сервер, мы можем перейти к следующему этапу и средствами Delphi создать клиентское CORBA-приложение. Не мудрствуя лукаво, построим простейшее клиентское приложение, в котором разработанный нами интерфейс `IQueryServer` будет использоваться для считывания данных из сервера и заполнения считанными данными сетки с одной строкой. Важно понимать, что здесь мы активно пользуемся преимуществами многоуровневой

архитектуры — создаваемому клиенту необходимо лишь получить доступ к программному обеспечению ORB-брокера VisiBroker и совершенно не требуется "знать" что-либо о наборах данных Delphi или механизме доступа к базам данных, именуемом Borland Database Engine (BDE).

CORBA-клиент может общаться с CORBA-объектом двумя способами: с использованием раннего или позднего связывания. Раннее связывание означает, что компилятор может компилировать непосредственные обращения к виртуальной таблице (v-table) заглушки. В этом случае положительный эффект ощущается не только в смысле выигрыша в производительности, но и в том, что компилятор может обеспечить проверку типов, чтобы гарантировать передачу параметров с корректными типами данных. В сценарии позднего связывания все удаленные вызовы выполняются с использованием типа данных `Any`. Такие вызовы работают медленнее, поскольку информация о параметрах должна быть получена из хранилища интерфейсов (Interface Repository) брокера VisiBroker, и некорректные типы параметров не будут выявлены до наступления времени выполнения. Чтобы обеспечить раннее связывание Delphi-объекта с заглушкой, компилятору должно быть предоставлено некоторое Pascal-представление интерфейса заглушки. В случае объектов, построенных в других языках, эта задача весьма усложняется, поскольку Delphi 5 пока не включает утилиты перевода IDL-файлов на язык Pascal. В нашем случае сервер построен в среде Delphi, и мастера Delphi автоматически сгенерировали Pascal-версию интерфейса заглушки. Следовательно, применить к нашему серверу раннее связывание можно с помощью простого включения имени модуля заглушки и каркаса из предыдущего примера в директиву `uses` модуля нашего клиента.

Создание CORBA-клиента

Сначала создадим в среде Delphi простое GUI-приложение, предназначенное для просмотра результатов, получаемых при использовании интерфейса `IQueryServer`, как показано на рис. 27.8.

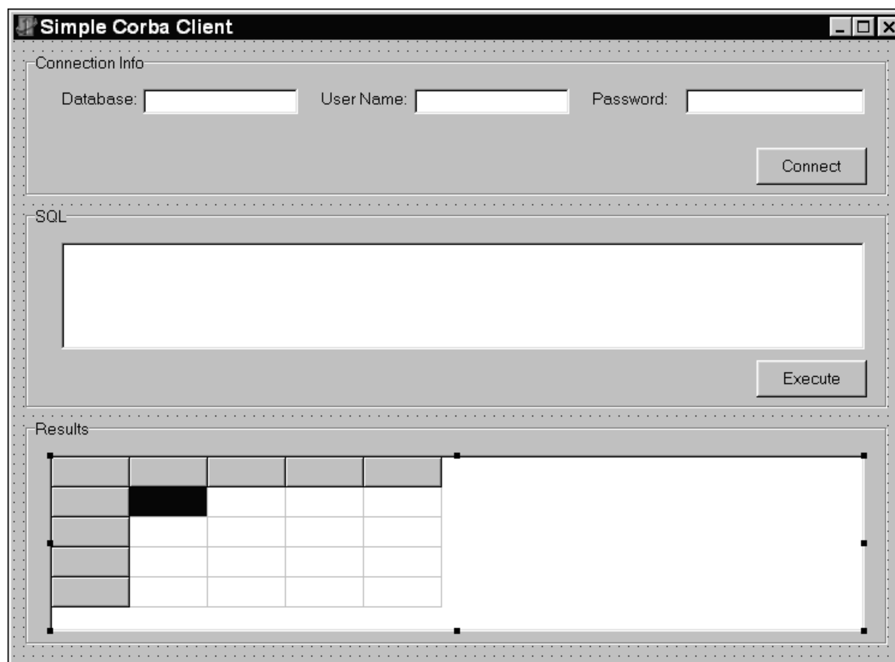


Рис. 27.8. Общий вид формы разрабатываемого CORBA-клиента

После этого добавим имя модуля заглушки и каркаса из примера сервера в директиву `uses` модуля формы (`SimpleCorbaServer_TLB.pas`).

Подключение к CORBA-серверу

Нам остается лишь подключиться к нашему серверу и приступить к формированию обращений к удаленному интерфейсу. Используемый модуль заглушки и каркаса определяет фабрику класса для интерфейса IQueryServer (с именем TQueryServerCorbaFactory). Этот класс содержит функцию класса CreateInstance (поэтому нам не нужно создавать экземпляр класса TQueryServerCorbaFactory), которая будет создавать соответствующий объект заглушки и возвращать интерфейс IQueryServer. Теперь можно перейти к созданию обращений к удаленному интерфейсу IQueryServer с использованием сценария раннего связывания. Единственным нетривиальным моментом в работе этого клиента является вызов метода Data интерфейса IQueryServer и заполнение сетки строки данными из массива типа OLEVariant. Эти действия осуществляются в событии ExecuteClick нашего клиента. Полная реализация CORBA-клиента представлена в листинге 27.6.

Листинг 27.6. Реализация CORBA-клиента `SimpleCorbaClient`

```
unit ufrmCorbaClient;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, SimpleCorbaServer_TLB, corbaObj, Grids;

type
  TForm1 = class(TForm)
    GroupBox1: TGroupBox;
    Label2: TLabel;
    edtDatabase: TEdit;
    Label3: TLabel;
    edtUserName: TEdit;
    Label4: TLabel;
    edtPassword: TEdit;
    Button5: TButton;
    GroupBox2: TGroupBox;
    memoSQL: TMemo;
    GroupBox3: TGroupBox;
    Button6: TButton;
    grdCorbaData: TStringGrid;
    procedure ConnectClick(Sender: TObject);
    procedure ExecuteClick(Sender: TObject);
  private
    { Закрытые объявления }
    FQueryServer: IQueryServer;
  public
    { Открытые объявления }
  end;
var
  Form1: TForm1;
```

```
implementation

{$R *.DFM}

procedure TForm1.ConnectClick(Sender: TObject);
begin
  if not(assigned(FQueryServer)) then
    FQueryServer := TQueryServerCorbaFactory.CreateInstance('SimpleServer');
  FQueryServer.Login(edtDatabase.Text,edtUserName.Text,edtPassword.Text);
end;

procedure TForm1.ExecuteClick(Sender: TObject);
var
  i,j: integer;
  CorbaData : OLEVariant;
begin
  FQueryServer.SQL := memoSQL.Text;
  FQueryServer.Execute;

  grdCorbaData.ColCount := FQueryServer.FieldCount;
  grdCorbaData.RowCount := 0;
  j := 0;

  while not(FQueryServer.EOF) do
  begin
    inc(j);
    grdCorbaData.RowCount := j;
    CorbaData := (FQueryServer.Data);
    for i := 0 to FQueryServer.FieldCount - 1 do
    begin
      grdCorbaData.Cells[i + 1,j-1] := CorbaData[i];
    end;
    FQueryServer.Next;
  end;
end;

end.
```

Если агент Smart Agent уже запущен и CORBA-сервер работает “в поле зрения” этого агента, запускайте приложение клиента и считывайте данные из CORBA-сервера сколько угодно раз!

Построение CORBA-клиента с использованием сценария позднего связывания

Теперь попытаемся так модифицировать текст только что созданного клиентского CORBA-приложения, чтобы для связи с удаленным интерфейсом использовался сценарий позднего связывания. В технологии CORBA мы используем то, что называется *интерфейсом динамических вызовов* (Dynamic Invocation Interface, или DII). Позднее связывание в нашем случае

не является обязательным, поскольку как сервер, так и клиент были разработаны с помощью средств Delphi. Однако, если вы хотите без особых проблем использовать CORBA-серверы, разработанные в других языках, знакомство с этой методикой будет весьма полезным.

Во-первых, следует удалить имя модуля заглушки и каркаса из директивы `uses` модуля формы. Помните, что, если бы сервер был написан на языке Java (к примеру), определить его использование в этой директиве было бы просто невозможно.

Во-вторых, наш клиент "ничего не знает" об интерфейсе `IQueryServer`. Поэтому для инкапсулированного поля `FQueryServer` тип данных `IQueryServer` заменим типом `TAny`.

В-третьих, общую (групповую) CORBA-заглушку следует получить иным способом, отличным от примененного в предыдущем случае. Необходимо вызвать глобальный метод языка Pascal `CorbaBind` (из модуля `CorbaObj`) и передать ему идентификатор требуемой фабрики классов в хранилище интерфейсов. Получив доступ к фабрике, следует вызвать метод `CreateInstance` этой фабрики, которая вернет общий интерфейс. Этот интерфейс требуется сохранить в переменной типа `Any`, после чего, пользуясь полученной ссылкой, вызвать методы позднего связывания. Полный исходный код клиента с использованием предложенного выше сценария позднего связывания приведен в листинге 27.7.

Листинг 27.7. Клиентское CORBA-приложение, использующее позднее связывание для отправки запросов серверу

```
unit ufrmCorbaClientLate;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, corbaObj, Grids;

type
  TForm1 = class(TForm)
    GroupBox1: TGroupBox;
    Label2: TLabel;
    edtDatabase: TEdit;
    Label3: TLabel;
    edtUserName: TEdit;
    Label4: TLabel;
    edtPassword: TEdit;
    Button5: TButton;
    GroupBox2: TGroupBox;
    memoSQL: TMemo;
    GroupBox3: TGroupBox;
    Button6: TButton;
    grdCorbaData: TStringGrid;
    procedure ConnectClick(Sender: TObject);
    procedure ExecuteClick(Sender: TObject);
  private
    { Закрытые объявления }
    FQueryServer: TAny;
  public
    { Открытые объявления }
```

```
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.ConnectClick(Sender: TObject);
var
  Factory: TAny;
  User, Pass: WideString;
begin
  Factory := CorbaBind('IDL:SimpleCorbaServer/QueryServerFactory:1.0');
  FQueryServer := Factory.CreateInstance('');
  User := WideString(edtUserName.Text);
  Pass := WideString(edtPassword.Text);
  FQueryServer.Login(WideString(edtDatabase.Text),User,Pass);
end;

procedure TForm1.ExecuteClick(Sender: TObject);
var
  i,j: integer;
  CorbaData : OLEVariant;
begin
  FQueryServer.Set_SQL((memoSQL.Text));
  FQueryServer.Execute;

  grdCorbaData.ColCount := FQueryServer.Get_FieldCount;
  grdCorbaData.RowCount := 0;
  j := 0;

  while not(FQueryServer.Get_EOF) do
  begin
    inc(j);
    grdCorbaData.RowCount := j;
    CorbaData := FQueryServer.Data;
    for i := 0 to FQueryServer.Get_FieldCount - 1 do
    begin
      grdCorbaData.Cells[i + 1,j-1] := CorbaData[i];
    end;
    FQueryServer.Next;
  end;
end;

end.
```

В исходном коде клиента, использующего сценарий позднего связывания с сервером, нетрудно заметить и некоторые другие изменения.

В языке IDL, в отличие от технологии COM, не поддерживается понятие “свойства”. При использовании раннего связывания можно обойтись и без этого, поскольку компилятор обычным образом разрешает адресные ссылки, используемые для получения доступа к методам чтения и установки свойств. При использовании же позднего связывания интерфейс DII не имеет “понятия” ни о каких свойствах, поэтому методы доступа к свойствам необходимо будет вызывать в явном виде. Например, вместо чтения значения `FieldCount` нужно вызвать метод `Get_FieldCount`.

Все параметры интерфейса DII передаются как типы `Any`, в которых также хранятся сведения и о реальном типе данных. Для корректной установки типа данных `Any` для некоторых значений нужно явно выполнять операцию приведения типа. Например, отправка строкового значения для параметра `Db` метода `Login` вызовет необходимость установки типа, хранящегося в типе `Any`, равным типу `varString`. Это приведет к возникновению ошибки использования неверных параметров, если строка не будет приведена к типу `WideString`, для того чтобы тип, хранящийся в типе `Any`, был установлен равным типу `varOleStr` (`WideString`).

Наконец, в дополнение к агенту Smart Agent, где-нибудь в локальной сети должна работать также служба хранилища интерфейсов (Interface Repository) брокера VisiBroker, и интерфейс `IQueryServer` должен быть зарегистрирован в этом хранилище интерфейсов. Хранилище интерфейсов можно сравнить с оперативной базой данных, которая позволяет ORB-брокеру находить нужную информацию об интерфейсах для использования совместно с интерфейсом DII. Службу хранилища интерфейсов ORB-брокера VisiBroker можно запустить из командной строки, с помощью следующей команды:

```
IREP [-console] IRname [file.idl]
```

Единственным обязательным аргументом здесь является аргумент `IRname`. Поскольку одновременно могут работать несколько экземпляров службы хранилища интерфейсов, каждый из них потребует идентификации. Аргумент *`-console`* указывает, что службу хранилища интерфейсов необходимо запустить в режиме консоли (по умолчанию устанавливается режим GUI), а с помощью аргумента *`file.idl`* можно задать IDL-файл инициализации, загружаемый при запуске хранилища интерфейсов. Дополнительные IDL-файлы можно загрузить, используя опцию меню (при работе в режиме GUI) или запустив утилиту `idl2ir`.

Кросс-язык CORBA

На момент написания этой книги в Delphi еще не был представлен компилятор `Idl2Pas` фирмы Inprise, тем не менее, существует предварительная версия этого инструментального средства. В данном разделе мы рассмотрим действия, необходимые для ручного выполнения раннего связывания с CORBA-сервером, написанным на другом языке, а также познакомимся с компилятором `Idl2Pas`, который должен быть выпущен в свет в ближайшем будущем.

Ручной маршалинг для CORBA-сервера, написанного на языке Java

В следующем примере используется очень простой CORBA-сервер, созданный на языке Java (JBuilder), который будет вызываться из приложения Delphi. Код IDL этого CORBA-сервера представлен в листинге 27.8.

Листинг 27.8. IDL-код простого Java-сервера

```
module CorbaServer {
  interface SimpleText {
    string setText(in string txt);
  };
};
```

Если CORBA-сервер был зарегистрирован с использованием службы Interface Repository, то Delphi сможет легко получить к нему доступ с помощью интерфейса DII (реализация этих действий приведена в листинге 27.9 — в методе `btnDelphiTextEarly`).

Чтобы реализовать сценарий раннего связывания без использования компилятора `Idl2Pas`, необходимо вручную написать код собственного класса заглушки для выполнения маршалинга передаваемых параметров. И хотя эту задачу трудно сравнивать с проблемами точного ракетостроения, но в случае использования большого числа методов данное действие может представлять собой достаточно утомительный процесс, способствующий появлению различных ошибок. Кроме того, необходимо зарегистрировать класс заглушки и интерфейс для класса заглушки, используя для этого надлежащие механизмы Delphi. Полный код доступа к серверу представлен в листинге 27.9.

Листинг 27.9. Код доступа к Java-серверу из Delphi-клиента (по сценариям раннего и позднего связывания)

```
unit uDelphiClient;

interface

uses
  Windows, Messages, SysUtils, CorbInit, CorbaObj, orbpas, Classes,
    Graphics, Controls, Forms, Dialogs,  StdCtrls;

type

  ISimpleText = interface
  ['{49F25940-3C3C-11D3-9703-0000861F6726}']
    function SetText(const txt: String): String;
  end;

  TSimpleTextStub = class(TCorbaStub, ISimpleText)
  public
    function SetText(const txt: String): String;
  end;

  TForm1 = class(TForm)
    edtDelphiText: TEdit;
    btnDelphiTextLate: TButton;
    btnDelphiTextEarlyClick: TButton;
    edtResult: TEdit;
    procedure btnDelphiTextLateClick(Sender: TObject);
    procedure btnDelphiTextEarlyClickClick(Sender: TObject);
```

```
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.btnDelphiTextLateClick(Sender: TObject);
var
  JavaServer: TAny;
begin
  JavaServer := ORB.Bind('IDL:CorbaServer/SimpleText:1.0');
  edtResult.Text := JavaServer.setText(edtDelphiText.text);
end;

{ TSimpleTextStub }

function TSimpleTextStub.SetText(const txt: String): String;
var
  InBuf: IMarshalInBuffer;
  OutBuf: IMarshalOutBuffer;
begin
  FStub.CreateRequest('setText',True,OutBuf);
  OutBuf.PutText(pchar(txt));
  FStub.Invoke(OutBuf, InBuf);
  Result := UnmarshalText(InBuf);
end;

procedure TForm1.btnDelphiTextEarlyClickClick(Sender: TObject);
var
  JavaServer: ISimpleText;
begin
  JavaServer := CorbaBind(ISimpleText) as ISimpleText;
  edtResult.Text := JavaServer.SetText(edtDelphiText.text);
end;

initialization
  CorbaStubManager.RegisterStub(ISimpleText, TSimpleTextStub);
  CorbaInterfaceIDManager.RegisterInterface(ISimpleText,
    'IDL:CorbaServer/SimpleText:1.0');

end.
```

Нетрудно заметить, что приведенный выше код очень напоминает код, сгенерированный редактором библиотек типов при создании CORBA-объекта в среде Delphi. Мы добавили наш собственный потомок класса TCorbaStub, предназначенный для обеспечения маршалинга на стороне клиента. Обратите внимание на отсутствие необходимости в выведении потомка из класса TCorbaDispatchStub (поскольку использование редактора библиотек типов здесь не предусматривается). Затем реализуется собственная заглушка, предназначенная для маршалинга параметров "в" и "из" CORBA-интерфейсов буферов маршалинга: IMarshalInBuffer и IMarshalOutBuffer. Эти интерфейсы содержат удобные методы, предназначенные для чтения и записи различных типов данных в буферы. За более подробной информацией об использовании этих методов лучше всего обратиться к интерактивной справочной системе Delphi 5. Наконец, необходимо зарегистрировать созданную пользовательскую заглушку и интерфейс с помощью CORBA-структуры в среде Delphi. Этот код приведен в разделе initialization данного модуля.

Компилятор Idl2Pas фирмы Inprise

Рассмотрев листинг 27.9, вы наверняка поймете, что ручной маршалинг крупного CORBA-объекта потребует выполнения большого объема работы. Решение этой проблемы состоит в использовании способности компилятора Idl2Pas автоматически генерировать соответствующий код маршалинга для нашей заглушки. К тому времени, когда вы будете читать эту главу, этот инструмент, вероятно, сможет приобрести в фирме Inprise каждый желающий. Мы решили завершить данный раздел кратким обзором существующей (предварительной) версии компилятора Idl2Pas.

Компилятор Idl2Pas фирмы Inprise реализован на языке Java и, следовательно, для его установки на компьютере потребуется наличие продукта Java VM. При инсталляции Delphi 5 в качестве подходящего варианта предлагается программа Java Runtime Environment (JRE). Настоящая предварительная версия компилятора Idl2Pas пока не интегрирована в оболочку IDE Delphi, поэтому следует вызывать этот компилятор из командной строки, используя специальный пакетный файл Idl2Pas.bat. Команда, предназначенная для вызова компилятора Idl2Pas (с использованием файла SimpleText.idl) и сохранения сгенерированных файлов в папке c:\idl, будет иметь следующий вид:

```
IDL2PAS -root_dir c:\idl SimpleText.idl
```

Компилятор Idl2Pas сгенерирует в указанной папке два файла, имена которых будут зависеть от имени модуля, включенного в idl-файл. В нашем примере модуль CorbaServer_i.pas будет содержать Pascal-объявления idl-интерфейсов (листинг 27.10).

Листинг 27.10. Определения интерфейсов, сгенерированные компилятором Idl2Pas

```
unit CorbaServer_i;

// Этот файл был сгенерирован 4 ноября 1999 17:58:12 GMT версией
// 01.09.00.A2.032c CORBA-IDL-компилятора idl2pas
// (брокер VisiBroker) компании Inprise.

// Delphi-Pascal-модуль CorbaServer_i для IDL-модуля CorbaServer.
// Назначение этого файла - объявить интерфейсы и переменные, используемые в
// присоединяемых модулях клиента CorbaServer_c и/или сервера CorbaServer_s

//Этот модуль содержит pascal-код интерфейса для IDL-модуля CorbaServer.
```

```
 (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                      SimpleText.idl", line 1
** IDL Name       : module
** Repository Id  : IDL:CorbaServer:1.0
** IDL definition :
*)

interface

uses
  CORBA;

type
  // Эти ссылки используются для разрешения зависимостей между
  // следующими интерфейсами.
  SimpleText = interface;
  // Эти определения интерфейсов были сгенерированы на основании
  // кода IDL, на основе которого и был сгенерирован этот модуль.

  //Сигнатура для интерфейса "CorbaServer_i.SimpleText" является
  // производной от IDL-интерфейса "SimpleText".

  (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                      SimpleText.idl", line 2
  ** IDL Name       : interface
  ** Repository Id  : IDL:CorbaServer/SimpleText:1.0
  ** IDL definition :
  *)
  SimpleText = interface
    ['{C8864064-C211-B145-29DB-CD5119D884CD}']

    // Методы интерфейса, представляющие IDL-операции.

    (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                        SimpleText.idl", line 3
    ** IDL Name       : operation
    ** Repository Id  : IDL:CorbaServer/SimpleText/setText:1.0
    ** IDL definition :
    *)
    function   setText (const txt : AnsiString): AnsiString;
  end;

implementation

  // Код реализации (если таковой существует)
  // находится в присоединенном _C-файле.

initialization

end.
```

Второй, сгенерированный компилятором Idl2Pas файл, CorbaServer_c.pas, содержит код реализации класса заглушки, а также вспомогательного объекта (TSimpleTextHelper), который облегчает процесс передачи таких непростых типов данных, как строки, объединения, а также типов данных, определенных пользователем. Сгенерированный код реализации приведен в листинге 27.11.

Листинг 27.11. Класс заглушки и вспомогательный класс, сгенерированные компилятором Idl2Pas

```
unit CorbaServer_c;

// c:\icon99\MultiLanguage\MyProjects\CorbaServer\SimpleText.idl.

// Delphi-Pascal-модуль CorbaServer_i для IDL-модуля CorbaServer.
// Назначние этого файла - реализовать классы, работающие на стороне
// клиента (заглушки), требуемые присоединенным модулем интерфейса
// (CorbaServer_i). Этот модуль должен соответствовать
// присоединенному модулю каркаса на стороне сервера.

// Этот модуль содержит код заглушки для IDL-модуля CorbaServer.

(* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                     SimpleText.idl", line 1
** IDL Name       : module
** Repository Id  : IDL:CorbaServer:1.0
** IDL definition :
*)

interface

uses
  CORBA,
  CorbaServer_i;

type
  // Эти ссылки поддерживаются для разрешения зависимостей между
  // следующими интерфейсами.
  TSimpleTextHelper = class;
  TSimpleTextStub = class;
  // Эти интерфейсы заглушки и вспомогательного объекта были сгенерированы
  // на основании IDL-кода, из которого был произведен этот модуль.

  // Вспомогательный Pascal-класс "CorbaServer_c.TSimpleTextHelper" для
  // Pascal-интерфейса "CorbaServer_i.SimpleText".

  (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                       SimpleText.idl", line 2
  ** IDL Name       : interface
  ** Repository Id  : IDL:CorbaServer/SimpleText:1.0
  ** IDL definition :
```

```
  *)

  TSimpleTextHelper = class
    class procedure Insert(const A: CORBA.Any;
      const Value: CorbaServer_i.SimpleText);
    class function  Extract(const A: CORBA.Any): CorbaServer_i.SimpleText;
    class function  TypeCode: CORBA.TypeCode;
    class function  RepositoryId: string;
    class function  Read(const Input: CORBA.InputStream):
      CorbaServer_i.SimpleText;
    class procedure Write(const Output: CORBA.OutputStream;
      const Value: CorbaServer_i.SimpleText);
    class function  Narrow(const Obj: CORBA.CORBAObject; IsA: Boolean = False):
      CorbaServer_i.SimpleText;
    class function  Bind(const InstanceName: string = '';
      HostName : string = ''): CorbaServer_i.SimpleText; overload;
    class function  Bind(Options: BindOptions;
      const InstanceName: string = ''; HostName: string = ''):
      CorbaServer_i.SimpleText; overload;
  end;

  // Pascal-класс заглушки "CorbaServer_c.TSimpleTextStub, поддерживающий
  // Pascal-интерфейс "CorbaServer_i.SimpleText".

  (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                       SimpleText.idl", line 2
  ** IDL Name       : interface
  ** Repository Id  : IDL:CorbaServer/SimpleText:1.0
  ** IDL definition :
  *)

  TSimpleTextStub = class(CORBA.TCORBAObject, CorbaServer_i.SimpleText)
  public

    (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                         SimpleText.idl", line 3
    ** IDL Name       : operation
    ** Repository Id  : IDL:CorbaServer/SimpleText/setText:1.0
    ** IDL definition :
    *)
    function setText ( const txt : AnsiString): AnsiString; virtual;

  end;

implementation
  // Эти реализации заглушки и вспомогательного объекта были сгенерированы
  // на основании IDL-кода, от которого и произошел этот модуль.

  // Реализация вспомогательного Pascal-класса "CorbaServer_c.TSimpleTextHelper",
  // поддерживающего Pascal-интерфейс "CorbaServer_i.SimpleText.
```

```
(* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                     SimpleText.idl", line 2
** IDL Name       : interface
** Repository Id  : IDL:CorbaServer/SimpleText:1.0
** IDL definition :
*)

class procedure TSimpleTextHelper.Insert(const A: CORBA.Any;
  const Value: CorbaServer_i.SimpleText);
begin
  // TAnyHelper.InsertObject(Value);
end;

class function TSimpleTextHelper.Extract(const A: CORBA.Any):
  CorbaServer_i.SimpleText;
begin
  // TAnyHelper.ExtractObject as CorbaServer_i.SimpleText;
end;

class function TSimpleTextHelper.TypeCode: CORBA.TypeCode;
begin
  Result := ORB.CreateInterfaceTC(RepositoryId, 'CorbaServer_i.SimpleText');
end;

class function TSimpleTextHelper.RepositoryId: string;
begin
  Result := 'IDL:CorbaServer/SimpleText:1.0';
end;

class function TSimpleTextHelper.Read(const Input: CORBA.InputStream):
  CorbaServer_i.SimpleText;
var
  Obj: CORBA.CORBAObject;
begin
  Input.ReadObject(Obj);
  Result := Narrow(Obj, True)
end;

class procedure TSimpleTextHelper.Write(const Output: CORBA.OutputStream;
  const Value: CorbaServer_i.SimpleText);
begin
  Output.WriteObject(Value as CORBA.CORBAObject);
end;

class function TSimpleTextHelper.Narrow(const Obj: CORBA.CORBAObject;
  IsA: Boolean): CorbaServer_i.SimpleText;
begin
  Result := nil;
  if (Obj = nil) or (Obj.QueryInterface(CorbaServer_i.SimpleText, Result) = 0)
    then Exit;
```

```
    if IsA and Obj._IsA(RepositoryId) then
      Result := TSimpleTextStub.Create(Obj);
  end;

class function TSimpleTextHelper.Bind(const InstanceName: string = '';
    HostName: string = ''): CorbaServer_i.SimpleText;
  begin
    Result := Narrow(ORB.bind(RepositoryId, InstanceName, HostName), True);
  end;

  class function TSimpleTextHelper.Bind(
    Options: BindOPtions; const InstanceName: string = '';
    HostName: string = ''): CorbaServer_i.SimpleText;
  begin
    Result := Narrow(ORB.bind(RepositoryId, Options, InstanceName, HostName),
      True);
  end;
  // Реализация Pascal-класса заглушки "CorbaServer_c.TSimpleTextStub",
  // поддерживающего Pascal-интерфейс "CorbaServer_i.SimpleText".

  // Реализация методов интерфейса, представляющих IDL-операции.

  (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                        SimpleText.idl", line 3
  ** IDL Name       : operation
  ** Repository Id  : IDL:CorbaServer/SimpleText/setText:1.0
  ** IDL definition :
  *)
  function   TSimpleTextStub.setText ( const txt : AnsiString): AnsiString;
  var
    Output: CORBA.OutputStream;
    Input : CORBA.InputStream;
  begin
    inherited _CreateRequest('setText',True, Output);
    Output.WriteString(txt);
    inherited _Invoke(Output, Input);
    Input.ReadString(Result);
  end;

initialization

  // Эти вызовы инициализации заглушки и вспомогательного объекта были
  // сгенерированы на основании IDL-кода, от которого и произошел этот модуль.

  // Инициализация вспомогательного Pascal-класса
  // "CorbaServer_c.TSimpleTextStub".

  (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                        SimpleText.idl", line 2
  ** IDL Name       : interface
```

```
  ** Repository Id  : IDL:CorbaServer/SimpleText:1.0
  ** IDL definition :
  *)
  CORBA.InterfaceIDManager.RegisterInterface(CorbaServer_i.SimpleText,
    CorbaServer_c.TSimpleTextHelper.RepositoryId);

  // Инициализация заглушки интерфейса CorbaServer_c.TSimpleTextStub
  // для CorbaServer_i.SimpleTextInterface.

  (* IDL Source     : "c:\icon99\MultiLanguage\MyProjects\CorbaServer\
                        SimpleText.idl", line 2
  ** IDL Name       : interface
  ** Repository Id  : IDL:CorbaServer/SimpleText:1.0
  ** IDL definition :
  *)
  CORBA.StubManager.RegisterStub(CorbaServer_i.SimpleText,
    CorbaServer_c.TSimpleTextStub);

end.
```

Нетрудно заметить, что код маршалинга, содержащийся внутри метода setText сгенерированного кода, слегка отличается от кода, который мы написали, чтобы выполнить ручной маршалинг того же самого интерфейса. Дело в том, что компилятор Idl2Pas использует другую DLL-библиотеку для обеспечения ORB/Pascal-доступа (OrbPas33.dll) и предоставляет два новых Pascal-модуля, которые поддерживают CORBA-структуру в среде Delphi (Corba.pas, OrbPas30.pas). Эти два новых дополнения будут мирно сосуществовать и не будут заменять модули и библиотеки, поставляемые в настоящее время с Delphi 5.

Данная версия компилятора Idl2Pas фирмы Inprise поможет упростить некоторые из наиболее сложных задач в архитектуре CORBA — например, вызов серверов, написанных на других языках, маршалинг нетривиальных типов данных и обработку исключительных ситуаций, определенных пользователем.

Развертывание ORB-брокера VisiBroker

Для использования ORB-брокера VisiBroker нужна специальная лицензия на эксплуатацию. И хотя вариант Delphi 5 Enterprise включает службы брокера VisiBroker в среду разработки, тем не менее, прежде чем по-настоящему развертывать свои программные решения, следует проконсультироваться на этот предмет в компании Inprise.

Службы ORB-брокера понадобятся для развертывания на компьютерах-серверах и компьютерах-клиентах. Как упоминалось выше, многие службы брокера VisiBroker (такие как osagent, irep и oad) могут работать в любом месте вашей локальной сети; следовательно, разворачивать эти службы необязательно на всех компьютерах, использующих программное обеспечение ORB-брокера. Уже отмечалось, что главным C++-брокером ORB, используемым с Delphi, является библиотека динамической компоновки orb_br.dll. Общая проблема, связанная с установками брокера в Windows, состоит в некорректном определении пути DOS. Это должно быть сделано для того, чтобы система могла находить, принадлежащие ORB-брокеру DLL-

библиотеки. Кроме того, не забывайте, что в Delphi используется специальный санкинг-слой (orbpas50.dll), предназначенный для отображения IDL-интерфейсов на Delphi-интерфейсы и обеспечения доступа к ORB-брокеру, созданному в среде C++. Поэтому библиотеку Orbpas50.dll также нужно устанавливать во всех CORBA-инсталляциях в среде Delphi 5.

Резюме

В этой главе мы рассмотрели основы разработки с использованием технологии CORBA в среде Delphi 5. Собственными силами мы создали и CORBA-клиент и CORBA-сервер, при этом были проведены эксперименты как с ранним, так и с поздним связыванием. Надеемся, вы поняли, что необходимо для реализации сценария раннего связывания с CORBA-сервером, написанным на другом языке. Наконец, мы кратко рассмотрели компилятор Idl2Pas фирмы Inprise и продемонстрировали, как предварительная версия этого инструмента может упростить CORBA-разработку в среде Delphi.

Работа с базами данных

ЧАСТЬ

IV

СОЗДАНИЕ ЛОКАЛЬНЫХ ПРИЛОЖЕНИЙ БАЗ ДАННЫХ 540

РАЗРАБОТКА ПРИЛОЖЕНИЙ АРХИТЕКТУРЫ КЛИЕНТ/СЕРВЕР 604

РАСШИРЕНИЯ БАЗ ДАННЫХ VCL 650

КОМПОНЕНТЫ WEBBROKER ОТКРЫВАЮТ ДВЕРИ В INTERNET 700

РАЗРАБОТКА ПРИЛОЖЕНИЙ MIDAS 732

Глава

28 Создание локальных приложений баз данных

Работа с наборами данных	541
Использование компонента `TTable`	568
Модули данных	574
Пример приложения с поиском, фильтрацией и выделением диапазона данных	575
Другие типы наборов данных: `TQuery` и `TStoredProc`	585
Таблицы в текстовом файле	586
Подключение с помощью ODBC	590
Объекты данных ActiveX (ADO)	596
Резюме	603

В этой главе рассматриваются вопросы доступа к файлам внешних баз данных из приложения Delphi. Если вы новичок в этой области программирования, вашему вниманию будет предложено немного теории, которая, надеемся, поможет вам найти свой путь к созданию высококачественных приложений для работы с базами данных. Если подобные приложения вам уже хорошо известны, вы получите полезную информацию о программировании баз данных именно в Delphi. Вначале речь пойдет о наборах данных и управлении ими, а затем будет рассмотрена конкретная работа с таблицами и запросами. По всей главе расставлены необходимые "дорожные указатели", помечающие места с информацией, знание которой повысит продуктивность разработки баз данных в Delphi.

Delphi 5 поставляется с версией 5.0 Borland Database Engine (BDE), которая предоставляет возможности унифицированного подключения к базам данных Paradox, dBASE, Access, FoxPro, ODBC, обычному ASCII-тексту и серверам баз данных SQL. В отличие от предыдущих версий, Delphi 5 Standard не поддерживает подключение к файлам баз данных. Только Delphi 5 Professional, помимо средств для подключения к источникам данных Local InterBase и ODBC, предоставляет возможность подключения к файлам баз данных Paradox, dBASE, Access, FoxPro и обычному ASCII-тексту. Версия Delphi 5 Enterprise построена на основе версии Professional и, дополнительно, содержит высокопроизводительные средства подключения к SQL-серверам InterBase, Microsoft SQL Server, Oracle, Informix Dynamic Server, Sybase Adaptive Server и DB2. В комплект поставки Delphi Enterprise входят также компоненты библиотеки ADOExpress, обеспечивающие простой доступ к источникам данных ADO. Темы, обсуждаемые в этой главе, в первую очередь относятся к использованию Delphi с такими данными в файлах, как таблицы Paradox и dBASE, хотя в ней затрагиваются также вопросы доступа к данным с помощью ODBC и ADO. Эта глава послужит основой для изучения материала следующей главы, "Разработка приложений архитектуры клиент/сервер".

Работа с наборами данных

Набор данных (dataset) — это коллекция строк и столбцов данных. Каждый *столбец* (column) является некоторым однородным типом данных, а каждая *строка* (row) представляет собой коллекцию данных каждого из типов данных столбцов. Столбец иногда называют *полем* (field), а строку *записью* (record). Библиотека VCL инкапсулирует набор данных в абстрактный компонент, называемый `TDataSet`, который предоставляет многие свойства и методы, необходимые для управления и перемещения по набору данных.

Чтобы облегчить понимание терминологии, ниже приводится список чаще всего встречающихся терминов из области баз данных, которые используются в этой и других главах, посвященных базам данных.

- *Набор данных* — это коллекция дискретных записей данных. Каждая запись создается множеством полей. Каждое поле может содержать различные типы данных (целые числа, строковые значения, десятичные числа, графику и т.д.). Наборы данных представлены в библиотеке VCL абстрактным классом `TDataSet`.
- *Таблица* — это специальный тип набора данных. Как правило, она представляет собой файл, содержащий записи и физически хранящийся где-нибудь на диске. В библиотеке VCL эту функциональность инкапсулирует класс `TTable`.
- *Запрос* — это специальный тип набора данных. Можно представлять себе запросы как "таблицы в памяти", которые сгенерированы с помощью специальных команд и предназначены для управления некоторыми физическими таблицами или наборами таблиц. Библиотека VCL для работы с запросами включает класс `TQuery`.

- *База данных* — это каталог на диске (если данные не размещены на сервере, например, как в случае файлов Paradox и dBASE) или SQL-база данных (если данные размещены на SQL-серверах). База данных может содержать множество таблиц. Как вы уже догадались, в библиотеке VCL есть соответствующий класс, и это — класс `TDatabase`.
- *Индекс* определяет правила упорядочения таблиц. Используя в качестве индекса отдельное поле в таблице, можно сортировать ее записи на основе значений, которые содержатся в этом поле для каждой записи. Компонент `TTable` содержит свойства и методы, которые необходимы для управления индексами.

На заметку

В начале главы упоминалось, что в ней будут затронуты некоторые вопросы теории баз данных. Однако эта глава не является начальным курсом программирования приложений для работы с базами данных. Поэтому предполагается, что все термины из приведенного выше списка вам уже известны. Если же вы встречаетесь с ними впервые, предварительно рекомендуем вам обратиться литературе, посвященной основам работы с базами данных.

Архитектура компонентов баз данных библиотеки VCL

При разработке Delphi 3 архитектура компонентов баз данных библиотеки VCL была значительно переработана для того, чтобы упростить использование *не* BDE-наборов данных в Delphi. Основой этой архитектуры является базовый класс `TDataSet`. Компонент `TDataSet` содержит абстрактное представление записей и полей набора данных. Некоторые методы класса `TDataSet` могут быть переопределены с целью создания компонента, подключаемого к определенному физическому формату данных. Исходя из этого, класс `TBDEDataSet` определен как производный от класса `TDataSet` и является основным классом источников данных, подключение к которым осуществляется с помощью BDE. Создание потомков класса `TDataSet` и подключение к некоторым типам пользовательских данных в пределах этой архитектурной схемы описывается в главе 30, “Расширения баз данных VCL”.

Компоненты доступа к данным BDE

Во вкладке Data Access палитры компонентов Component Palette содержатся компоненты библиотеки VCL, которые можно использовать для доступа и управления наборами данных BDE. Общий вид этой вкладки показан на рис. 28.1. В библиотеке VCL наборы данных представлены тремя классами: `TTable`, `TQuery` и `TStoredProc`. Все эти компоненты являются производными от класса `TDBDataSet`, который является производным от класса `TBDEDataSet` (а он, в свою очередь, является производным от класса `TDataSet`). Как уже отмечалось ранее в этой главе, класс `TDataSet` — это абстрактный класс, инкапсулирующий управление, навигацию и манипулирование набором данных. Класс `TBDEDataSet` — это также абстрактный класс, который представляет специфический BDE-набор данных. Класс `TDBDataSet` вводит такие концепции, как BDE-базы данных и сеансы (они будут более детально рассмотрены в следующей главе). Далее в этой главе упомянутый тип BDE-набора данных будет именоваться просто *набором данных*.

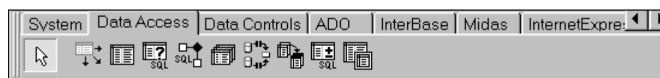


Рис. 28.1. Вкладка Data Access *в палитре компонентов*

Компонент TTable представляет структуру и данные, содержащиеся в таблице базы данных, а компонент TQuery — набор данных, содержащий информацию, возвращенную в результате выполнения SQL-запроса. Компонент TStoredProc инкапсулирует хранимые процедуры на SQL-сервере. В этой главе для ясности изложения компонент TTable будет использоваться всякий раз, когда речь пойдет о наборах данных. Компонент TQuery детально рассматривается несколько позже.

Открытие набора данных

Прежде чем выполнять манипуляции с набором данных, его необходимо открыть. Для этого достаточно вызвать метод Open(), как показано в приведенном ниже фрагменте кода:

```
Table1.Open;
```

Это, кстати, эквивалентно присвоению свойству Active набора данных значения True:

```
Table1.Active := True;
```

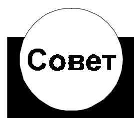
Второй способ является менее накладным, поскольку метод Open() в конечном счете также присваивает свойству Active значение True. Однако потери при этом настолько незначительны, что ни о чем не стоит беспокоиться.

После открытия набора данных, им можно свободно манипулировать. По завершении использования набора данных его необходимо закрыть, вызвав для этого метод Close():

```
Table1.Close;
```

Альтернативный способ закрытия — присвоение свойству Active значения False:

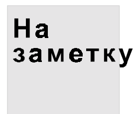
```
Table1.Active := False;
```



При взаимодействии с SQL-сервером соединение с базой данных должно быть установлено при первом открытии набора данных в этой базе. При закрытии последнего набора данных в базе, установленное с ней соединение будет закрыто. Открытие и закрытие подобных соединений создает значительную нагрузку на систему. Если открытие и закрытие соединений с базой данных требуется выполнять слишком часто, используйте компонент TDatabase вместо многократного выполнения операций открытия и закрытия соединения с базой данных SQL-сервера. Компонент TDatabase более детально рассматривается в следующей главе.

Навигация по набору данных

Компонент TDataSet предоставляет несколько простых методов работы с записями. Методы First() и Last() перемещают указатель текущей записи к первой и последней записям в наборе данных соответственно, а методы Next() и Prior() — на одну запись вперед или назад. Методу MoveBy() передается параметр Integer, в котором указывается, на какое количество записей следует переместить указатель вперед или назад.



Существует одно большое, но неочевидное преимущество использования BDE — возможность управлять SQL-таблицами и запросами. В общем случае по SQL-данным нельзя свободно перемещаться: можно перемещаться по строкам запроса вперед, но не назад. В отличие от ODBC, BDE позволяет свободно перемещаться по SQL-данным.

Свойства BOF, EOF и циклическая обработка

Свойства BOF и EOF класса TDataSet имеют тип Boolean и показывают, является ли текущая запись первой или последней в наборе данных. Например, пусть необходимо выполнить выборку каждой записи набора данных, вплоть до его последней записи. Эту простую задачу можно решить с помощью цикла while, в котором выборка записей будет продолжаться до тех пор, пока свойство EOF не примет значение True, как показано в приведенном ниже фрагменте кода:

```
Table1.First;            // Переход к началу набора данных
while not Table1.EOF do  // Перебор всех записей в таблице
  begin
                         // Выполнение обработки текущей записи
    Table1.Next;         // Перемещение к следующей записи
  end;
```

Вызывайте метод Next() только внутри цикла while-not-EOF, иначе приложение может попасть в бесконечный цикл.

Избегайте использования цикла repeat..until для выполнения действий над набором данных. Следующий код на первый взгляд выглядит вполне нормально, но если попытаться выполнить его с пустым набором данных, может случиться неприятность. Появление ошибки можно объяснить тем, что процедура DoSomeStuff() будет выполняться по крайней мере один раз, независимо от того, содержит или нет набор данных записи.

```
repeat
  DoSomeStuff;
  Table1.Next;
until Table1.EOF;
```

Поскольку цикл while-not-EOF вначале выполняет проверку условия, при использовании этой конструкции описанной выше проблемы удается избежать.

Закладки

Закладки (bookmarks) позволяют сохранить положение в наборе данных, чтобы позднее можно было вновь вернуться к этому же месту. В Delphi работать с закладками очень просто, поскольку необходимо запомнить значение всего одного свойства.

Delphi представляет закладу как тип данных TBookmarkStr. Класс TTable имеет свойство этого типа, называемое Bookmark. Если считать значение этого свойства, то будет получено имя закладки, а если же установить его, то будет выполнен переход к указанной закладке. Если какое-то место в наборе данных вызвало определенный интерес и к нему потребуется вернуться позднее, используйте следующий фрагмент кода:

```
var
  BM: TBookmarkStr;
begin
  BM := Table1.Bookmark;
```

Когда потребуется вернуться в выделенное место в наборе данных, сделать это будет очень просто — достаточно присвоить свойству Bookmark значение, полученное ранее при чтении свойства Bookmark:

```
Table1.Bookmark := BM;
```

Тип TBookmarkStr определен как AnsiString, поэтому память для закладки выделяется автоматически (вам не нужно заботится о ее освобождении). Если существующую закладку требуется удалить, то достаточно установить ее значение равным пустой строке:

```
BM := '';
```

Обратите внимание на то, что тип TBookmarkStr определяется как AnsiString. Его можно рассматривать как черный ящик, который не зависит от реализации, поскольку реальные данные закладки полностью определяются BDE и более низкими уровнями управления данными.

На заметку В 32-разрядной версии Delphi по-прежнему поддерживаются методы GetBookmark(), GotoBookmark() и FreeBookmark(), пришедшие из Delphi 1. Однако используйте их только в том случае, если необходимо сохранить совместимость с 16-разрядными проектами.

Пример навигации

Сейчас мы создадим небольшой проект, в котором продемонстрируем использование навигационных методов и других свойств компонента TDataSet, рассмотренных в предыдущем разделе. Назовем его Navig8. Главная форма этого проекта показана на рис. 28.2.

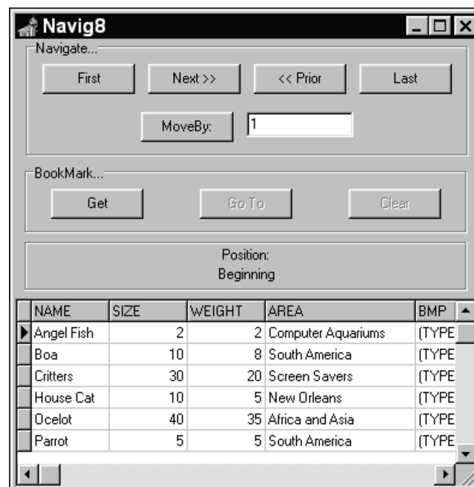
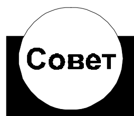


Рис. 28.2. Главная форма проекта Navig8

Для отображения данных, содержащихся в объекте TTable, в этом проекте используется компонент TDBGrid. Процесс создания элемента управления для работы с данными, такого как компонент TDBGrid, требует выполнения нескольких действий. Ниже приведена последовательность действий, которую необходимо выполнить для организации отображения данных объекта Table1 в объекте класса DBGrid1.

1. Назначьте свойству DatabaseName объекта Table1 существующий псевдоним или папку. Используйте псевдоним DBDEMOS, если вы установили примеры программ, поставляемых с Delphi.
2. Выберите таблицу из списка, представленного в свойстве TableName объекта Table1.
3. Переместите компонент TDataSource в форму и свяжите его с компонентом TTable, присвоив свойству DataSource1 компонента TDataSource значение Table1. Компонент TDataSource выступает в качестве передаточного звена между источником данных и элементами управления. Подробнее его работа обсуждается в следующей главе.
4. Свяжите компонент TDBGrid с компонентом TDataSource, установив свойство DataSource объекта DBGrid1 равным DataSource1.
5. Откройте таблицу, установив свойство Active объекта Table1 равным True.
6. Все, теперь данные таблицы находятся там, где следует, — в компоненте TDBGrid.



С целью быстрого выбора компонентов в раскрывающемся списке, который выводится для свойств DataSet и DataSource, дважды щелкните в окне Object Inspector — правее имени требуемого свойства. Это действие приведет к установке значения свойства равным первому элементу в раскрывающемся списке.

Исходный текст главного модуля приложения Navig8 (под именем Nav.pas) приведен в листинге 28.1.

Листинг 28.1. Исходный код модуля Nav.pas

```
unit Nav;

interface

uses
  SysUtils, Windows, Messages, Classes, Controls, Forms, StdCtrls,
  Grids, DBGrids, DB, DBTables, ExtCtrls;

type
  TForm1 = class(TForm)
    Table1: TTable;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    GroupBox1: TGroupBox;
    GetButton: TButton;
    GotoButton: TButton;
    ClearButton: TButton;
    GroupBox2: TGroupBox;
    FirstButton: TButton;
    LastButton: TButton;
    NextButton: TButton;
    PriorButton: TButton;
    MoveByButton: TButton;
    Edit1: TEdit;
    Panel1: TPanel;
```

```
    PosLbl: TLabel;
    Label1: TLabel;
    procedure FirstButtonClick(Sender: TObject);
    procedure LastButtonClick(Sender: TObject);
    procedure NextButtonClick(Sender: TObject);
    procedure PriorButtonClick(Sender: TObject);
    procedure MoveByButtonClick(Sender: TObject);
    procedure DataSource1DataChange(Sender: TObject; Field: TField);
    procedure GetButtonClick(Sender: TObject);
    procedure GotoButtonClick(Sender: TObject);
    procedure ClearButtonClick(Sender: TObject);
  private
     BM: TBookmarkStr;
  public
    { Открытые объявления }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FirstButtonClick(Sender: TObject);
begin
  Table1.First;   // Перейти к первой записи в таблице
end;

procedure TForm1.LastButtonClick(Sender: TObject);
begin
  Table1.Last;   // Перейти к последней записи в таблице
end;

procedure TForm1.NextButtonClick(Sender: TObject);
begin
  Table1.Next;   // Перейти к следующей записи в таблице
end;

procedure TForm1.PriorButtonClick(Sender: TObject);
begin
  Table1.Prior;   // Перейти к предыдущей записи в таблице
end;

procedure TForm1.MoveByButtonClick(Sender: TObject);
begin
  // Переместиться в таблице на установленное число записей вперед или назад
  Table1.MoveBy(StrToInt(Edit1.Text));
end;
```

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
begin
  //Установить заголовок в зависимости от состояния таблицы Table1, BOF/EOF
  if Table1.BOF then PosLbl.Caption := 'Beginning'
  else if Table1.EOF then PosLbl.Caption := 'End'
  else PosLbl.Caption := 'Somewheres in between';
end;

procedure TForm1.GetButtonClick(Sender: TObject);
begin
  BM := Table1.Bookmark;       // Получить закладку
  GotoButton.Enabled := True;  // Разрешить/запретить кнопки
  GetButton.Enabled := False;
  ClearButton.Enabled := True;
end;

procedure TForm1.GotoButtonClick(Sender: TObject);
begin
  Table1.Bookmark := BM;       // Перейти в позицию закладки
end;

procedure TForm1.ClearButtonClick(Sender: TObject);
begin
  BM := '';                     // Удалить закладку
  GotoButton.Enabled := False; // Разрешить/запретить соответствующие кнопки
  GetButton.Enabled := True;
  ClearButton.Enabled := False;
end;

end.
```

Приведенный пример отлично демонстрирует тот факт, что при использовании классов баз данных Delphi для манипулирования базой данных достаточно минимального объема программного текста.

Запомните, что при инициализации объектов GotoButton и FreeButton необходимо установить их свойство Enabled равным False, поскольку этими объектами нельзя пользоваться до тех пор, пока закладка не установлена. Методы FreeButtonClick() и GetButtonClick() гарантируют доступность соответствующих кнопок, в зависимости от того, установлены закладки или нет.

Многие из процедур в этом примере представляют собой всего одну строку, однако метод TForm1.DataSource1DataChange() требует некоторого пояснения. Этот метод обрабатывает событие OnDataChange объекта DataSource1, которое генерируется всякий раз при изменении значения любого поля (например, при перемещении от одной записи к другой). Это событие проверяет положение в наборе данных (начало, середина или конец) и соответствующим образом изменяет надписи в форме. События компонентов TTable и TDataSource рассматриваются далее в этой главе.

Свойства BOF и EOF

Вероятно, вы заметили, что при запуске проекта Navig8 надпись PosLbl показывает на позицию в начале набора данных, как оно и есть на самом деле. Однако, если перейти к следующей записи и вернуться назад, надпись PosLbl не отобразит тот факт, что в файле вновь выбрана первая запись. Если

щелкнуть на кнопке Prior еще раз, достижение начала файла все же будет распознано программой. Подобная ситуация возникает и в конце набора данных со свойством EOF. Почему, спросите вы?

Дело в том, что BDE не способен определить, где вы находитесь в данный момент — в начале или в конце набора данных, поскольку другой пользователь таблицы (если она многопользовательская) или даже другой процесс внутри вашей программы может добавить запись в начало или в конец таблицы, пока вы перемещались с первой записи на вторую, а затем обратно. Поэтому свойство BOF может принять значение True в одном из следующих случаев:

- если набор данных только что открыт;
- если только что был вызван метод First() набора данных;
- если вызов метода TDataSet.Prior() завершился неудачей — оказалось, что предыдущей записи нет.

Аналогично, свойство EOF может принять значение True в одном из следующих случаев:

- если открыт пустой набор данных;
- если только что вызван метод Last() набора данных;
- если вызов метода TDataSet.Next() завершился неудачей — оказалось, что следующей записи нет.

Из приведенного выше следует один важный вывод: набор данных пуст, если оба свойства — BOF и EOF — равны значению True.

Компонент TDataSource

Компонент TDataSource использовался в последнем примере, теперь самое время обсудить этот очень важный объект. Компонент TDataSource — это канал, с помощью которого компоненты доступа к данным (такие, как TTable) могут подключаться к элементам управления данными (например, к компонентам TDBEdit или TDBLookupCombo). Кроме предоставления интерфейса между наборами данных и элементами управления, необходимого для работы с данными, компонент TDataSource содержит несколько свойств и событий, упрощающих манипулирование данными.

Свойство State объекта TDataSource отображает текущее состояние связанного с ним набора данных. Значение свойства State может указывать, что набор данных в настоящее время неактивен или же находится в режиме Insert, Edit, SetKey или CalcFields. Свойство State объекта TDataSet более детально рассматривается ниже в этой главе. Событие OnStateChange генерируется всякий раз при изменении значения этого свойства.

Возникновение события OnDataChange компонента TDataSet показывает, что набор данных стал активным или элемент управления для работы с данными информирует набор данных о каком-то изменении.

Событие OnUpdateData возникает всякий раз, когда запись вводится или обновляется. Это событие, которое заставляет элементы управления для работы с данными изменять свое значение в соответствии с содержимым таблицы. Вы можете реагировать на это событие самостоятельно, отслеживая соответствующие изменения в самом приложении.

Работа с полями

Delphi позволяет получить доступ к полям любого набора данных с помощью класса TField и его потомков. Класс позволяет не только считать или установить значение выбранного поля текущей записи набора данных, но и изменить характеристики поля посредством

модификации его свойств. Кроме того, можно модифицировать набор данных в целом, изменяя визуальный порядок расположения полей, удаляя поля или же создавая новые вычисляемые или подстановочные поля.

Значения полей

Получить доступ к значениям полей в Delphi очень просто. Компонент TDataSet по умолчанию предлагает массив свойств с именем FieldValues[], который возвращает значение определенного поля как значение типа Variant. Поскольку массив FieldValues[] — это массив свойств по умолчанию, вам не нужно определять имя свойства для доступа к массиву. Например, в следующем фрагменте кода значение поля CustName таблицы Table1 присваивается переменной S типа String:

```
S := Table1['CustName'];
```

Так же просто можно присвоить значение поля CustNo целого типа переменной I:

```
I := Table1['CustNo'];
```

Мощным следствием из сказанного выше является возможность сохранения значений нескольких полей в массиве типа Variant. Единственным осложнением является то, что индекс массива типа Variant должен начинаться с нуля, а его содержимое представляет собой переменные varVariant. Приведенный ниже фрагмент кода демонстрирует эту возможность:

```
const
  AStr = 'The %s is of the %s category and its length is %f in.';
var
  VarArr: Variant;
  F: Double;
begin
  VarArr := VarArrayCreate([0, 2], varVariant);
  { Предполагаем, что объект Table1 связан с таблицей Biolife }
  VarArr := Table1['Common_Name;Category;Length_In'];
  F := VarArr[2];
  ShowMessage(Format(AStr, [VarArr[0], VarArr[1], F]));
end;
```

Программисты на Delphi 1 могут заметить, что работа с массивом свойств FieldValues[] намного проще, чем с прежними средствами доступа к значениям полей. По старой технологии (которая по-прежнему может использоваться и в среде Win32) для доступа к отдельному объекту TField, ассоциированному с набором данных, использовался массив свойств Fields[] объекта TDataSet или вызывалась функция FieldsByName(). Компонент TField содержал информацию о конкретном поле.

Массив Fields[] — это массив (с индексами, начинающимися с нуля) объектов класса TField. Так, элемент Fields[0] возвращает объект TField, представляющий первое логическое поле записи. Функции FieldsByName() передается строковый параметр, который соответствует некоторому имени поля в таблице. Таким образом, при вызове этой функции в виде FieldsByName('OrderNo') она должна возвратить компонент TField, представляющий поле OrderNo в текущей записи набора данных.

Получив объект TField, можно считать или присвоить значение полю с помощью одного из свойств этого объекта, приведенных в табл. 28.1.

Таблица 28.1. Свойства объекта TField для доступа к значениям полей

Свойство	Возвращаемый тип
AsBoolean	Boolean
AsFloat	Double
AsInteger	Longint
AsString	String
AsDateTime	TDateTime
Value	Variant

Если первое поле в текущем наборе данных — строковое, сохранить его значение в переменной S типа String можно следующим образом:

```
S := Table1.Fields[0].AsString;
```

В следующем фрагменте кода целой переменной I присваивается значение поля 'OrderNo' в текущей записи таблицы:

```
I := Table1.FieldsByName('OrderNo').AsInteger;
```

Типы полей данных

Если вы хотите узнать тип поля, используйте свойство DataType объекта TField, которое отображает тип данных таблицы базы данных (независимо от типа Object Pascal). Свойство DataType имеет тип TFieldType, который определяется следующим образом:

```
type
  TFieldType = (ftUnknown, ftString, ftSmallint, ftInteger, ftWord,
    ftBoolean, ftFloat, ftCurrency, ftBCD, ftDate, ftTime, ftDateTime,
    ftBytes, ftVarBytes, ftAutoInc, ftBlob, ftMemo, ftGraphic, ftFmtMemo,
    ftParadoxOle, ftDBaseOle, ftTypedBinary, ftCursor, ftFixedChar,
    ftWideString, ftLargeint, ftADT, ftArray, ftReference, ftDataSet,
    ftOraBlob, ftOraClob, ftVariant, ftInterface, ftDispatch, ftGuid);
```

Существуют классы, производные от класса TField и предназначенные специально для работы с упоминаемыми выше типами данных. Они рассматриваются далее в этой главе.

Имена и номера полей

Для поиска имени определенного поля используйте свойство FieldName класса TField. Например, в следующем фрагменте кода имя первого поля в текущей таблице записывается в переменную S типа String:

```
var
  S: String;
begin
  S := Table1.Fields[0].FieldName;
end;
```

Таким же образом, зная имя поля, можно получить его номер, используя свойство FieldNo. В следующем фрагменте кода номер поля OrderNo помещается в переменную I типа Integer:

```
var
  I: integer;
begin
  I := Table1.FieldsByName('OrderNo').FieldNo;
end;
```

На заметку

Для определения количества полей, содержащихся в наборе данных, используйте свойство FieldList объекта TDataSet. Свойство FieldList представляет собой линейное представление всех вложенных полей в таблице, содержащей поля, имеющие абстрактный тип данных (*abstract data type — ADT*).

Для обеспечения обратной совместимости свойство FieldCount также поддерживается в текущей версии Delphi, однако оно не поддерживает поля типа ADT, которые просто пропускаются.

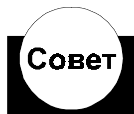
Манипулирование полем данных

Ниже приведена последовательность действий, которые необходимо выполнить для редактирования одного или нескольких полей в текущей записи.

1. Вызовите метод Edit() набора данных для переключения в режим редактирования (Edit).
2. Назначьте новые значения требуемым полям.
3. Внесите изменения в набор данных посредством вызова метода Post() или перемещения на новую запись, в результате чего изменения будут внесены автоматически.

Например, типичная процедура изменения записи выглядит следующим образом:

```
Table1.Edit;
Table1['Age'] := 23;
Table1.Post;
```



Иногда приходится работать с наборами данных, доступными только для чтения. Примером таких данных может служить информация на компакт-диске или запрос с автономным результирующим набором, не доступным для редактирования. Прежде чем предпринимать попытки редактирования данных, следует проверить, не содержит ли текущий набор данные, доступные только для чтения. Для этого достаточно проверить значение свойства CanModify. Если возвращаемое значение свойства CanModify равно True, редактирование набора данных разрешается.

Как и в случае редактирования, вставлять или добавлять записи в конец набора данных можно следующим образом.

1. Вызовите метод Insert() или Append() набора данных для переключения в режим вставки (Insert) или добавления (Append).
2. Присвойте значения полям новой записи набора данных.
3. Внесите новую запись в набор данных посредством вызова метода Post() или перемещения на новую запись, в результате чего изменения будут внесены автоматически.

На заметку

Когда вы находитесь в одном из режимов — Edit, Insert или Append (редактирования, вставки или дополнения), — помните, что ваши изменения будут безусловно внесены при перемещении от текущей записи к следующей. Поэтому будьте внимательны при использовании методов `Next()`, `Prior()`, `First()`, `Last()` и `MoveBy()` в процедурах редактирования записей.

Если потребуется отменить внесенные в запись, но еще не зафиксированные в наборе данных изменения, это можно сделать посредством вызова метода `Cancel()`. Например, следующий фрагмент кода отменяет выполненное редактирование до внесения изменений в таблицу:

```
Table1.Edit;
Table1['Age'] := 23;
Table1.Cancel;
```

Метод `Cancel()` отменяет внесенные, но не зафиксированные изменения и возвращает набор данных из режима Edit, Append или Insert в режим Browse.

Завершим описание методов, манипулирующих записью компонента `TDataSet`, методом `Delete()`, который удаляет текущую запись из набора данных. Например, в следующем фрагменте кода удаляется последняя запись в таблице:

```
Table1.Last;
Table1.Delete;
```

Редактор полей

В Delphi существует утилита Fields Editor (Редактор полей), предоставляющая возможность более гибкого управления полями набора данных. Этот инструмент можно использовать для просмотра отдельного набора данных в окне конструктора форм, для чего следует дважды щелкнуть на компоненте `TTable`, `TQuery` или `TStoredProc` либо выбрать команду Fields Editor в контекстном меню для набора данных. В окне редактора полей можно выбрать поля набора данных, с которыми требуется работать, или же создать новые вычисляемые либо подстановочные поля. Для этого используйте контекстное меню редактора. На рис. 28.3 показано окно редактора полей с раскрытым контекстным меню.

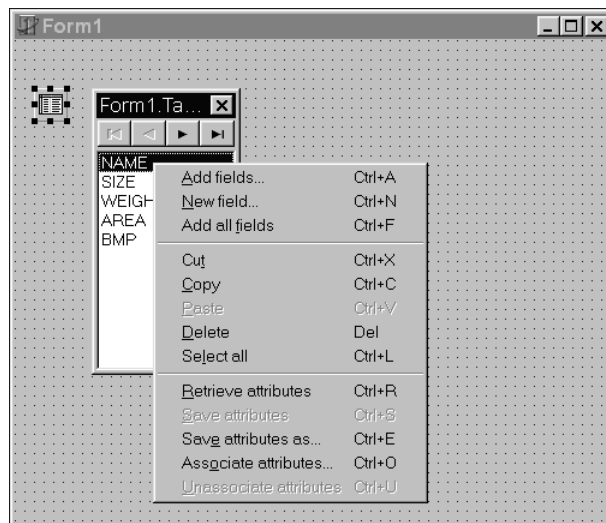


Рис. 28.3. Контекстное меню окна редактора полей (Fields Editor)

Для демонстрации методов использования редактора полей откройте новый проект и поместите в главную форму компонент TTable. Установите свойство DatabaseName объекта Table1 равным DBDEMOS (это псевдоним, который заставляет Delphi использовать таблицы примеров), а свойство TableName — равным ORDERS.DB. Поместите в форму компоненты TDataSource и TDBGrid. Свяжите объект DataSource1 с объектом Table1, а объект DBGrid1 — с объектом DataSource1. Теперь установите свойство Active объекта Table1 в True, и вы увидите данные объекта Table1.

Добавление полей

Откройте окно редактора полей (см. рис. 28.3), дважды щелкнув на объекте Table1. Предположим, что требуется ограничить представление таблицы несколькими полями. В контекстном меню окна редактора полей выберите команду Add Fields (Добавить поля). На экране раскроется диалоговое окно Add Fields. Выделите в списке Available fields поля OrderNo, CustNo и ItemsTotal, а затем щелкните на кнопке OK. Эти три выделенных поля появятся в окне редактора полей и в сетке с данными.

Для представления полей набора данных, выделенных в окне редактора полей, Delphi создает объекты, производные от объекта TField. Например, для выбранных только что трех полей таблицы ORDERS.DB Delphi помещает в исходный текст формы следующие объявления:

```
Table1OrderNo: TFloatField;
Table1CustNo: TFloatField;
Table1ItemsTotal: TCurrencyField;
```

Заметьте, что имя объекта поля — это соединение имени TTable и имени поля. Поскольку эти объекты полей создаются программно, можно получить доступ ко всем унаследованным ими от класса TField свойствам и методам непосредственно во время выполнения программы, а не только во время проектирования.

Потомки компонента TField

Давайте отвлечемся на минуту от рассмотрения компонентов TField. Для каждого типа поля (типы полей описывались выше, в разделе “Типы полей данных”) существует один или несколько различных классов, производных от класса TField. Многие из этих типов полей также соответствуют типам данных, существующим в языке Object Pascal. В табл. 28.2 приведены сведения о различных классах иерархии TField, типах их родительских классов, типах данных их полей и эквивалентных типах данных Object Pascal.

Таблица 28.2. Потомки класса TField и типы их полей

Производный класс	Класс-предок	Тип поля	Тип Object Pascal
TStringField	TField	ftString	String
TWideStringField	TStringField	ftWideString	WideString
TGuidField	TStringField	ftGuid	TGUID
TNumericField	TField	*	*
TIntegerField	TNumericField	ftInteger	Integer
TSmallIntField	TIntegerField	ftSmallInt	SmallInt
TLargeintField	TNumericField	ftLargeint	Int64

Окончание табл. 28.2

Производный класс	Класс-предок	Тип поля	Тип Object Pascal
TWordField	TIntegerField	ftWord	Word
TAutoIncField	TIntegerField	ftAutoInc	Integer
TFloatField	TNumericField	ftFloat	Double
TCurrencyField	TFloatField	ftCurrency	Currency
TBCDField	TNumericField	ftBCD	Double
TBooleanField	TField	ftBoolean	Boolean
TDateTimeField	TField	ftDateTime	TDateTime
TDateField	TDateTimeField	ftDate	TDateTime
TTimeField	TDateTimeField	ftTime	TDateTime
TBinaryField	TField	*	*
TBytesField	TBinaryField	ftBytes	Нет
TVarBytesField	TBytesField	ftVarBytes	Нет
TBlobField	TField	ftBlob	Нет
TMemoField	TBlobField	ftMemo	Нет
TGraphicField	TBlobField	ftGraphic	Нет
TObjectField	TField	*	*
TADTField	TObjectField	ftADT	Нет
TArrayField	TObjectField	ftArray	Нет
TDataSetField	TObjectField	ftDataSet	TDataSet
TReferenceField	TDataSetField	ftReference	
TVariantField	TField	ftVariant	OleVariant
TInterfaceField	TField	ftInterface	IUnknown
TIDispatchField	TInterfaceField	ftIDispatch	IDispatch
TAggregateField	TField	Нет	Нет

* Означает абстрактный базовый класс в иерархии

Как видно из табл. 28.2, типы полей BLOB и Object — это специальные типы, которые не имеют прямого аналога среди типов Object Pascal. Поля типа BLOB будут подробно рассматриваться далее в этой главе.

Поля и инспектор объектов

Если выделить поле в окне редактора полей, в окне инспектора объектов (Object Inspector) можно будет получить доступ к свойствам и событиям, ассоциированным с данным потомком объекта TField. Это позволяет модифицировать свойства полей (например, определять минимальное и максимальное значение, формат отображения, а также делать их доступными только для чтения). Назначение одних свойств (таких как ReadOnly — только чтение) очевидно из их названия, а назначение других может быть не совсем понятно. Некоторые из этих "интуитивно непонятных" свойств будут рассматриваться ниже в этой главе. На рис. 28.4 показаны свойства поля OrderNo, отображаемые в окне инспектора объектов.

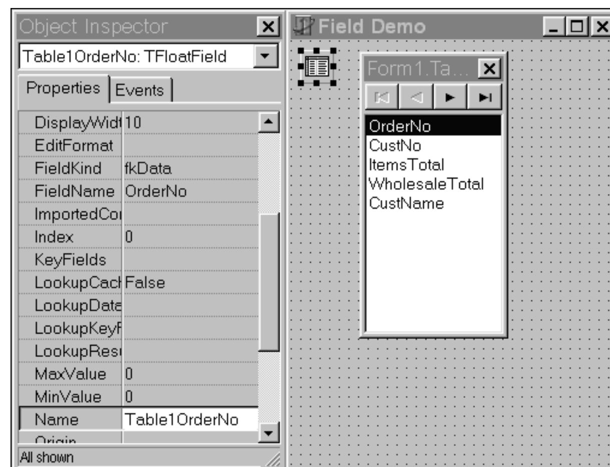


Рис. 28.4. Редактирование свойства поля

Открыв в окне инспектора объектов вкладку Events, можно увидеть, что с объектами поля ассоциированы и некоторые события. События `OnChange`, `OnGetText`, `OnSetText` и `OnValidate` подробно описаны в интерактивной справочной системе. Для получения справки по событию достаточно щелкнуть слева от его имени, а затем нажать клавишу <F1>. Из всех событий чаще всего используется, пожалуй, `OnChange`. Оно позволяет выполнять некоторые действия при каждом изменении содержимого поля (например, переходить на другую запись или добавлять новую запись).

Вычисляемые поля

Помимо прочего, в окне редактора полей к набору данных можно добавить вычисляемые поля. Например, допустим, что в набор данных необходимо добавить поле, отображающее для каждой строки в таблице `ORDERS` объем оптовой продажи, составляющий 32% от общего объема. Выберите в контекстном меню окна редактора полей команду New Field. На экране раскроется диалоговое окно New Field, показанное на рис. 28.5. В поле Name этого окна введите имя нового поля — **`WholesaleTotal`**. Тип этого поля — **`Currency`**, поэтому в раскрывающемся списке Type выберите именно это значение. В группе Field Type установите переключатель в положение Calculated и щелкните на кнопке OK. Новое поле появится в сетке, но пока не будет содержать никаких данных.

Чтобы заполнить новое поле данными, необходимо назначить требуемый метод событию `OnCalcFields` объекта `Table1`. В тексте обработчика этого события полю `WholesaleTotal` следует просто присвоить значение, равное 32% от существующего значения поля `SalesTotal`. Соответствующий текст метода обработки события `Table1.OnCalcFields` показан ниже:

```
procedure TForm1.Table1CalcFields(DataSet: TDataSet);
begin
  DataSet['WholesaleTotal'] := DataSet['ItemsTotal'] * 0.32;
end;
```

На рис. 28.6 показано поле `WholesaleTotal`, расположенное в сетке и содержащее правильные данные.

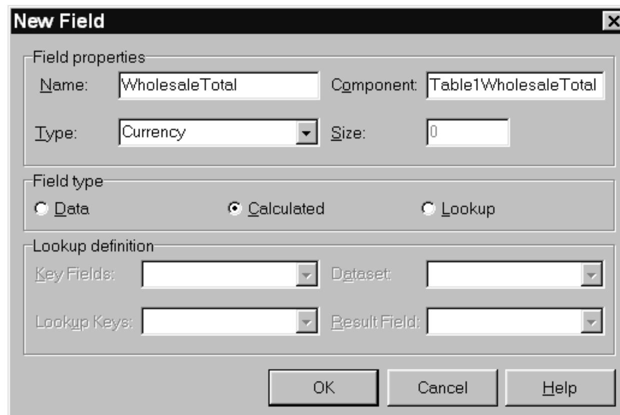


Рис. 28.5. Добавление вычисляемого поля в диалоговом окне New Field

Field Demo

OrderNo	CustNo	ItemsTotal	WholesaleTotal	CustName
#1003	CN 1351	1 250,00	850,00	Phyllis Spooner
#1004	CN 2156	7 885,00	5 361,80	Tanya Wagner
#1005	CN 1356	4 807,00	3 268,76	Chris Thomas
#1006	CN 1380	31 987,00	21 751,16	Ernest Barratt
#1007	CN 1384	6 500,00	4 420,00	Russell Christo
#1008	CN 1510	1 449,50	985,66	Paul Gardner
#1009	CN 1513	5 587,00	3 799,16	Susan Wong
#1010	CN 1551	4 996,00	3 397,28	Joyce Marsh
#1011	CN 1560	2 679,85	1 822,30	Sam Witherspoon
#1012	CN 1563	5 201,00	3 536,68	Theresa Kunec
#1013	CN 1624	3 115,00	2 118,20	Donna Siaus
#1014	CN 1645	134,85	91,70	Michael Spurlin

Рис. 28.6. Вычисляемое поле добавлено к таблице

Подстановочные поля

Подстановочные (lookup) поля позволяют создавать в наборе данных такие поля, значения которых будут выбираться из другого набора данных. Для иллюстрации сказанного добавим такое поле к текущему проекту. Вряд ли по номеру клиента (поле `CustNo` таблицы `ORDERS`) можно будет вспомнить его имя. Поэтому целесообразно добавить к таблице `Table1` подстановочное поле, связанное с таблицей `CUSTOMER`, из которой по номеру клиента будет выбираться его имя.

Вначале поместите в форму второй объект `TTable` и присвойте его свойству `DatabaseName` значение `DBDEMOS`, а свойству `TableName` — значение `CUSTOMER.DB`. Это будет объект `Table2`. Затем вновь откройте окно New Field, для чего выберите команду New Field в контекстном меню окна редактора поля. Присвойте новому полю имя **`CustName`** и тип **`String`**. Размер поля установите равным 15 символам. Не забудьте установить переключатель в группе Field Type в положение Lookup. В списке Dataset этого диалогового окна выберите значение `Table2` — именно этот набор данных необходимо просматривать. В обоих списках Key Fields и Lookup Keys этого диалогового

окна выберите значение `CustNo` — это то общее поле, по значению которого будет выполняться поиск. И, наконец, в списке Result выберите значение `Contact` — именно это поле необходимо отображать в нашем наборе данных. На рис. 28.7 показано диалоговое окно New Field в процессе создания подстановочного поля, а на рис. 28.8 — готовая форма с подстановочными данными.

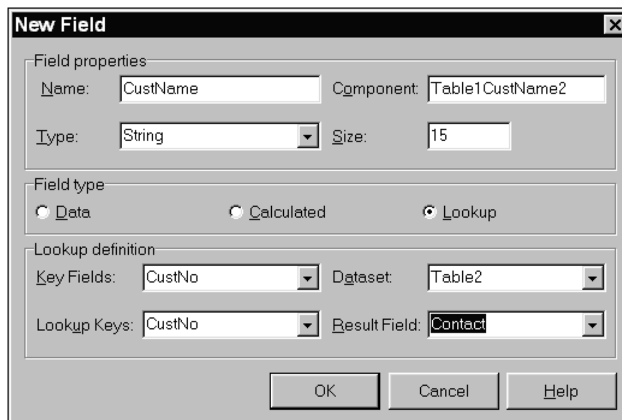
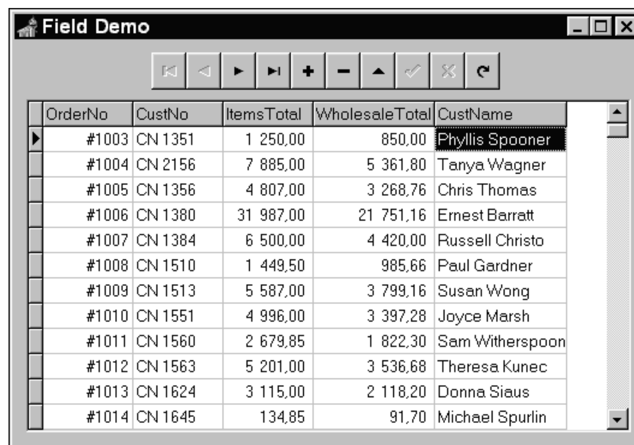


Рис. 28.7. Добавление подстановочного поля в диалоговом окне New Filed



OrderNo	CustNo	ItemsTotal	WholesaleTotal	CustName
#1003	CN 1351	1 250,00	850,00	Phyllis Spooner
#1004	CN 2156	7 885,00	5 361,80	Tanya Wagner
#1005	CN 1356	4 807,00	3 268,76	Chris Thomas
#1006	CN 1380	31 987,00	21 751,16	Ernest Barratt
#1007	CN 1384	6 500,00	4 420,00	Russell Christo
#1008	CN 1510	1 449,50	985,66	Paul Gardner
#1009	CN 1513	5 587,00	3 799,16	Susan Wong
#1010	CN 1551	4 996,00	3 397,28	Joyce Marsh
#1011	CN 1560	2 679,85	1 822,30	Sam Witherspoon
#1012	CN 1563	5 201,00	3 536,68	Theresa Kunec
#1013	CN 1624	3 115,00	2 118,20	Donna Siaus
#1014	CN 1645	134,85	91,70	Michael Spurlin

Рис. 28.8. Форма, содержащая подстановочное поле

Перетаскивание полей мышью

Другое, менее очевидное свойство окна редактора полей — возможность перетаскивать поля из списка полей в создаваемую форму. Это свойство можно продемонстрировать следующим образом: создайте новый проект, который будет содержать в главной форме только объект `TTable`. Установите свойство `Table1.DatabaseName` равным `DBDEMOS`, а свойство `Table1.TableName` — равным `BIOLIFE.DB`. Откройте для этой таблицы окно редактора полей и добавьте все поля таблицы в список полей набора данных. Теперь появилась возможность перетащить одно или несколько полей из окна редактора полей в главную форму.

Отметим несколько особенно впечатляющих особенностей этой процедуры. Во-первых, Delphi распознает, тип помещаемого в форму поля и создает соответствующий элемент управления для отображения его данных (например, для строкового поля создается объект класса

TDBEdit, а для графического поля — объект класса TDBImage). Во-вторых, Delphi проверяет, существует ли объект класса TDataSource, связанный с этим набором данных. Если это так, именно он ставится в соответствие новому полю, в противном случае объект создается заново. На рис. 28.9 показан результат перетаскивания с помощью мыши полей таблицы BIOLIFE в форму.

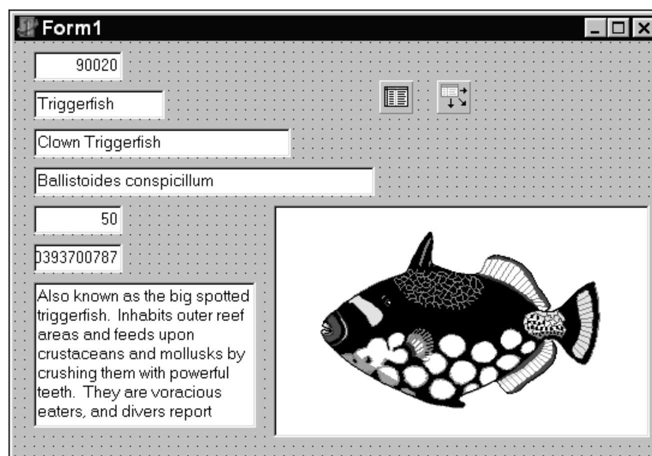


Рис. 28.9. Помещение полей в форму

Работа с BLOB-полями

Поля BLOB (Binary Large Object) разработаны для размещения в них неопределенного количества данных. BLOB-поля в одной записи набора данных могут содержать три байта данных, в то время как подобное поле в другой записи может содержать 3 Кбайт данных. Эти поля наиболее удобны для хранения большого количества текста, графики или потоков структурированных данных, таких как OLE-объекты.

Класс TBlobField и типы поля

Как уже отмечалось выше в этой главе, в библиотеке VCL существует класс TBlobField, производный от класса TField и специально предназначенный для инкапсуляции BLOB-полей. Класс TBlobField имеет свойство BlobType типа TBlobType, которое отображает, какой именно тип данных хранится в его BLOB-поле. Тип TBlobType определяется в модуле DB следующим образом:

```
TBlobType = ftBlob..ftOraClob;
```

Все возможные типы BLOB-поля и ассоциированные с ними типы данных приведены в табл. 28.3.

Таблица 28.3. Типы полей класса TBlobField

Тип поля	Тип данных
ftBlob	Нетипизированные или определенные пользователем данные
ftMemo	Текст
ftGraphic	Растровая графика Windows
ftFmtMemo	Форматированное поле типа memo Paradox

Окончание табл. 28.3

Тип поля	Тип данных
ftParadoxOle	OLE-объект Paradox
ftDBaseOLE	OLE-объект dBASE
ftTypedBinary	Неструктурированные данные, представляющие существующий тип
ftCursor..ftDataSet	Недопустимые BLOB-поля
ftOraBlob	BLOB-поля в таблицах Oracle8
ftOraClob	CLOB-поля в таблицах Oracle8

Как правило, основная часть работы по выборке и помещению данных в компонент класса TBlobField может быть выполнена посредством их загрузки или сохранения в файле или с помощью компонента TBlobStream. Класс TBlobStream — это специализированный класс, производный от класса TStream, который использует BLOB-поле внутри физической таблицы как место размещения потока. Для демонстрации этих методов работы с компонентом TBlobField создадим приложение-пример.

На заметку

При запуске программы Setup на прилагаемом компакт-диске будет создан псевдоним BDE, указывающий на подкаталог \Data в каталоге с программным обеспечением. В этом каталоге вы найдете все таблицы, используемые в примерах этой книги. В некоторых примерах, содержащихся на компакт-диске, предполагается наличие псевдонима DDGData.

Пример использования BLOB-поля

В описанном ниже проекте создается приложение, которое позволяет пользователю сохранять .wav-файлы в таблице базы данных, а затем воспроизводить их непосредственно из этой таблицы. Начните проект с создания главной формы с компонентами, показанными на рис. 28.10. Компонент TTable может описывать таблицу Wavez, расположенную в соответствии с псевдонимом DDGUtils, или вашу таблицу с подобной же структурой. Структура таблицы Wavez приведена ниже.

Имя поля	Тип поля	Размер
WaveTitle	Character	25
FileName	Character	25
Wave	BLOB	

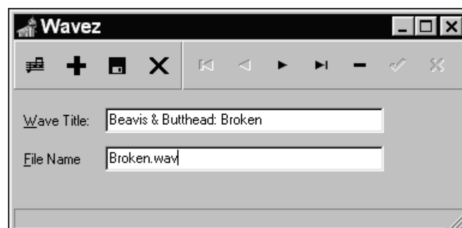


Рис. 28.10. Главная форма для таблицы Wavez — примера использования BLOB-поля

Кнопка Add используется для загрузки .wav-файла с диска и добавления его в таблицу. Подпрограмма обработки события OnClick кнопки Add, имеет следующий вид:

```
procedure TMainForm.sbAddClick(Sender: TObject);
begin
  if OpenDialog.Execute then
  begin
    tblSounds.Append;
    tblSounds['FileName'] := ExtractFileName(OpenDialog.FileName);
    tblSoundsWave.LoadFromFile(OpenDialog.FileName);
    edTitle.SetFocus;
  end;
end;
```

В этой процедуре сначала предпринимается попытка выполнить метод OpenDialog. Если эта операция выполняется успешно, источник данных tblSound переключается в режим Append, полю FileName присваивается значение и BLOB-поле Wave заполняется данными из файла, определенного параметром OpenDialog. Обратите внимание, насколько удобно использовать здесь метод LoadFromFile класса TBlobField и насколько ясно и просто выглядит код загрузки файла в BLOB-поле.

Подобным же образом щелчок на кнопке Save приводит к сохранению во внешнем файле текста звукового файла, расположенного в поле Wave. Код процедуры для этой кнопки выглядит следующим образом:

```
procedure TMainForm.sbSaveClick(Sender: TObject);
begin
  with SaveDialog do
  begin
    FileName := tblSounds['FileName']; // Инициализация имени файла
    if Execute then                    // Открытие диалогового окна
      tblSoundsWave.SaveToFile(FileName);// Сохранение BLOB-поля в файле
  end;
end;
```

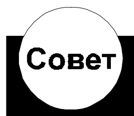
В этом методе используется еще меньше кода. Объект SaveDialog инициализируется значением поля FileName. Если выполнение процедуры SaveDialog выполняется успешно, для сохранения содержимого BLOB-поля в файле вызывается метод SaveToFile объекта tblSoundsWave.

Обработчик кнопки Play считывает .wav-данные из BLOB-поля и передает их функции Win32 API PlaySound() для воспроизведения. Код этого обработчика приведен ниже. Обратите внимание на то, что этот код немного сложнее приведенного выше кода для кнопки Save.

```
procedure TMainForm.sbPlayClick(Sender: TObject);
var
  B: TBlobStream;
  M: TMemoryStream;
begin
  B := TBlobStream.Create(tblSoundsWave, bmRead);   // Создание BLOB-потока
  Screen.Cursor := crHourGlass;  // Отображение песочных часов
  try
```

```
    M := TMemoryStream.Create; // Создание потока памяти
    try
      M.CopyFrom(B, B.Size); // Копирование из BLOB-потока в поток памяти
      // Попытка воспроизвести звук
      // Генерация исключительной ситуации при неправильном выполнении.
      Win32Check(PlaySound(M.Memory, 0, SND_SYNC or SND_MEMORY));
    finally
      M.Free;
    end;
  finally
    Screen.Cursor := crDefault;
      B.Free;  // Очистка
  end;
end;
```

Сначала этот метод создает экземпляр класса TBlobStream с именем B, используя BLOB-поле таблицы tblSoundsWave. Первый параметр, передаваемый методу TBlobStream.Create(), является объектом BLOB-поля, а второй параметр указывает, как следует открыть поток. Для доступа к BLOB-потоку "только для чтения" используется значение bmRead, а при доступе "для чтения и записи" — значение bmReadWrite.



Для открытия потока TBlobStream с параметром bmReadWrite используемый набор данных должен быть переведен в режим *Edit*, *Insert* или *Append*.

Затем создается экземпляр M класса потока TMemoryStream. В этот момент обычный указатель мыши изменяется на изображение песочных часов, что указывает пользователю на продолжительность выполняемой операции. Затем поток B копируется в поток M. Функции PlaySound(), используемой для воспроизведения звукового файла, в качестве первого параметра должно указываться имя файла или указатель памяти. Класс TBlobStream не предоставляет доступа к данным потока с помощью указателя, а класс TMemoryStream поддерживает эту возможность с помощью свойства Memory. Воспользовавшись этим, можно вызвать функцию PlaySound() для воспроизведения данных, указатель на которые содержится в свойстве M.Memory. После завершения работы этой функции необходимо освободить потоки и восстановить вид указателя мыши. Полный текст основного модуля этого проекта приведен в листинге 28.2.

Листинг 28.2. Основной модуль проекта Wavez

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, DBCtrls, DB, DBTables, StdCtrls, Mask, Buttons, ComCtrls;

type
  TMainForm = class(TForm)
    tblSounds: TTable;
```

```
    dsSounds: TDataSource;
    tblSoundsWaveTitle: TStringField;
    tblSoundsWave: TBlobField;
    edTitle: TDBEdit;
    edFileName: TDBEdit;
    Label1: TLabel;
    Label2: TLabel;
    OpenDialog: TOpenDialog;
    tblSoundsFileName: TStringField;
    SaveDialog: TSaveDialog;
    pnlToobar: TPanel;
    sbPlay: TSpeedButton;
    sbAdd: TSpeedButton;
    sbSave: TSpeedButton;
    sbExit: TSpeedButton;
    Bevel1: TBevel;
    dbnNavigator: TDBNavigator;
    stbStatus: TStatusBar;
    procedure sbPlayClick(Sender: TObject);
    procedure sbAddClick(Sender: TObject);
    procedure sbSaveClick(Sender: TObject);
    procedure sbExitClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    procedure OnAppHint(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses MMSystem;

procedure TMainForm.sbPlayClick(Sender: TObject);
var
  B: TBlobStream;
  M: TMemoryStream;
begin
  B := TBlobStream.Create(tblSoundsWave, bmRead);  // Создание blob-потока
  Screen.Cursor := crHourGlass;  // Отображение песочных часов
  try
    M := TMemoryStream.Create; // Создание потока в памяти
    try
      M.CopyFrom(B, B.Size); // Копирование из blob-потока в поток памяти
      // Попытка воспроизвести звук.
```

```
      // Генерирование исключительной ситуации при неправильном выполнении.
      Win32Check(PlaySound(M.Memory, 0, SND_SYNC or SND_MEMORY));
    finally
      M.Free;
    end;
  finally
    Screen.Cursor := crDefault;
    B.Free;   // Очистка
  end;
end;

procedure TMainForm.sbAddClick(Sender: TObject);
begin
  if OpenDialog.Execute then
  begin
    tblSounds.Append;
    tblSounds['FileName'] := ExtractFileName(OpenDialog.FileName);
    tblSoundsWave.LoadFromFile(OpenDialog.FileName);
    edTitle.SetFocus;
  end;
end;

procedure TMainForm.sbSaveClick(Sender: TObject);
begin
  with SaveDialog do
  begin
    FileName := tblSounds['FileName'];  // Инициализация имени файла
    if Execute then              // Отображение диалогового окна
      tblSoundsWave.SaveToFile(FileName);   // Сохранение blob-поля в файле
  end;
end;

procedure TMainForm.sbExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.OnHint := OnAppHint;
end;

procedure TMainForm.OnAppHint(Sender: TObject);
begin
  stbStatus.SimpleText := Application.Hint;
end;

end.
```

Обновление набора данных

При разработке приложения необходимо учитывать, что набор данных, с которым вы работаете, находится в постоянном движении. Его записи постоянно добавляются, удаляются или модифицируются, особенно при работе в локальной сети. Поэтому изредка необходимо перечитывать информацию о наборе данных с диска или из памяти для обновления содержимого набора данных, отображаемого в формах.

Обновить набор данных можно с помощью метода `Refresh()` компонента `TDataSet`. Функционально он напоминает последовательный вызов для набора данных методов `Close()` и `Open()`, но выполняется несколько быстрее. Метод `Refresh()` может успешно применяться ко всем локальным таблицам, однако при работе с базой данных на сервере SQL все же имеются некоторые ограничения.

Компонент `TTable`, подключенный к базе данных SQL, должен использовать уникальный индекс, и только в этом случае BDE может предпринять попытку выполнить операцию `Refresh()`. Суть в том, что метод `Refresh()` всегда пытается сохранить текущую запись (если это возможно). Это означает, что для перемещения к текущей (на данный момент) записи BDE вынужден использовать метод `Seek()`. Однако это возможно лишь в том случае, если для набора данных SQL существует уникальный индекс. Метод `Refresh()` нельзя использовать для компонентов `TQuery`, связанных с базой данных SQL.

При вызове в программе метода `Refresh()` работающий с ней пользователь может столкнуться с некоторыми неожиданными эффектами. Например, если первый пользователь просматривает запись в сетевой таблице, и в этот момент данная запись будет удалена вторым пользователем, вызов метода `Refresh()` приведет к тому, что на мониторе первого пользователя просматриваемая им запись исчезнет без всяких видимых причин. При вызове этой функции следует помнить, что данные могут изменяться и без участия данного пользователя.

Изменение состояния набора данных

Иногда бывает необходимо уточнить, в каком режиме находится таблица (Edit или Append) и активна ли она вообще. Получить эту информацию можно с помощью свойства `State` компонента `TDataSet`. Свойство `State` имеет тип `TDataSetState`, а его значения приведены в табл. 28.4.

Таблица 28.4. Возможные значения свойства TDataSet.State

Значение	Описание
`dsBrowse`	Набор данных находится в режиме Browse (обычный режим просмотра данных)
`dsCalcFields`	Вызван обработчик события `OnCalcFields`, и значение полей записи в настоящий момент пересчитывается
`dsEdit`	Набор данных находится в режиме Edit (Редактирование). Это означает, что был вызван метод `Edit()`, но отредактированная запись еще не была внесена в таблицу
`dsInactive`	Набор данных закрыт
`dsInsert`	Набор данных находится в режиме Insert (Вставка). Обычно это означает, что был вызван метод `Insert()`, но вставляемая запись еще не была внесена в таблицу

Окончание табл. 28.4

Значение	Описание
dsSetKey	Набор данных находится в режиме SetKey (Задание ключа). Был вызван метод SetKey(), но метод GotoKey() еще не вызывался
dsNewValue	Набор данных находится во временном состоянии, когда осуществляется доступ к свойству NewValue
dsOldValue	Набор данных находится во временном состоянии, когда осуществляется доступ к свойству OldValue
dsCurValue	Набор данных находится во временном состоянии, когда осуществляется доступ к свойству CurValue
dsFilter	В наборе данных в настоящее время выполняется фильтрация записей, выбор подстановочных значений или другая операция с использованием фильтра
dsBlockRead	Набор данных буферизируется, поэтому при установке этого значения перемещение курсора не приводит к обновлению данных в элементах управления и генерации событий
dsInternalCalc	В настоящее время вычисляется значение поля, у которого свойство FieldKind имеет значение fkInternalCalc
dsOpening	Набор данных находится в состоянии открытия, которое в настоящей момент еще не завершено. Это состояние устанавливается только тогда, когда набор данных открывается для асинхронной выборки данных

Фильтры

Фильтры позволяют упростить поиск или отбор записей в наборе данных, выполняемые в программах Object Pascal. Основное преимущество использования фильтров состоит в том, что они не используют индекс и не нуждаются в каких-либо иных подготовительных действиях над набором данных, с которым работают. Чаще всего фильтрация выполняется несколько медленнее, чем поиск, основанный на индексе (речь об этом пойдет ниже, в этой же главе), тем не менее они могут эффективно использоваться практически во всех типах приложений.

Фильтрация набора данных

Один из наиболее общих механизмов фильтрации, используемых в Delphi, — ограничение представления набора данных до некоторых указанных записей. Этот процесс выполняется следующим образом.

1. Назначьте процедуру событию OnFilterRecord набора данных. В эту процедуру следует поместить операторы, выполняющие отбор записей на основе значений одного или нескольких полей.
2. Установите свойство Filtered набора данных равным True.

На рис. 28.11 показана форма, содержащая объект TDBGrid, в котором отображены неотфильтрованные данные таблицы CUSTOMER.

На первом этапе создадим обработчик события `OnFilterRecord` для этой таблицы. В данном случае будут отбираться только те записи, в которых значение поле `Company` начинается с прописной буквы *S*. Текст этой процедуры выглядит следующим образом:

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet; var Accept: Boolean);
var
  FieldVal: String;
begin
  FieldVal := DataSet['Company']; // Получение значения поля Company
  Accept := FieldVal[1] = 'S';  // Принять запись, если поле начинается с буквы S
end;
```

После выполнения второго этапа (установки свойства `Filtered` таблицы равным `True`) таблица примет вид, показанный на рис. 28.12. В сетке отображаются только те записи, которые удовлетворяют критерию фильтрации.

Filter Demo

CustNo	Company	Addr1
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy
1231	Unisco	PO Box Z-547
1351	Sight Diver	1 Neptune Lane
1354	Cayman Divers World Unlimited	PO Box 541
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj
1380	Blue Jack Aqua Center	23-738 Paddington Lane
1384	VIP Divers Club	32 Main St.
1510	Ocean Paradise	PO Box 8745
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.
1551	Marmot Divers Club	872 Queen St.
1560	The Depth Charge	15243 Underwater Fwy.
1563	Blue Sports	203 12th Ave. Box 746

Рис. 28.11. Неотфильтрованные данные таблицы `CUSTOMER`

Filter Demo

CustNo	Company	Addr1
1351	Sight Diver	1 Neptune Lane
2163	SCUBA Heaven	PO Box Q-8874
2165	Shangri-La Sports Center	PO Box D-5495
3051	San Pablo Dive Center	1701-D N Broadway
5163	Safari Under the Sea	PO Box 7456

Рис. 28.12. Отфильтрованные данные таблицы `CUSTOMER`

На заметку

Событие `OnFilterRecord` должно использоваться только в тех случаях, когда фильтр не может быть задан как значение свойства `Filter`. В последнем случае может быть достигнут значительный выигрыш в производительности. Например, при работе с базой данных SQL компонент `TTable` будет передавать в базу данных содержимое свойства `FILTER` как условие выражения `WHERE`, которое обычно обрабатывается значительно быстрее, — по сравнению с перебором всех записей, выполняемым в обработчике события `OnFilterRecord`.

Методы `FindFirst/FindNext`

Компонент `TDataSet` содержит методы `FindFirst()`, `FindNext()`, `FindPrior()` и `FindLast()`, которые используют фильтр для поиска записей, отвечающих его критерию поиска. Все эти функции работают на неотфильтрованных наборах данных и предусматривают вызов обработчика события `OnFilterRecord` набора данных. На основе критерия поиска в обработчике события эти функции должны найти соответственно первую, следующую, предыдущую или последнюю запись. Каждая из этих функций не имеет параметров и возвращает значение `Boolean`, которое отображает, существует ли искомая запись.

Поиск записи

Фильтры эффективны не только при определении просматриваемого подмножества записей некоторого набора данных, они также могут использоваться для поиска записей в наборе данных по значению одного или нескольких полей. Для этой цели класс TDataSet предоставляет метод Locate(). Обратите внимание на то, что функция Locate() выполняет поиск с помощью средств фильтрации и работает независимо от любых созданных для набора данных индексов. Метод Locate() определяется следующим образом:

```
function Locate(const KeyFields: string; const KeyValues: Variant; Options:
TLocateOptions): Boolean;
```

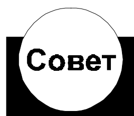
Первый параметр, KeyFields, содержит имя поля (или полей), по которому проводится поиск. Второй параметр, KeyValues, содержит значение (или значения) поля, которое отыскивается. Третий параметр, Options, позволяет указать требуемый тип поиска. Этот параметр имеет тип TLocateOptions, который представляет собой множество, определяемое следующим образом:

```
type
  TLocateOption = (loCaseInsensitive, loPartialKey);
  TLocateOptions = set of TLocateOption;
```

Если множество содержит член loCaseInsensitive, поиск будет выполняться без учета регистра. Если набор включает элемент loPartialKey, значения, содержащиеся в KeyValues, будут считаться удовлетворяющими критерию, даже если они являются подстрокой искомого значения.

Метод Locate() возвращает значение True, если искомая запись найдена. Например, для поиска первого появления значения 1356 в поле CustNo набора данных Table1 можно использовать следующий оператор:

```
Table1.Locate('CustNo', 1356, []);
```



Везде, где только это возможно, используйте для поиска записей метод Locate(), поскольку он всегда пытается применить самый быстрый из возможных методов поиска элемента, в случае необходимости временно переключаясь на индексный метод поиска. Это сделает вашу программу независимой от индексов. Кроме того, если выяснится, что индекс по некоторому из полей больше не потребуется, или, напротив, что добавление индекса увеличит производительность приложения, можно будет изменить только данные, не изменяя при этом код приложения.

Использование компонента TTable

В этом разделе речь пойдет об основных свойствах и методах компонента TTable, а также о способах их использования. В частности, мы обсудим вопросы поиска записей, фильтрации записей с использованием диапазонов, а также способы создания таблиц. Здесь же рассматриваются события компонента TTable.

Поиск записей

Для выполнения поиска записей в таблице библиотека VCL предлагает несколько методов. При работе с таблицами dBASE или Paradox, Delphi предполагает, что используемые для поиска поля индексированы. Для SQL-таблиц быстродействие поиска будет уменьшаться при проведении поиска по неиндексированным полям.

В качестве примера будем использовать некоторую таблицу, проиндексированную по первому полю с числовым типом и по второму полю с текстовым типом. В этом случае поиск записей можно будет выполнять одним из двух способов: с помощью метода FindKey() или с помощью пары методов SetKey()..GotoKey().

Метод FindKey()

Метод FindKey() класса TTable позволяет искать запись, используя одно или несколько ключевых полей при одном вызове функции. В качестве параметра методу FindKey() передается массив типа array of const, содержащий критерии поиска. Если поиск прошел успешно, метод возвращает значение True. Например, следующий оператор вызывает перемещение в наборе данных к записи, где первое поле в индексе имеет значение 123, а второе содержит строку Hello:

```
if not Table1.FindKey([123, 'Hello']) then MessageBeep(0);
```

Если поле не найдено, метод FindKey() возвращает значение False и компьютер издает звуковой сигнал.

Методы SetKey()..GotoKey()

Вызов метода SetKey() класса TTable переводит таблицу в режим, при котором ее поля подготавливаются к заполнению значениями, представляющими собой критерии поиска. Как только критерий поиска будет установлен, можно будет вызвать метод GotoKey() для выполнения поиска требуемой записи в направлении сверху вниз. Описанный в предыдущем разделе пример с парой методов SetKey()..GotoKey() можно записать следующим образом:

```
with Table1 do begin
    SetKey;
    Fields[0].AsInteger := 123;
    Fields[1].AsString := 'Hello';
    if not GotoKey then MessageBeep(0);
end;
```

Поиск ближайшего соответствия

Подобным же образом метод FindNearest() или пару методов SetKey()..GotoNearest() можно использовать для поиска в таблице значения, которое наиболее близко соответствует критерию поиска. Для поиска первой записи, где значение первого индексированного поля наиболее близко (больше или равно) 123, используйте следующий код:

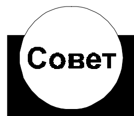
```
Table1.FindNearest([123]);
```

И опять-таки, в качестве параметра методу FindNearest() передается массив типа array of const, содержащий значения полей, по которым требуется выполнить поиск.

Методы SetKey()..GotoNearest() можно использовать для поиска следующим образом:

```
with Table1 do begin
  SetKey;
  Fields[0].AsInteger := 123;
  GotoNearest;
end;
```

Если поиск завершится успешно и свойство KeyExclusive объекта таблицы имеет значение False, указатель текущей записи будет установлен на первую из записей соответствующих критерию поиска. Если свойство KeyExclusive имеет значение True, текущей записью будет сделана запись, следующая за последней из всех записей, соответствующих условию поиска.



Если поиск выполняется по проиндексированному полю таблицы, предпочтительнее использовать методы FindKey() и FindNearest(), а не SetKey()..Goto*X*() — это уменьшит объем программы и, следовательно, количество возможных ошибок.

Использование индексов

Во всех описанных выше методах поиска подразумевается, что поиск проводится с использованием первичного индекса таблицы. Если требуется выполнить поиск, используя вторичный индекс, необходимо поместить имя этого индекса в параметр IndexName объекта таблицы. Например, если в таблице существует вторичный индекс по полю Company с именем ByCompany, то поиск записи по компании Unisco можно выполнить следующим образом:

```
with Table1 do begin
  IndexName := 'ByCompany';
  SetKey;
  FieldValues['Company'] := 'Unisco';
  GotoKey;
end;
```

Помните, что переключение индекса открытой таблицы создает заметную дополнительную нагрузку на систему. Поэтому присвоение свойству IndexName нового значения будет связано с ощутимой задержкой в работе программы — секунда и более.

Диапазоны (range) позволяют отфильтровывать таблицу так, что остаются только записи со значениями полей, попадающими внутрь определенных границ. Диапазоны обрабатываются подобно поиску по ключу, и, как и при поиске, существует несколько способов применения диапазона к выбранной таблице. Можно использовать метод SetRange() или группу методов SetRangeStart(), SetRangeEnd() и ApplyRange().

Если работа ведется с таблицами dBASE или Paradox, диапазоны могут применяться только к проиндексированным полям. Если вы обрабатываются SQL-данные, то отсутствие индексирования сказывается снижением производительности обработки.

Метод SetRange()

Как и метод FindKey() или FindNearest(), метод SetRange() позволяет выполнить достаточно сложные действия над таблицей с помощью единственного вызова этой функции. В качестве параметров методу SetRange() передаются два массива типа array of const. Первый представляет значения полей начала диапазона, а второй — значения полей конца диапазона. В качестве примера ниже показан оператор, выполняющий фильтрацию записей, у которых значение первого поля больше или равно 10, но меньше или равно 15:

```
Table1.SetRange([10], [15]);
```

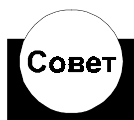
Метод ApplyRange()

Использование метода установки диапазона ApplyRange() предполагает выполнение следующих действий.

1. Вызовите метод SetRangeStart(), а затем модифицируйте массив свойств Fields[] таблицы, установив в нем начальное значение ключевого поля (или полей).
2. Вызовите метод SetRangeEnd() и вновь модифицируйте массив свойств Fields[], установив конечное значение ключевого поля (или полей).
3. Вызовите метод ApplyRange() для установки нового фильтра диапазона.

Продемонстрируем описанные действия на примере.

```
with Table1 do begin
  SetRangeStart;
  Fields[0].AsInteger := 10;    // Установка начального значения
  SetRangeEnd;
  Fields[0].AsInteger := 15;    // Установка конечного значения
  ApplyRange;
end;
```



Для фильтрации записей везде где только возможно используйте метод SetRange() — это уменьшит вероятность появления ошибок в вашей программе.

Для удаления установленного методом ApplyRange() или SetRange() фильтра и приведения таблицы в исходное состояние следует вызвать метод CancelRange() объекта TTable.

Главная/подчиненная таблицы

Очень часто при программировании работающих с базами данных приложений приходится обрабатывать информацию, структура которой требует разделения ее на несколько отдельных таблиц. Классический пример такой ситуации — таблица клиентов с одной записью по каждому клиенту и таблица заказов с одной записью по каждому заказу. Поскольку каждый заказ выполняется одним из клиентов, между двумя наборами данных естественным образом возникает взаимосвязь, называемая *отношением*. Данное отношение имеет тип *один ко многим* (one-to-many), поскольку один пользователь может сделать несколько заказов (в этом случае таблица клиентов становится главной, а таблица заказов — подчиненной).

Delphi упрощает создание подобных типов отношений между таблицами. Фактически все требуемые отношения устанавливаются еще на этапе разработки в окне инспектора объектов, т.е. для этого абсолютно не нужно вводить какой-либо программный текст. Откройте новый проект и поместите в его форму по два компонента TTable, TDataSource и TDBGrid. Компонент TDBGrid1 подключается к TTable1 через TDataSource1, а TDBGrid2 — к TTable2 через TDataSource2. Параметру DatabaseName обеих таблиц присвойте значение DBDEMOS, а затем подключите объект Table1 к таблице CUSTOMER.DB, а объект Table2 — к таблице ORDERS.DB. Созданная форма будет иметь вид, показанный на рис. 28.13.

Теперь у нас есть две несвязанные таблицы, помещенные в одну форму. Осталось лишь определить отношения между таблицами, для чего требуется присвоить нужные значения свойствам MasterSource и MasterFields объекта подчиненной таблицы. Свойству

MasterSource таблицы Table2 следует присвоить значение DataSource1. При попытке изменить значение свойства MasterFields в строке инспектора объектов появляется кнопка, по щелчку на которой на экране раскрывается окно редактора свойств Field Link Designer, показанное на рис. 28.14.

Рис. 28.13. Форма с главной и подчиненной таблицами в процессе разработки

Рис. 28.14. Окно редактора свойств Field Link Designer

В этом диалоговом окне следует указать, какие общие поля будут использоваться для связывания таблиц друг с другом. Для этих двух таблиц существует одно общее поле CustNo — числовой идентификатор, представляющий клиента. Поскольку поле CustNo не является частью первичного индекса таблицы ORDERS, необходимо переключиться на вторичный индекс, который включает поле CustNo. Это делается с помощью раскрывающегося списка Available Indexes диалогового окна Field Link Designer — в нем следует выбрать индекс CustNo. Затем в списках Detail Fields и Master Fields нужно выбрать общие поля и, щелкнув на кнопке Add, создать связь между таблицами. Для закрытия окна Field Link Designer щелкните на кнопке OK.

Обратите внимание на то, что теперь при перемещении по записям таблицы Table1 содержимое таблицы Table2 будет ограничиваться только теми записями, которые имеют значение поля CustNo, одинаковое со значением поля CustNo таблицы Table1. Окончательный вид формы приложения показан на рис. 28.15.

Рис. 28.15. Форма приложения с главной и подчиненной таблицами

События компонента TTable

Компонент TTable предлагает события, которые генерируются перед и после удаления, редактирования и вставки записей в таблицу, при внесении или отмене изменений и при открытии или закрытии таблицы. Таким образом, разработчику приложений для работы с базами данных предоставляется полный контроль над источником данных. Эти события имеют следующий вид: Before*XXX* и After*XXX*, где *XXX* — это Delete, Edit, Insert, Open и т.д. Упомянутые события не требуют особых разъяснений, а их использование в приложениях баз данных было описано в частях II, "Профессиональное программирование", и III, "Компонентно-ориентированная разработка", этой книги.

Событие OnNewRecord объекта TTable генерируется всякий раз при помещении новой записи в таблицу. Обработчик этого события — идеальное место для выполнения небольших обслуживающих задач. В качестве примера можно привести подсчет итоговых значений по всем новым записям, добавляемым в таблицу в данном сеансе работы с базой.

Событие OnCalcFields генерируется всякий раз, когда табличный курсор перемещается с текущей записи или текущая запись изменяется. Подготовив обработчик события OnCalcFields, можно обеспечить актуальность значений вычисляемых полей при любых модификациях таблицы.

Создание таблицы в программе

Рано или поздно, но обязательно потребуется вместо предварительного создания всех таблиц базы данных (например, с помощью утилиты Database Desktop) с последующим использованием их в приложениях, создавать локальные таблицы непосредственно в программе (для ее собственных нужд). Компоненты библиотеки VCL предусматривают и такую возможность. Компонент TTable включает метод CreateTable(), который позволяет создавать таблицы данных на диске. Для создания таблицы в программе выполните следующие действия.

1. Создайте экземпляр компонента TTable.
2. Назначьте свойству DatabaseName таблицы папку или существующий псевдоним.
3. Присвойте таблице уникальное имя, поместив его в свойство TableName.
4. Установите требуемое значение свойства TableType, определяющее тип создаваемой таблицы. Если его значение будет равно ttDefault, тип создаваемой таблицы будет соответствовать расширению имени таблицы, указанному в свойстве TableName (например, расширение .DB соответствует таблицам Paradox, а расширение .DBF — таблицам dBASE).
5. Для добавления полей к таблице воспользуйтесь методом Add() объекта TTable.FieldDefs. Этому методу следует передавать четыре параметра:
   - строку, задающую имя поля;
   - переменную типа TFieldType, задающую тип поля;
   - параметр word, устанавливающий размер поля. Запомните, что этот параметр действителен только для типов String и Memo, т.е. полей переменного размера. Такие типы полей, как Integer и Date, всегда одного размера, поэтому данный параметр к ним не применим;
   - параметр Boolean, указывающий на обязательность заполнения этого поля. Всем обязательным полям должны быть присвоены значения, прежде чем новая запись сможет быть помещена в таблицу.

6. Если требуется создать для таблицы индекс, для его организации воспользуйтесь методом Add() объекта TTable.IndexDefs. Методу IndexDefs.Add() передаются следующие параметры:
   - строка, задающая имя индекса;
   - строка, задающая имя индексируемого поля. Можно создать составной ключ индекса (индекс по нескольким полям), для чего указывается список имен полей, разделенных точкой с запятой;
   - множество типа TIndexOptions, определяющее тип создаваемого индекса.
7. Вызовите метод TTable.CreateTable().

В приведенном ниже фрагменте кода создается таблица с полями типа Integer, String и Float, с индексом по полю Integer. Таблице присваивается имя FOO.DB, и она помещается в папку C:\TEMP.

```
begin
  with TTable.Create(Self) do begin   // Создание объекта TTable
  DatabaseName := 'c:\temp';         // Указание папки или псевдонима
  TableName := 'FOO';                // Присваивание таблице имени
  TableType := ttParadox;           // Создание таблицы СУБД Paradox
  with FieldDefs do begin
    Add('Age', ftInteger, 0, True);   // Добавление поля типа integer
    Add('Name', ftString, 25, False); // Добавление поля типа string
    Add('Weight', ftFloat, 0, False); // Добавление поля типа float
  end;
  { Создание первичного индекса по полю Age... }
  IndexDefs.Add('', 'Age', [ixPrimary, ixUnique]);
  CreateTable;                      // Создание таблицы
  end;
end;
```

На заметку

Как упоминалось ранее, метод TTable.CreateTable() "работает" только для локальных таблиц. Для SQL-таблиц необходимо использовать возможности компонента TQuery (они будут рассмотрены в следующей главе).

Модули данных

Модули данных (data modules) позволяют расположить все правила и отношения базы данных в одном месте для того, чтобы обеспечить их совместное использование отдельными проектами, группами или организациями. В библиотеке VCL модули данных инкапсулируются компонентом TDataModule. Этот компонент можно представить себе как невидимую форму, в которую помещаются компоненты доступа к данным, предназначенные для использования их в проекте. Создать экземпляр компонента TDataModule несложно — в главном меню выберите команду File⇨New, а затем во вкладке New раскрывшегося окна New Item — объект Data Module.

Самая очевидная причина, по которой предпочтительнее использовать компонент TDataModule, нежели помещать компоненты доступа к данным в форму, состоит в том, что это упрощает доступ к одним и тем же данным из нескольких форм и модулей проекта. В

более сложных ситуациях может потребоваться упорядочить использование многочисленных компонентов TTable, TQuery и/или TStoredProc, установить постоянные отношения между компонентами и, возможно, правила, определенные на уровне поля (например, минимальные/максимальные значения или форматы отображения). В целом, этот ассортимент компонентов доступа к данным будет отображать бизнес-правила вашего предприятия. Выполнив множество подобных действий, вряд ли вы захотите повторять их вновь и вновь для других приложений. Чтобы избежать этой печальной необходимости, поместите созданный модуль данных в хранилище объектов (Object Repository). Это позволит вам многократно его использовать. При проведении работ группой разработчиков хранилище объектов следует разместить на общедоступном сетевом диске — это обеспечит доступ к нему всем разработчикам группы.

В приведенном далее примере создается экземпляр простого модуля данных, благодаря которому из многих форм можно будет получить доступ к одним и тем же данным. В приложениях работы с базами данных, рассматриваемых в последующих главах, в модулях данных будут устанавливаться более сложные отношения.

Пример приложения с поиском, фильтрацией и выделением диапазона данных

Теперь рассмотрим демонстрационное приложение, в котором использованы важнейшие концепции, обсуждавшиеся в этой главе. В частности, в этом приложении продемонстрировано правильное использование методов фильтрации, поиска по ключу и выделения диапазонов данных. Предлагаемый проект называется SRF и содержит несколько форм. В главной форме содержится сетка, предназначенная для просмотра данных таблицы, а остальные формы демонстрируют различные рассмотренные ранее концепции. Назначение каждой формы поясняется далее в этой главе.

Модуль данных

Модуль данных этого проекта необходимо рассмотреть в первую очередь. Он называется DM и содержит компоненты TTable и TDataSource. Компонент TTable с именем Table1 связан с таблицей CUSTOMERS.DB базы данных под псевдонимом DBDEMOS. Компонент типа TDataSource с именем DataSource1 связан с компонентом Table1. Все элементы управления для работы с данными в этом проекте будут использовать компонент DataSource1 в качестве источника данных. Модуль данных DM содержится в программном модуле DataMod. На рис. 28.16 этот модуль показан в режиме разработки.

Главная форма

Главная форма проекта SRF называется MainForm и представлена на рис. 28.17. Ее текст содержится в модуле Main. Форма содержит элемент управления TDBGrid с именем DBGrid1, предназначенный для просмотра таблицы, а также переключатель, позволяющий переключаться между различными индексами таблицы. Компонент DBGrid1, как уже упоминалось ранее, связан с источником данных DM.DataSource1.

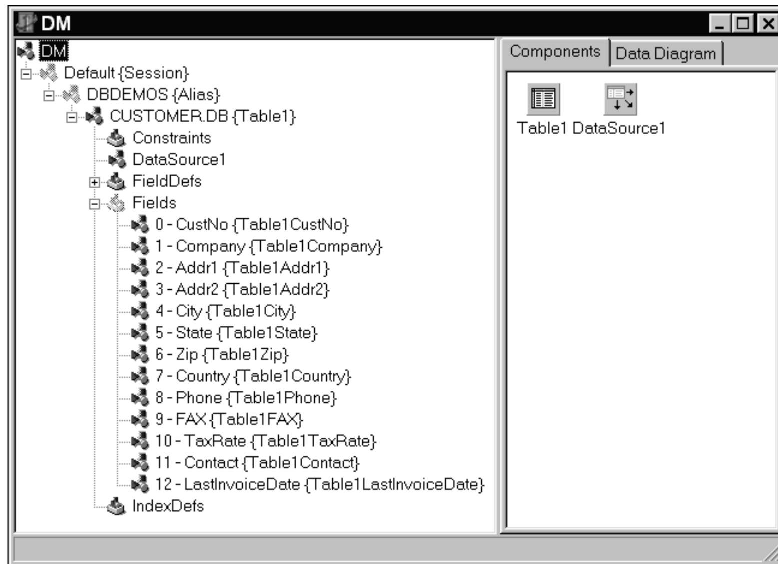


Рис. 28.16. Модуль данных `DM`

На заметку

Для того чтобы компонент `DBGrid1` можно было связать с источником данных `DM.DataSource1` в режиме разработки, модуль `DataMod` должен быть упомянут в операторе `uses` модуля `Main`. Простейший способ сделать это — отобразить модуль `Main` в окне редактора кода, а затем выбрать команду File⇨Use Unit главного меню. На экране раскроется диалоговое окно со списком модулей проекта, в котором следует выбрать модуль `DataMod`. Эту же операцию необходимо повторить для каждого модуля, из которого необходимо получить доступ к данным, определяемым в модуле `DM`.

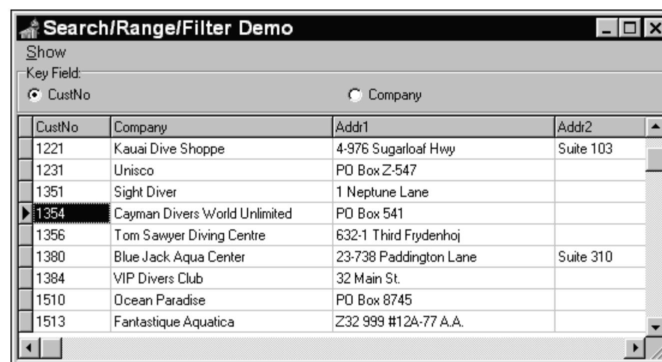


Рис. 28.17. Форма `MainForm` *в проекте* `SRF`

Переключатель `RGKeyField` используется для определения того, какой из двух индексов таблицы будет активен в текущий момент. Код обработчика события `OnClick` переключателя `RGKeyField` выглядит следующим образом:

```
procedure TMainForm.RGKeyFieldClick(Sender: TObject);
begin
  case RGKeyField.ItemIndex of
    0: DM.Table1.IndexName := '';          // Первичный индекс
    1: DM.Table1.IndexName := 'ByCompany'; // Вторичный индекс, по компании
  end;
end;
```

Форма MainForm содержит также компонент TMainMenu с именем MainMenu1, позволяющий открывать и закрывать каждую из форм приложения. Существуют следующие элементы меню: Key Search, Range, Filter и Exit. Полный текст модуля Main.pas приведен в листинге 28.3.

Листинг 28.3. Текст модуля Main.pas

```
unit Main;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Grids, DBGrids, DB, DBTables,
  Buttons, Mask, DBCtrls, Menus;

type
  TMainForm = class(TForm)
    DBGrid1: TDBGrid;
    RGKeyField: TRadioGroup;
    MainMenu1: TMainMenu;
    Forms1: TMenuItem;
    KeySearch1: TMenuItem;
    Range1: TMenuItem;
    Filter1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    procedure RGKeyFieldClick(Sender: TObject);
    procedure KeySearch1Click(Sender: TObject);
    procedure Range1Click(Sender: TObject);
    procedure Filter1Click(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
  private
    { Закрытые объявления }
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation
```

```
uses DataMod, KeySrch, Rng, Fltr;

{$R *.DFM}

procedure TMainForm.RGKeyFieldClick(Sender: TObject);
begin
  case RGKeyField.ItemIndex of
    0: DM.Table1.IndexName := '';          // Первичный индекс
    1: DM.Table1.IndexName := 'ByCompany'; // Вторичный индекс, по компании
  end;
end;

procedure TMainForm.KeySearch1Click(Sender: TObject);
begin
  KeySearch1.Checked := not KeySearch1.Checked;
  KeySearchForm.Visible := KeySearch1.Checked;
end;

procedure TMainForm.Range1Click(Sender: TObject);
begin
  Range1.Checked := not Range1.Checked;
  RangeForm.Visible := Range1.Checked;
end;

procedure TMainForm.Filter1Click(Sender: TObject);
begin
  Filter1.Checked := not Filter1.Checked;
  FilterForm.Visible := Filter1.Checked;
end;

procedure TMainForm.Exit1Click(Sender: TObject);
begin
  Close;
end;

end.
```

Форма выделения диапазона данных

Текст формы RangeForm (рис. 28.18) находится в модуле Rng. Эта форма позволяет устанавливать диапазон данных, отображаемых в форме MainForm. В зависимости от активного индекса, вводимые в поля Range Start (Начало диапазона) и Range End (Конец диапазона) значения могут быть числовыми (первичный индекс) или текстовыми (вторичный индекс). Текст модуля RNG.PAS приведен в листинге 28.4.

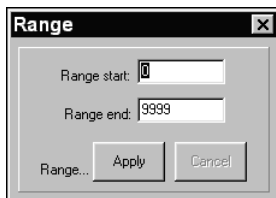


Рис. 28.18. Форма RangeForm

Листинг 28.4. Текст модуля RNG.PAS

```
unit Rng;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls;

type
  TRangeForm = class(TForm)
    Panel1: TPanel;
    Label2: TLabel;
    StartEdit: TEdit;
    Label1: TLabel;
    EndEdit: TEdit;
    Label7: TLabel;
    ApplyButton: TButton;
    CancelButton: TButton;
    procedure ApplyButtonClick(Sender: TObject);
    procedure CancelButtonClick(Sender: TObject);
  private
    { Закрытые объявления }
    procedure ToggleRangeButtons;
  public
    { Открытые объявления }
  end;

var
  RangeForm: TRangeForm;

implementation

uses DataMod;

{$R *.DFM}

procedure TRangeForm.ApplyButtonClick(Sender: TObject);
begin
  { Установка диапазона выбираемых из набора данных записей от значения
    StartEdit до значения EndEdit. Строки будут вновь неявно
    конвертированы в числовые значения. }
  DM.Table1.SetRange([StartEdit.Text], [EndEdit.Text]);
  ToggleRangeButtons;  // Активизация соответствующих кнопок
end;

procedure TRangeForm.CancelButtonClick(Sender: TObject);
begin
  DM.Table1.CancelRange; // Удаление установленного диапазона
```

```
  ToggleRangeButtons;    // Активизация соответствующих кнопок
end;

procedure TRangeForm.ToggleRangeButtons;
begin
  { Переключение свойства Enabled кнопок диапазона }
  ApplyButton.Enabled := not ApplyButton.Enabled;
  CancelButton.Enabled := not CancelButton.Enabled;
end;

end.
```

На заметку

Обратите внимание на следующую строку кода из модуля Rng:

`DM.Table1.SetRange([StartEdit.Text], [EndEdit.Text]);`

Не кажется ли вам странным тот факт, что методу `SetRange()`, всегда передается строковое значение, несмотря на то что ключевое поле может быть числового или текстового типа? Delphi допускает это, поскольку методы `SetRange()`, `FindKey()` и `FindNearest()` автоматически выполняют преобразование из типа `String` в тип `Integer`, и наоборот.
Это означает, что в данной ситуации нет необходимости заботится о вызове функции `IntToStr()` или `StrToInt()` — все преобразования будут выполнены автоматически.

Форма поиска по ключу

Текст формы KeySearchForm содержится в модуле KeySrch. Она предназначена для поиска в таблице записи с конкретным значением ключа. Поиск может выполняться одним из двух предлагаемых способов. Если переключатель установлен в положение Normal, пользователь может ввести в поле Search for требуемое значение ключа и с помощью щелчка на кнопке Exact или Nearest выполнять поиск точно отвечающей условию записи или ближайшей к ней. Если переключатель установлен в положение Incremental, в таблице будет выполняться пошаговый поиск непосредственно в процессе ввода условия поиска в поле Search for. При этом обращение к таблице выполняется при каждом изменении значения в этом поле. Описываемая форма показана на рис. 28.19, а текст модуля KeySrch приведен в листинге 28.5.

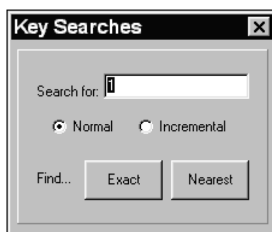


Рис. 28.19. Форма KeySearchForm

Листинг 28.5. Текст модуля KeySrch.PAS

```
unit KeySrch;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls;
```

```
type
  TKeySearchForm = class(TForm)
    Panel1: TPanel;
    Label3: TLabel;
    SearchEdit: TEdit;
    RBNormal: TRadioButton;
    Incremental: TRadioButton;
    Label6: TLabel;
    ExactButton: TButton;
    NearestButton: TButton;
    procedure ExactButtonClick(Sender: TObject);
    procedure NearestButtonClick(Sender: TObject);
    procedure RBNormalClick(Sender: TObject);
    procedure IncrementalClick(Sender: TObject);
  private
    procedure NewSearch(Sender: TObject);
  end;

var
  KeySearchForm: TKeySearchForm;

implementation

uses DataMod;

{$R *.DFM}

procedure TKeySearchForm.ExactButtonClick(Sender: TObject);
begin
  { Попытка найти запись, где ключевое поле точно соответствует значению }
  { поля SearchEdit. Заметьте, что Delphi обеспечивает преобразование }
  { типов из строкового значения поля в числовое значение ключевого поля. }
  if not DM.Table1.FindKey([SearchEdit.Text]) then
    MessageDlg(Format('Match for "%s" not found.', [SearchEdit.Text]),
               mtInformation, [mbOk], 0);
end;

procedure TKeySearchForm.NearestButtonClick(Sender: TObject);
begin
  { Поиск ближайшего соответствия значению в поле SearchEdit.}
  {Заметьте, что снова выполняется неявное преобразование типа.}
  DM.Table1.FindNearest([SearchEdit.Text]);
end;

procedure TKeySearchForm.NewSearch(Sender: TObject);
{ Этот метод связывается с событием OnChange поля SearchEdit,}
{при установке переключателя в положение Incremental. }
begin
  DM.Table1.FindNearest([SearchEdit.Text]);  // Поиск текста
end;
```

```
procedure TKeySearchForm.RBNormalClick(Sender: TObject);
begin
  ExactButton.Enabled := True;  // Активизация кнопок поиска
  NearestButton.Enabled := True;
  SearchEdit.OnChange := Nil;   // Отключение обработчика события OnChange
end;

procedure TKeySearchForm.IncrementalClick(Sender: TObject);
begin
  ExactButton.Enabled := False;     // Деактивизация кнопок поиска
  NearestButton.Enabled := False;
  SearchEdit.OnChange := NewSearch; // Организация перехвата события OnChange
  NewSearch(Sender);                // Поиск текущего фрагмента текста
end;

end.
```

Текст приведенного модуля не должен вызывать у вас вопросов. В нем методам FindKey() и FindNearest() вновь передается строковое значение в полной уверенности, что требуемое преобразование типов произойдет автоматически. Полагаем, вы оцените небольшой трюк, используемый для включения и отключения пошагового поиска “на лету”. С этой целью событию OnChange элемента управления SearchEdit присваивается либо необходимый метод, либо значение Nil. При назначении в качестве обработчика реального метода, событие OnChange будет генерироваться всякий раз при изменении значения в текстовом поле. Вызов в этом обработчике метода FindNearest() позволяет организовать в таблице пошаговый поиск, выполняемый при любом изменении пользователем условия поиска.

Форма фильтрации

Расположенная в модуле Fltr форма FilterForm (рис. 28.20) предназначена для решения двух задач. Во-первых, она позволяет фильтровать отображаемые в таблице данные по значению поля State, которое должно соответствовать заданному пользователем значению. Во-вторых, эта форма позволяет выбирать из таблицы записи, в которых значение некоторого поля равно введенному пользователем значению.

Для решения задачи фильтрации записей не требуется вводить большой код. Состояние флажка опции Filter on this State (объект cbFiltered) определяет значение свойства Filtered объекта DM.Table1. Это выполняется с помощью приведенного ниже оператора в обработчике события cbFiltered.OnClick:

```
DM.Table1.Filtered := cbFiltered.Checked;
```

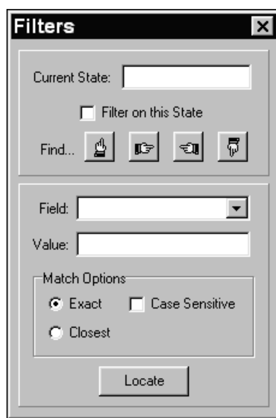


Рис. 28.20. Форма FilterForm

Когда свойство DM.Table1.Filtered принимает значение True, таблица Table1 фильтрует записи с помощью приведенного ниже метода OnFilterRecord, который расположен в модуле DataMod.

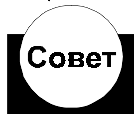
```
procedure TDM.Table1FilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
begin
  { Запись выбирается на обработку, если значение поля State  }
  { равно значению, введенному в текстовое поле DBEdit1.Text. }
  Accept := Table1State.Value = FilterForm.DBEdit1.Text;
end;
```

Для выполнения условного поиска необходимо следующим образом использовать метод Locate() компонента TTable:

```
DM.Table1.Locate(CBField.Text, EValue.Text, LO);
```

Имя поля выбирается пользователем из раскрывающегося списка CBField. Содержимое списка генерируется событием OnCreate формы с помощью следующего кода, выполняющего просмотр полей таблицы Table1:

```
procedure TFilterForm.FormCreate(Sender: TObject);
var
  i: integer;
begin
  with DM.Table1 do begin
    for i := 0 to FieldCount - 1 do
      CBField.Items.Add(Fields[i].FieldName);
  end;
end;
```



Приведенный код работает только в том случае, если модуль данных DM создан раньше формы. Любые попытки доступа к модулю данных до его создания приведут, скорее всего, к появлению ошибки типа Access Violation. Чтобы иметь уверенность, что модуль данных DM будет создан раньше, чем любая из дочерних форм, мы вручную отредактировали порядок создания форм в списке Auto-create forms, расположенном во вкладке Forms диалогового окна Project Options (это окно выводится с помощью команды Project⇨Options главного меню).
Безусловно, прежде всего должна создаваться главная форма, однако следует иметь гарантии, что сразу за ней последует модуль данных — прежде, чем будет создана любая другая форма приложения.

Полный текст модуля Fltr приведен в листинге 28.6.

Листинг 28.6. Текст модуля Fltr.PAS

```
unit Fltr;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Buttons, Mask, DBCtrls, ExtCtrls;
```

```
type
  TFilterForm = class(TForm)
    Panel1: TPanel;
    Label4: TLabel;
    DBEdit1: TDBEdit;
    cbFiltered: TCheckBox;
    Label5: TLabel;
    SpeedButton1: TSpeedButton;
    SpeedButton2: TSpeedButton;
    SpeedButton3: TSpeedButton;
    SpeedButton4: TSpeedButton;
    Panel2: TPanel;
    EValue: TEdit;
    LocateBtn: TButton;
    Label1: TLabel;
    Label2: TLabel;
    CBField: TComboBox;
    MatchGB: TGroupBox;
    RBExact: TRadioButton;
    RBClosest: TRadioButton;
    CBCaseSens: TCheckBox;
    procedure cbFilteredClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure LocateBtnClick(Sender: TObject);
    procedure SpeedButton1Click(Sender: TObject);
    procedure SpeedButton2Click(Sender: TObject);
    procedure SpeedButton3Click(Sender: TObject);
    procedure SpeedButton4Click(Sender: TObject);
  end;

var
  FilterForm: TFilterForm;

implementation

uses DataMod, DB;

{$R *.DFM}

procedure TFilterForm.cbFilteredClick(Sender: TObject);
begin
  { Таблица фильтруется, если установлен соответствующий флажок опции }
  DM.Table1.Filtered := cbFiltered.Checked;
end;

procedure TFilterForm.FormCreate(Sender: TObject);
var
  i: integer;
```

```
begin
  with DM.Table1 do begin
    for i := 0 to FieldCount - 1 do
      CBField.Items.Add(Fields[i].FieldName);
  end;
end;

procedure TFilterForm.LocateBtnClick(Sender: TObject);
var
  LO: TLocateOptions;
begin
  LO := [];
  if not CBCaseSens.Checked then Include(LO, loCaseInsensitive);
  if RBClosest.Checked then Include(LO, loPartialKey);
  if not DM.Table1.Locate(CBField.Text, EValue.Text, LO) then
    MessageDlg('Unable to locate match', mtInformation, [mbOk], 0);
end;

procedure TFilterForm.SpeedButton1Click(Sender: TObject);
begin
  DM.Table1.FindFirst;
end;

procedure TFilterForm.SpeedButton2Click(Sender: TObject);
begin
  DM.Table1.FindNext;
end;

procedure TFilterForm.SpeedButton3Click(Sender: TObject);
begin
  DM.Table1.FindPrior;
end;

procedure TFilterForm.SpeedButton4Click(Sender: TObject);
begin
  DM.Table1.FindLast;
end;

end.
```

Другие типы наборов данных: TQuery и TStoredProc

В этой главе компоненты TQuery и TStoredProc мы рассмотрим достаточно кратко — лишь как потомков компонента TDataSet и близких родственников компонента TTable. Детальное их рассмотрение будет отложено до следующей главы.

Компонент TQuery

Компонент TQuery позволяет использовать язык SQL для получения определенного набора данных из одной или нескольких таблиц. В Delphi компонент TQuery можно использовать как с серверами данных файлового типа (например, Paradox и dBASE), так и с серверами SQL. После назначения свойству DatabaseName компонента TQuery псевдонима или пути к каталогу, можно ввести в его свойство SQL те операторы языка SQL, которые необходимо выполнить в выбранной базе данных. Например, если объект Query1 связан с псевдонимом DBDEMOS, следующий код будет возвращать все записи в таблице BIOLIFE, в которых значение поля Length (cm) больше 100:

```
select * from BIOLIFE where BIOLIFE."Length (cm)" > 100
```

Подобно другим наборам данных, запрос будет выполняться в том случае, если его свойству Active присвоено значение True или после вызова метода Open(). Если же требуется выполнить запрос, который не возвращает результирующий набор данных (например, insert into), для его выполнения необходимо использовать метод ExecuteSQL(), а не метод Open().

Другим важным свойством компонента TQuery является свойство RequestLive. Это свойство показывает, можно ли редактировать полученный результирующий набор данных. Если возвращаемые запросом данные требуется редактировать, его значение следует установить равным True.

На заметку

Простая установка значения свойства RequestLive не гарантирует, что результирующий набор данных можно будет редактировать. В зависимости от структуры запроса, ядро BDE может оказаться неспособным получить результирующий набор, доступный для редактирования. Например, запросы, в которых содержится ключевое слово HAVING, используется функция TO_DATE или поле с *абстрактным типом данных* (ADT — Abstract Data Type), предоставляют результаты, которые нельзя отредактировать. (Для получения полного перечня существующих ограничений обратитесь к документации по BDE.) Чтобы определить, можно ли будет редактировать результаты запроса, проверьте перед его открытием значение свойства CanModify.

В следующей главе мы подробнее поговорим о свойствах компонента TQuery, таких как параметрические запросы и SQL-оптимизация.

Компонент TStoredProc

Компонент TStoredProc предлагает средства для выполнения хранимых процедур на SQL-сервере. Поскольку это средство является специфическим для каждого типа сервера (и не предназначено для начинающих разработчиков приложений работы с базами данных), мы отложим его подробное рассмотрение до следующей главы.

Таблицы в текстовом файле

В Delphi реализована ограниченная поддержка работы с таблицами в текстовом файле. Текстовые таблицы должны состоять из двух файлов: файла данных с расширением .TXT и файла-схемы с расширением .SCH. Оба файла должны иметь одно и то же имя, например FOO.TXT и FOO.SCH. Файл данных должен иметь записи фиксированной длины или разделители. Файл схемы указывает BDE способ интерпретации файла данных, предоставляя такую информацию, как имя поля, его размер и тип.

Формат файла-схемы

Формат файла-схемы подобен .INI-файлам в Windows. Название раздела в файле представляет собой имя таблицы (без расширения). В табл. 28.5 показаны элементы и возможные их значения в файле-схеме.

Таблица 28.5. Элементы и значения файла-схемы

Элемент	Возможное значение	Назначение
FILETYPE	VARYING	Поля в файле переменной длины и разделены специальным символом, а строки ограничены другим специальным символом
	FIXED	Каждое поле должно находиться на определенном смещении от начала строки
CHARSET	*(много)*	Определяет, какой языковой драйвер используется. В большинстве случаев это ASCII
DELIMITER	*(любой символ)*	Определяет, какой символ должен использоваться в качестве ограничителя для полей CHAR. Применяется только для таблиц типа VARYING
SEPARATOR	*(любой символ)*	Определяет, какой символ должен использоваться в качестве разделителя. Используется только для таблиц типа VARYING

С учетом информации, приведенной в табл. 28.5, файл-схема должен содержать запись следующего вида для каждого поля в таблице:

```
FieldX = Field Name, Field Type, Size, Decimal Places, Offset
```

Рассмотрим подробнее используемые параметры.

- *X* — представляет собой номер поля от 1 и до общего количества полей.
- *Field Name* — может быть любым строковым идентификатором. Не используйте в значениях кавычки или разделители строк.
- *Field Type* — может принимать одно из следующих значений:

Тип	Назначение
CHAR	Символьное или строковое поле
BOOL	Булево (T или F)
DATE	Дата в формате, определенном в BDE Config Tool
FLOAT	64-разрядное число с плавающей точкой
LONGINT	32-разрядное целое
NUMBER	16-разрядное целое
TIME	Время в формате, определенном в BDE Config Tool
TIMESTAMP	Дата и время в формате, определенном в BDE Config Tool

- *`Size`* — указывает общее число символов или разрядов. Для числовых полей это значение должно быть меньше или равно 20.
- *`Decimal Places`* (только для полей `FLOAT`) — определяет количество цифр после точки.
- *`Offset`* — используется только для таблиц `FIXED` и определяет положение символа, с которого начинается отдельное поле.

А теперь рассмотрим пример файла-схемы для таблицы `OPTeam`.

```
[OPTEAM]
FILETYPE = FIXED
CHARSET = ascii
Field1 = EmpNo,LONGINT,04,00,00
Field2 = Name,CHAR,16,00,05
Field3 = OfficeNo,CHAR,05,00,21
Field4 = PhoneExt,LONGINT,04,00,27
Field5 = Height,FLOAT,05,02,32
```

Ниже приведен файл схемы для версии `VARYING` подобной таблицы `OPTeam2`:

```
[OPTEAM2]
FILETYPE = VARYING
CHARSET = ascii
DELIMITER = "
SEPARATOR = ,
Field1 = EmpNo,LONGINT,04,00,00
Field2 = Name,CHAR,16,00,00
Field3 = OfficeNo,CHAR,05,00,00
Field4 = PhoneExt,LONGINT,04,00,00
Field5 = Height,FLOAT,05,02,00
```

Программы BDE весьма требовательны к формату файла-схемы. Если он содержит хотя бы один неверный символ или ошибочное слово, BDE не сможет корректно распознать весь файл данных. Если вы столкнулись с проблемой при доступе к текстовым данным, внимательно проверьте формат файла схемы.

Файл данных

Файл данных должен состоять либо из записей фиксированной длины (`FIXED`), либо из записей с разделителями (`VARYING`) и содержать по одной записи данных в каждой строке. Вот пример файла данных для описанной выше таблицы `OPTeam`:

```
2093 Steve Teixeira  C2121 1234 6.5
3265 Xavier Pacheco  C0001 3456 5.6
2610 Lino Tadros     E2126 5678 5.11
2900 Lance Bullock   C2221 9012 6.5
0007 Greg de Vries   F3169 7890 5.10
1001 Tillman Dickson C3456 0987 5.9
2611 Rory Bannon     E2127 6543 6.0
6908 Karl Santos     A1098 5893 5.6
0909 Mr. T           B0087 1234 5.9
```

Те же самые данные в файле OPTeam2 будут выглядеть следующим образом:

```
2093,"Steve Teixeira","C2121",1234,6.5
3265,"Xavier Pacheco","C0001",3456,5.6
2610,"Lino Tadros","E2126",5678,5.11
2900,"Lance Bullock","C2221",9012,6.5
0007,"Greg de Vries","F3169",7890,5.10
1001,"Tillman Dickson","C3456",0987,5.9
2611,"Rory Bannon","E2127",6543,6.0
6908,"Karl Santos","A1098",5893,5.6
0909,"Mr. T","B0087",1234,5.9
```

Использование текстовых таблиц

Текстовые таблицы можно использовать с компонентом TTable аналогично любому другому типу базы данных. Назначьте свойству DatabaseName псевдоним или каталог, содержащий .txt- и .sch-файлы, и установите свойство TableType равным ttASCII. Этого достаточно, чтобы увидеть все доступные в указанном расположении текстовые таблицы в раскрывающемся списке свойства TableName. Выберите одну из таблиц в этом списке — и вы увидите ее поля после связывания между собой компонентов TDataSource и TDBGrid. На рис. 28.21 показана форма, в которой отображается содержимое обеих таблиц OPTeam.

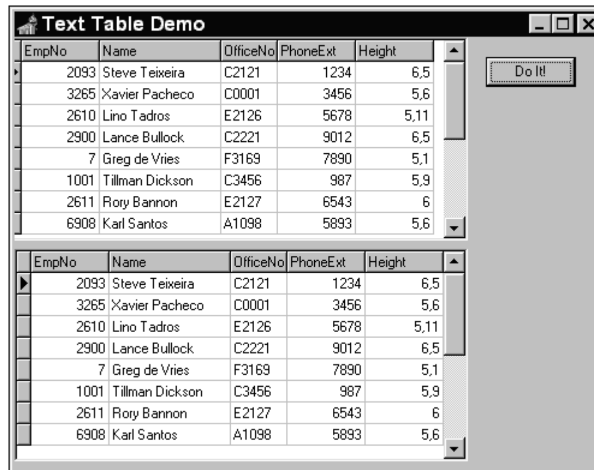


Рис. 28.21. Просмотр информации в текстовых таблицах

На заметку

Если все поля в текстовой таблице появляются собранными в одном поле, это значит, что BDE не может правильно интерпретировать ваш файл схемы.

Ограничения

Фирма Borland никогда не отдавала предпочтения использованию текстовых файлов (вместо настоящих баз данных того или иного типа). Поскольку текстовым файлам присущи определенные ограничения, мы решительно против применения файлов с текстовыми таблицами

для любых операций, кроме импортирования или экспортирования данных в реальные базы данных и из них. Ниже приведен список ограничений, которые необходимо учитывать при использовании текстовых таблиц.

- Индексы не поддерживаются, поэтому невозможно использовать ни один из методов компонента TTable, который требует наличия индекса.
- Использование компонента TQuery с текстовыми таблицами невозможно.
- Не поддерживается удаление записей.
- Вставка записей не поддерживается. Попытка вставить запись приведет к тому, что новая запись будет добавлена в конец таблицы.
- Не поддерживается ссылочная целостность.
- Не поддерживаются типы данных BLOB.
- Редактирование в таблице формата VARYING не поддерживается.
- Текстовые таблицы всегда открыты с исключительным правом доступа. Таким образом, таблица должна открываться в тексте программе, а не на стадии разработки.

Импорт текстовых таблиц

Как уже отмечалось ранее, текстовую таблицу имеет смысл использовать только для преобразования ее данных в какой-нибудь реальный формат базы данных. Ниже приведена последовательность действий, которые необходимо выполнить для копирования содержимого текстовой таблицы в базу данных формата Paradox с помощью компонента TBatchMove. Предположим, что существует форма, содержащая два объекта TTable и один объект TBatchMove. Первый объект TTable с именем TextTbl представляет собой текстовую таблицу, второй объект TTable с именем PDoxTbl — целевую таблицу Paradox. Компонент TBatchMove называется BM. Итак...

1. Свяжите объект TextTbl с текстовой таблицей, которую требуется импортировать (эта процедура описывалась выше).
2. Назначьте свойству DatabaseName объекта PDoxTbl целевой псевдоним или каталог, а свойству TableName — желаемое имя таблицы. Свойство TableType этого компонента установите равным ttParadox.
3. Установите свойство Source объекта BM равным TextTbl, а свойство Destination — равным PDoxTbl. Свойство Mode этого компонента должно иметь значение batCopy.
4. Щелкните правой кнопкой мыши на объекте BM и выберите в раскрывшемся контекстном меню команду Execute.
5. Поздравляем! Данные текстовой таблицы скопированы в таблицу СУБД Paradox.

Подключение с помощью ODBC

Известно, что ядро BDE может реально поддерживать только ограниченное число типов баз данных. Что же делать, если вам необходимо подключиться к такому типу базы данных, которая напрямую ядром BDE не поддерживается (например, Btrieve)? Можно ли в этом случае использовать Delphi в качестве среды разработки? Конечно же можно! Ядро BDE предлагает воспользоваться интерфейсом ODBC (Open Database Connectivity). Используя

драйвер ODBC, можно получить доступ к базе данных, которая напрямую ядром BDE не поддерживается. Эта возможность реализована только в версиях Delphi Professional и Delphi Enterprise Edition. ODBC — это стандарт, созданный фирмой Microsoft для разработки драйверов баз данных, не зависящих от производителя.

Где найти драйвер ODBC

Наилучший способ получить драйвер ODBC — использовать драйвер, предлагаемый разработчиком применяемой базы данных. Заказывая драйверы ODBC, имейте в виду, что существуют 16- и 32-разрядные драйверы ODBC, но Delphi 5 поддерживает только 32-разрядные драйверы. Кроме разработчика используемой вами СУБД, драйверы ODBC могут создаваться и другими компаниями. Так, фирма Microsoft распространяет драйверы ODBC для СУБД Access, Excel, SQL Server и FoxPro. Эти драйверы можно найти в ODBC Driver Pack или на компакт-диске MS Developer Network.

Не все драйверы ODBC эквивалентны! Многие предназначены только для конкретного пакета программ или имеют определенные функциональные ограничения. Примерами могут служить драйверы, поставлявшиеся с прежними версиями MS Office (они были способны работать только с MS Office). Удостоверьтесь, что приобретаемый драйвер ODBC сертифицирован для разработки приложений, а не только для использования с конкретным пакетом программ.

Пример использования ODBC: подключение к MS Access

Предположим, что у вас уже есть требуемый 32-разрядный драйвер фирмы Microsoft или другого поставщика. В этом разделе описана последовательность действий, необходимых для настройки драйвера с целью его использования компонентом `TTable` Delphi. СУБД Access напрямую поддерживается ядром BDE, но в данном случае это не имеет значения, поскольку цель этого раздела — привести пример использования драйверов ODBC с ядром BDE. Предполагается, что на жестком диске вашего компьютера пока еще нет какой-либо базы данных Access, и предлагаемая последовательность действий включает процедуру ее создания.

1. Установите драйвер, используя предоставленный поставщиком установочный диск. Затем откройте окно Control Panel системы Windows. В этом окне присутствует пиктограмма ODBC Data Source (32bit) (рис. 28.22). Дважды щелкните на ней. На экране раскроется диалоговое окно ODBC Data Source Administrator, показанное на рис. 28.23.
2. В диалоговом окне ODBC Data Source Administrator щелкните на кнопке Add. Раскроется диалоговое окно Create New Data Source, показанное на рис. 28.24. Выберите в этом окне значение Microsoft Access Driver (*.mdb) и щелкните на кнопке Finish.
3. На экране раскроется диалоговое окно ODBC Microsoft Access 97 Setup, показанное на рис. 28.25. Введите в нем любое имя источника данных и его описание. Например, введите в поле Data Source Name значение `AccessDB`, а в поле Description — значение `DDG Test For Access.`
4. Щелкните в диалоговом окне ODBC Microsoft Access 97 Setup на кнопке Create. Раскроется диалоговое окно New Database, в котором следует ввести имя новой базы

данных и папку, в которой этот файл базы данных будет храниться. После установки этих параметров щелкните на кнопке OK. На рис. 28.25 показано диалоговое окно ODBC Microsoft Access 97 Setup после выполнения пп. 3–4. Щелкните в этом окне на кнопке OK, а затем — на кнопке OK в диалоговом окне ODBC Data Source Administrator. Теперь источник данных настроен, и можно приступить к созданию псевдонима BDE, указывающего на этот источник данных.

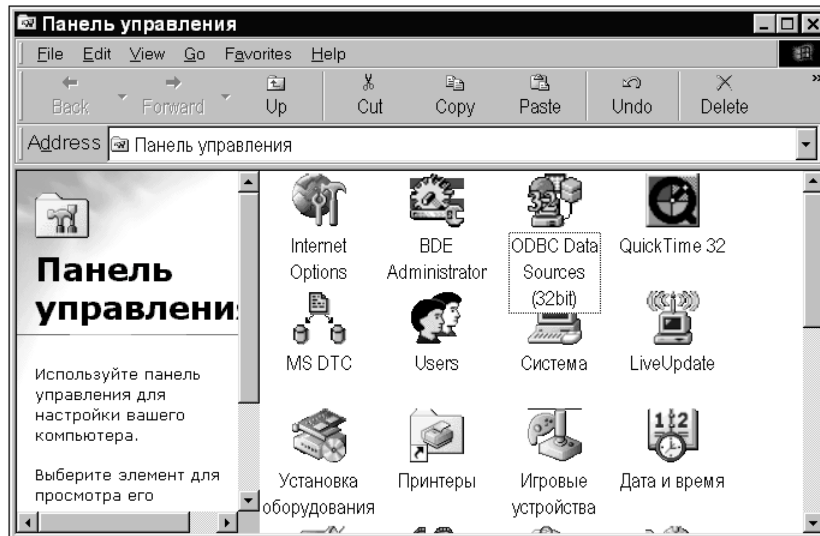


Рис. 28.22. В окне Control Panel *содержится пиктограмма* ODBC Data Source (32bit)

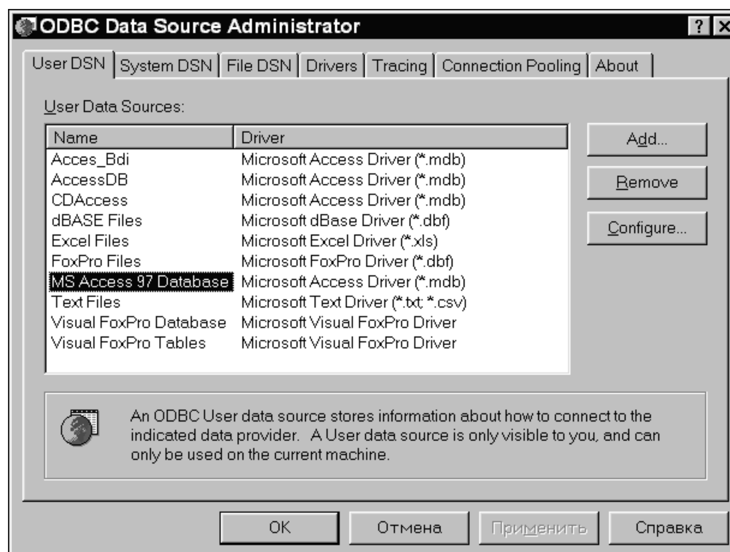


Рис. 28.23. Диалоговое окно ODBC Data Source Administrator

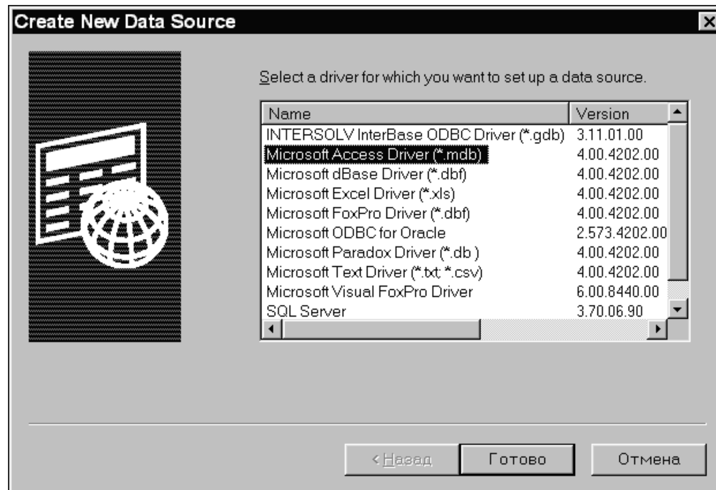


Рис. 28.24. Диалоговое окно Create New Data Source

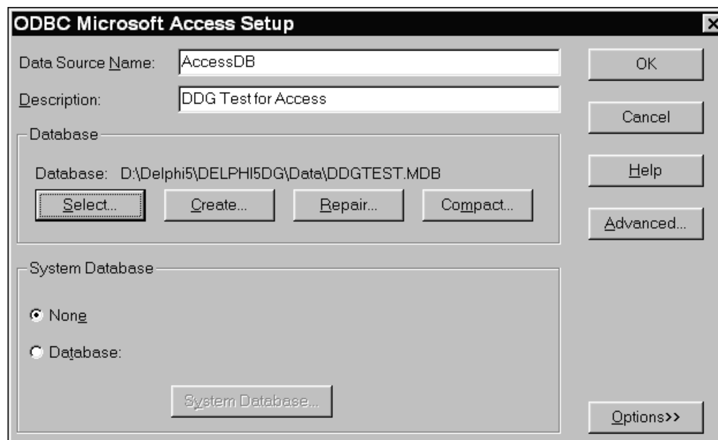


Рис. 28.25. Диалоговое окно ODBC Microsoft Access 97 Setup

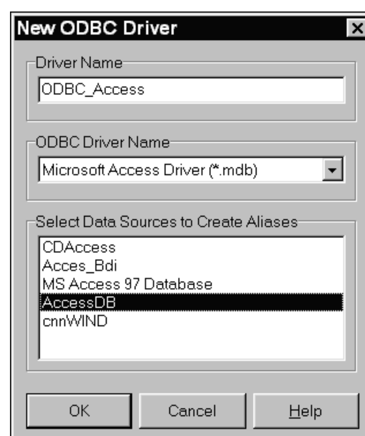


Рис. 28.26. Установленные параметры в диалоговом окне New ODBC Driver

5. Закройте все использующие BDE приложения. Запустите утилиту BDE Administrator, которая входит в состав Delphi, и перейдите во вкладку Configuration на левой панели ее окна. Разверните ветвь Drivers, щелкните правой кнопкой мыши на параметре ODBC и выберите в раскрывшемся контекстном меню команду New. На экране раскроется диалоговое окно New ODBC Driver. В его поле Driver Name может быть введено любое имя. Для нашего примера используем имя **`ODBC_Access`**. В раскрывающемся списке ODBC Driver Name выберите значение Microsoft Access Driver (*.mdb) (тот же драйвер, что и в п. 2). В списке имен источников данных автоматически должно появиться и значение `AccessDB` (имя, введенное в п. 3). Выберите эту строку. Описываемое диалоговое окно со всеми установленными параметрами будет иметь вид, показанный на рис. 28.26. Щелкните на кнопке OK, и вы вернетесь в окно BDE Administrator.
6. На левой панели окна BDE Administrator перейдите во вкладку Databases и выберите команду Object⇨New. На экране раскроется диалоговое окно New Database Alias. В этом окне выберите в списке Database Driver Name значение `ODBC_Access` (установленное в п. 5), а затем щелкните на кнопке OK. Создаваемому псевдониму можно присвоить любое имя — в данном случае введите значение **`Access`**. Рассматриваемое диалоговое окно со всеми установленными параметрами должно иметь вид, показанный на рис. 28.27. Выберите в окне BDE Administrator команду Object⇨Apply. Создание псевдонима закончено, и окно BDE Administrator теперь можно закрыть. Следующим шагом будет создание таблиц в новой базе данных.

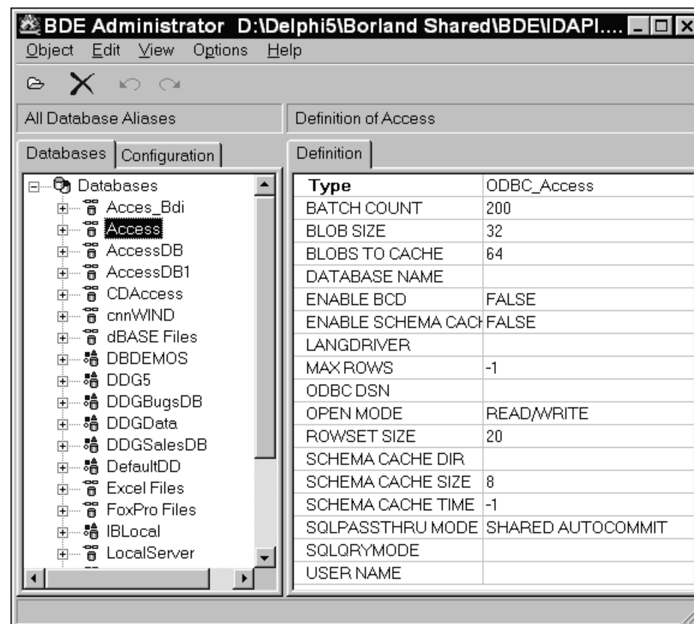


Рис. 28.27. Новый псевдоним Access в окне BDE Administrator

7. Для создания таблиц в новой базе данных Access воспользуемся утилитой Database Desktop. Запустите эту утилиту и выберите в ее окне команду File⇨New⇨Table. Появится диалоговое окно Create Table. В списке этого окна выберите в качестве типа таблицы значение `ODBC_Access` (использованное в пп. 5–6) и щелкните на кнопке OK. На экране раскроется диалоговое окно Create ODBC_Access Table.

8. Предполагается, что процедура создания таблиц в окне приложения Database Desktop вам знакома (если это не так, обратитесь к поставляемой с Delphi документации). Впрочем, создание таблицы в диалоговом окне Create ODBC_Access Table приложения Database Desktop ничем не отличается от создания таблицы в любой другой СУБД. Для демонстрационных целей поместите в новую таблицу одно поле типа `CHAR` и одно поле типа `INTEGER`. На рис. 28.28 показано диалоговое окно Create ODBC_Access Table после добавления этих двух полей.

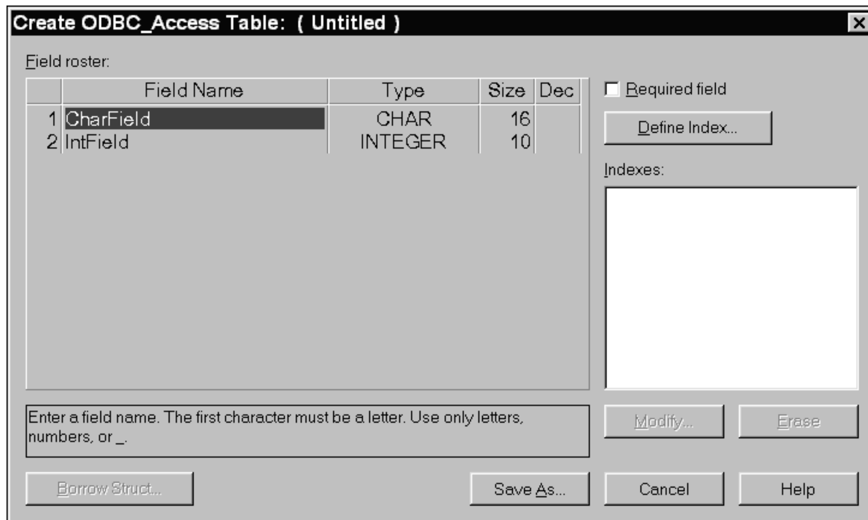


Рис. 28.28. Диалоговое окно Create ODBC_Access Table *с добавленными полями таблицы*

9. Щелкните на кнопке Save As, и на экране раскроется диалоговое окно Save Table As. В списке Alias этого окна выберите значение `Access` (псевдоним, созданный в п. 6). Раскроется диалоговое окно с предложением ввести пароль доступа к базе данных. Щелкните в этом окне на кнопке OK, поскольку имя пользователя или пароль устанавливать не нужно. Введите имя таблицы без расширения в поле File Name. В данном примере используется таблица **`TestTable`**. Щелкните на кнопке OK, и таблица будет добавлена к базе данных. Теперь все готово, чтобы получить доступ к этой базе данных из Delphi.

На заметку

Таблицы СУБД MS Access, входящие в одну базу данных, сохраняются в одном `.mdb`-файле. Этим СУБД Access отличается от Paradox и dBASE, которые сохраняют каждую таблицу в отдельном файле, как в базах данных на SQL-сервере.

10. Создайте в Delphi новый проект. Главная форма должна содержать по одному компоненту `TTable`, `TDataSource` и `TDBGrid`. Свяжите объект `DBGrid1` с объектом `Table1` с помощью компонента `DataSource1`. В свойстве `DatabaseName` объекта `Table1` выберите псевдоним `Access` (из пп. 6 и 9). Щелкните на свойстве `TableName` объекта `Table1`. Появится диалоговое окно с предложением ввести пароль. Щелкните на кнопке OK (для доступа к этой базе данных пароль не нужен), и вам будет предоставлена возможность выбрать любую доступную в базе данных `Access` таблицу. Поскольку сейчас в ней нет других таблиц, кроме только что созданной `TestTable`, вы-

берите ее. Теперь установите свойство Active объекта Table1 равным True, и имена полей этой таблицы появятся в сетке DBGrid1. Если запустить созданное приложение на выполнение, таблицу TestTable можно будет редактировать. Окно готового приложения показано на рис. 28.29.

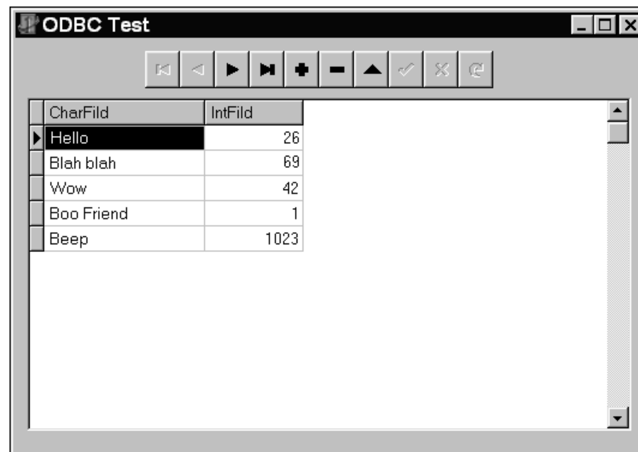


Рис. 28.29. Просмотр ODBC-таблицы в приложении Delphi

Объекты данных ActiveX (ADO)

Одним из наиболее значительных новшеств Delphi 5 является возможность прямого доступа к данным средствами технологии *объектов данных ActiveX* (ActiveX Data Objects — ADO), разработанных компанией Microsoft. Доступ к данным достигается с помощью набора новых компонентов, разработанных компанией Delphi Enterprise и получивших название *ADOExpress*. Эти компоненты находятся на странице ADO палитры компонентов. Благодаря наследованию свойств абстрактного класса TDataSet, уже обсуждавшегося выше в этой главе, компоненты ADOExpress обеспечивают подключение объектов ADO к приложению напрямую, без использования механизма BDE. Это упрощает развертывание готовых приложений у заказчика, уменьшает объем неконтролируемого разработчиками кода и повышает производительность приложения.

Кто есть кто среди стратегий доступа к данным Microsoft

Компания Microsoft за прошедшие годы разработала множество стратегий доступа к данным, поэтому не стоит особенно расстраиваться из-за того, что новая аббревиатура ADO вносит путаницу в и без того насыщенную “солянку” других сокращений типа ODBC, DAO, RDS и UDA. Чтобы систематизировать свои представления, давайте потратим некоторое время и рассмотрим подробнее весь набор терминов и аббревиатур, имеющих отношение к различным стратегиям доступа к данным, разработанным компанией Microsoft. Это позволит вам лучше понять, как технология ADO вписывается в общую картину.

- *UDA* (Universal Data Access — Универсальный доступ к данным) — это общий термин, который Microsoft использует для обозначения единой стратегии доступа к данным, включая такие ее элементы, как технологии ADO, OLE DB и ODBC. Следует отметить, что термин UDA относится не только к базам данных, но применим также к другим технологиям хранения данных, таким как службы каталогов, электронные таблицы Excel и сервер данных Exchange.
- *ODBC* (Open Database Connectivity — Открытое соединение с базами данных) — это наиболее полно разработанная технология доступа к данным компании Microsoft. Архитектура ODBC включает типовое API, построенное на использовании языка SQL, в рамках которого можно разрабатывать драйверы для доступа к конкретным базами данных. Поскольку технология ODBC успела утвердить свои позиции на рынке, в настоящее время можно найти драйверы практически для любой базы данных. А это, в свою очередь, означает, что ODBC будет интенсивно использоваться и впредь.
- *RDO* (Remote Data Objects — Удаленные объекты данных) — обеспечивает COM-оболочку для драйверов ODBC. Главная задача технологии RDO — упростить использование средств ODBC и средств поддержки удаленных ODBC-соединений в среде Visual Basic и VBA.
- *Jet* — это название механизма работы с базами данных (СУБД), встроенного в приложение Microsoft Access. Этот механизм поддерживает как собственные базы данных Access в формате MDB, так и ODBC.
- *DAO* (Data Access Objects) — еще один программный интерфейс приложений для доступа к данным, основанный на COM. DAO обеспечивает инкапсуляцию как Jet, так и ODBC.
- *ODBCDirect* — это технология Microsoft была добавлена к DAO позднее и имела целью — обеспечение прямого доступа к ODBC без необходимости промежуточной поддержки ODBC средствами Jet.
- *OLE DB* — базовая упрощенная COM-спецификация и API для доступа к данным. OLE DB проектировалась как независимая от любой конкретной СУБД архитектура и является нижним уровнем для всех новейших решений компании Microsoft в области работы с данными. Драйверы, получившие название *OLE DB-провайдеров*, могут быть созданы для подключения и организации работы посредством средств OLE DB с данными из практически любого источника.
- *ADO* (ActiveX Data Objects) обеспечивает более дружественную оболочку для базовой технологии OLE DB.
- *RDS* (Remote Data Services — Службы для удаленных данных) — это основанная на ADO технология, обеспечивающая удаленный доступ к источникам данных ADO с целью построения многоуровневых систем. Ранее технология RDS называлась ADC (Advanced Data Connector).
- *MDAC* (Microsoft Data Access Components — Компоненты Microsoft для доступа к данным) — это практическая реализация стратегии UDA. MDAC включает в себя четыре отдельные технологии: ODBC, OLE DB, ADO и RDS.

Компоненты ADOExpress

В состав ADOExpress входят шесть компонентов. Мы разделим их на три группы (по назначению): компоненты обеспечения соединения, доступа ADO и совместимости.

Компоненты обеспечения соединения

Компонент TADOConnection используется для установки соединения с хранилищем данных ADO. Один объект TADOConnection может быть связан с несколькими наборами данных и компонентами команд с целью совместного использования соединения для выполнения команд, извлечения данных и выполнения операций с метаданными. Этот компонент аналогичен компоненту TDataBase, используемому в приложениях, основанных на механизме BDE. Нет никакой необходимости использовать его в простых приложениях.

Компонент TRDSConnection инкапсулирует удаленное RDS-соединение с использованием функциональности RDS-объекта DataSpace. При работе с объектом TRDSConnection имя машины RDS-сервера необходимо указать в качестве значения его параметра ComputerName, а идентификатор ProgID RDS-сервера поместить в свойство ServerName.

Компоненты доступа ADO

Группа компонентов доступа ADO состоит из компонентов TADODataSet и TADOCommand. Эта группа получила такое название потому, что ее компоненты обеспечивают доступ к данным с использованием технологии ADO, а не традиционного механизма BDE, с которым разработчики приложений на Delphi уже хорошо знакомы.

Компонент TADODataSet — это основной компонент, используемый для извлечения данных ADO и выполнения над ними различных операций. Этот компонент позволяет оперировать с таблицами, выполнять SQL-запросы и хранимые процедуры, а также обеспечивает прямую связь с местом хранения данных или устанавливает соединение посредством компонента TADOConnection. Пользуясь терминологией библиотеки VCL, можно сказать, что компонент TADODataSet инкапсулирует функции компонентов TTable, TQuery и TStoredProc для приложений, основанных на механизме BDE.

Компонент TADOCommand используется для выполнения операторов SQL, не возвращающих результирующие наборы — подобно методам TQuery.Execute() и TStoredProc.ExecProc() в BDE-приложениях. Как и компонент TADODataSet, этот компонент может выполнять непосредственное соединение с местом хранения данных или устанавливать соединение с помощью объекта TADOConnection. Компонент TADOCommand также может использоваться для выполнения запросов SQL, возвращающих результирующий набор, но с этим набором необходимо будет оперировать с помощью компонента TADODataSet. Следующий фрагмент кода иллюстрирует перенаправление результирующего набора запроса TADOCommand компоненту TADODataSet:

```
ADODataSet.RecordSet := ADOCommand.Execute;
```

Компоненты обеспечения совместимости

Мы отнесли компоненты TADOTable, TADOQuery и TADOStoredProc к группе компонентов обеспечения совместимости, поскольку они предоставляют разработчикам средства работы с отдельными таблицами, запросами и хранимыми процедурами теми методами, которые уже хорошо им знакомы. Разработчики по своему усмотрению могут использовать либо эти компоненты, либо описанные выше компоненты прямого доступа ADO, однако при использовании компонентов этой группы несколько облегчается процесс переноса на платформу ADO приложений, основанных на использовании средств BDE. Подобно компонентам TADODataSet и TADOCommand, компоненты группы обеспечения совместимости могут напрямую связываться с источником данных или устанавливать соединение с помощью компонента TADOConnection.

Несложно догадаться, что компонент `TADOTable` используется для извлечения набора данных из одной таблицы. Компонент `TADOQuery` может применяться для получения набора данных, формируемого оператором SQL, или получаемого при выполнении команд языка определения данных *Data Definition Language* (DDL) SQL, например `CREATE TABLE`. Компонент `TADOStoredProc` используется для выполнения хранимых процедур, независимо от того, возвращают они результирующие наборы или нет.

Установка соединения с источником данных ADO

Компонент `TADOConnection`, как и каждый из компонентов группы доступа к данным ADO и группы обеспечения совместимости, имеет свойство `ConnectionString`, определяющее соединение с источником данных ADO и его атрибуты. Проще всего задать значение этого свойства с помощью редактора свойств, который вызывается щелчком на кнопке, расположенной в строке свойства в окне инспектора объектов. Общий вид диалогового окна редактора этого свойства показан на рис. 28.30.

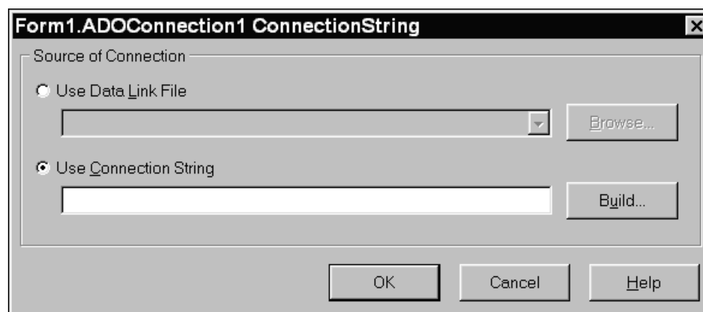


Рис. 28.30. Окно редактора свойства `ConnectString`

В этом диалоговом окне в качестве значения свойства можно выбрать либо файл связи с данными (data link file), либо строку соединения. Файл связи с данными — это файл на диске, обычно имеющий расширение `UDL`, в котором хранится строка соединения. Если требуется создать новую строку соединения, а не использовать UDL-файл, установите переключатель в положение Use Connection String (Использовать строку соединения) и щелкните на кнопке Build (Создать). Откроется окно Data Link Properties (Свойства связи с данными), показанное на рис. 28.31.

Создание UDL-файла

Если требуется создать UDL-файл, предназначенный для хранения строки описания соединения, которая будет многократно использоваться в дальнейшем, то можно воспользоваться приложением Windows Explorer, если только на вашем компьютере установлена поддержка MDAC (при установке Delphi 5 может инсталлироваться также и поддержка MDAC). Откройте окно Windows Explorer, перейдите в папку, в которой требуется создать новый UDL-файл, и щелкните в ее поле правой кнопкой мыши. Выберите в раскрывшемся контекстном меню команду New⇨Microsoft Data Link. В результате будет создан новый UDL-файл, которому можно присвоить любое имя. Щелкните правой кнопкой на пиктограмме нового UDL-файла и выберите в раскрывшемся контекстном меню команду Properties. На экране откроется окно Data Link Properties, уже описанное выше в этом разделе.

Во вкладке Provider этого диалогового окна можно выбрать драйвер OLE DB-провайдера, к которому требуется подключиться. Например, для установки соединения с драйвером ODBC средствами OLE DB следует выбрать значение `Microsoft OLE DB provider for ODBC Driver`.

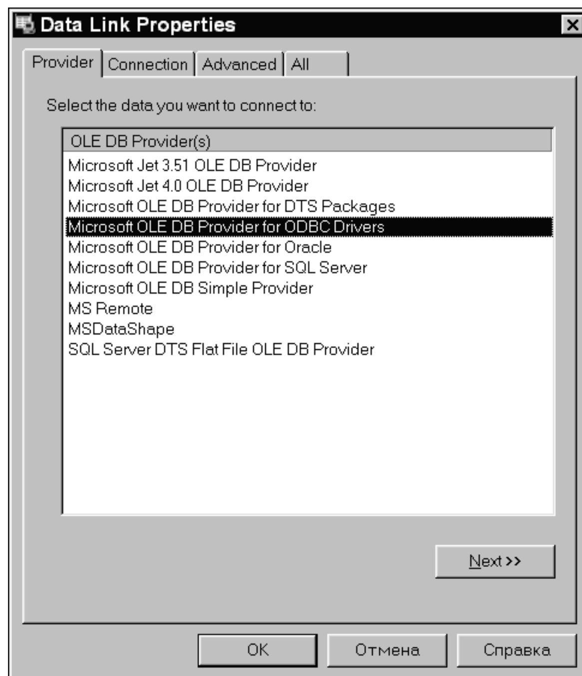


Рис. 28.31. Вкладка Provider *диалогового окна* Data Link Properties

После выбора провайдера щелкните на кнопке Next или перейдите во вкладку Connection, общий вид которой показан на рис. 28.32. В этой вкладке выполняется настройка драйвера для установки соединения с конкретной базой данных. Для примера мы подключимся к таблице dBASE, поэтому выберите значение `dBASE ODBC` в раскрывающемся списке Use Data Source Name, расположенном в первой части вкладки. Вторую часть вкладки можно пропустить, поскольку таблицы dBASE не защищены паролем. В части 3 вкладки Connection необходимо указать имя каталога, содержащего требуемые dBASE-таблицы. Для целей тестирования введите имя каталога, содержащего примеры данных компании Borland.

Чтобы удостовериться в правильности описания соединения, щелкните на кнопке Test Connection (Тестировать соединение) — и вы получите подтверждение корректности соединения либо сообщение об ошибке, сделанной при указании каталога.

Во вкладках Advanced и All диалогового окна Data Link Properties можно установить различные дополнительные свойства соединения, например, такие как Connect Timeout (Период ожидания), Access Permissions (Права доступа), Locale ID (Локальный идентификатор) и др. В рассматриваемом примере мы не будем изменять эти свойства, а воспользуемся установками, предлагаемыми по умолчанию. Щелкните на кнопке OK, а затем еще раз — в окне редактора свойств. Только что созданная строка соединения будет отображена в окне инспектора объектов, как показано на рис. 28.33.

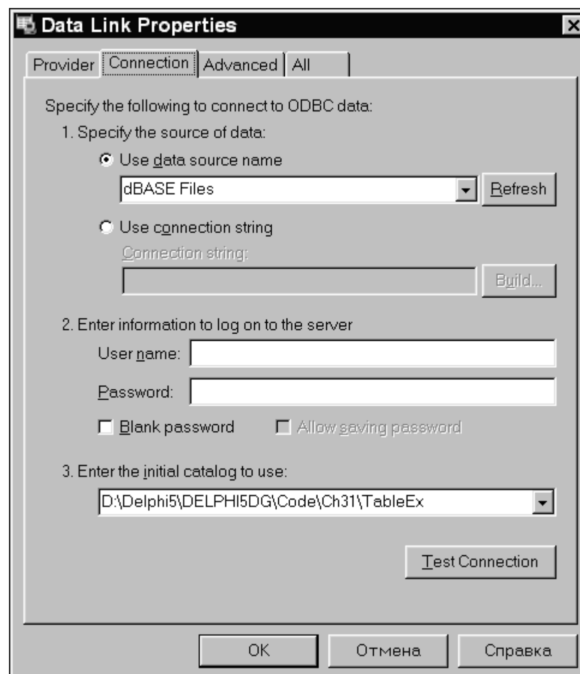


Рис. 28.32. Вкладка Connection *диалогового окна* Data Link Properties

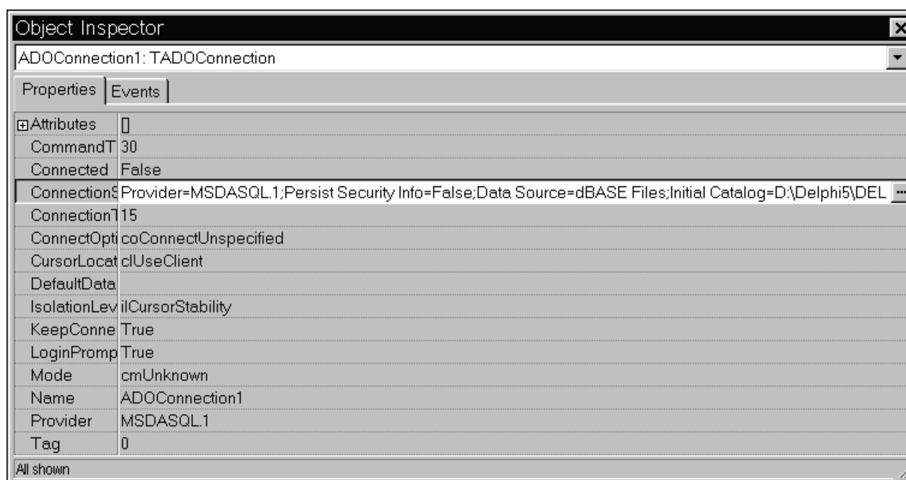


Рис. 28.33. Полный текст строки соединения отображается в значении свойства `ConnectString` *в окне инспектора объектов*

Пример: установка соединения средствами ADO

Теперь, когда вы научились создавать новую строку соединения, можно считать, что самая сложная часть пути доступа к данным средствами ADO пройдена. На следующем этапе мы осуществим в Delphi просмотр данных, полученных с помощью только что описанного соединения. Для этого понадобится лишь компонент `TADODataSet`. Выполните описанные выше действия по установке значения свойства `ConnectString` компонента `TADODataSet`. Затем, с помощью редактора свойств, создайте в свойстве `CommandText` оператор SQL, позволяющий просматривать содержимое таблицы — как показано на рис. 28.34). Щелкните на кнопке `OK` для закрытия диалогового окна.

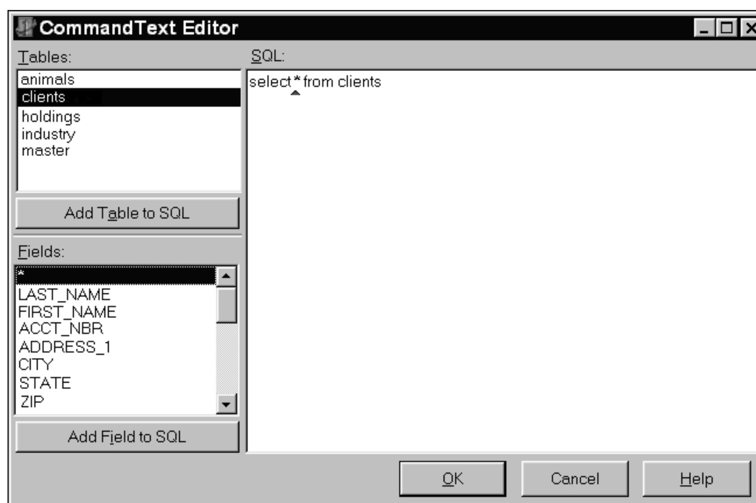


Рис. 28.34. Редактирование значения свойства `CommandText`

После задания свойства `CommandText` можно присвоить свойству `Active` компонента `TADODataSet` значение `True`. Теперь с помощью этого компонента можно осуществлять активный просмотр данных. Для этого необходимо поместить в приложение компонент `TDataSource`, подключив его к объекту `TADODataSet`, и компонент `TDBGrid`, подключив его к объекту `TDataSource`, как было описано выше в этой главе. Полученный результат представлен на рис. 28.35.

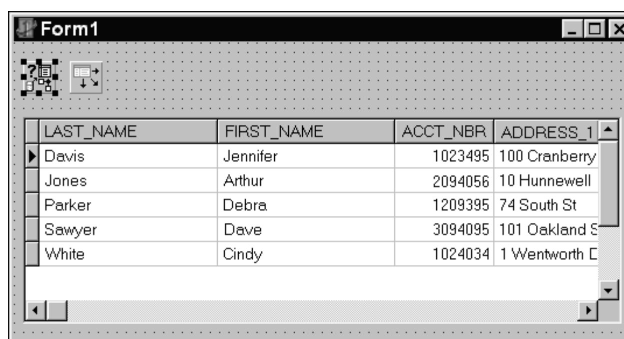


Рис. 28.35. Доступ к данным посредством использования компонента `TADODataSet`

Установка ADO

Для работы приложений, основанных на концепции ADO, под управлением Windows 95, Windows 98 и Windows NT в целевой системе необходимо установить пакет MDAC. Установочные файлы находятся в каталоге \MDAC на прилагаемом компакт-диске. Комплект поставки Windows 2000 включает пакет MDAC, поэтому для работы ADO-приложений под управлением Windows 2000 отдельно устанавливать пакет MDAC не нужно.

Резюме

Авторы полагают, что, прочитав эту главу, вы сможете программировать в Delphi любые приложения, работающие с различными СУБД, не поддерживающими язык SQL. В этой главе был рассмотрен базовый компонент TDataSet и производные от него компоненты TTable, TQuery и TStoredProc. Мы также обсудили, как оперировать с объектами TTable, управлять полями и работать с текстовыми таблицами. Помимо этого мы познакомили вас с различными стратегиями доступа к данным, включая BDE, ODBC и ADO. Как видите, библиотека VCL предоставляет в ваше распоряжение удобную объектно-ориентированную оболочку для доступа к процедурам BDE, а также возможности ее согласования с другими технологиями, например, такими как ADO.

В следующей главе вашему вниманию будет предложено более подробное описание технологии клиент/сервер и использование компонентов TQuery и TStoredProc в распределенной среде.

Глава

29 Разработка приложений архитектуры клиент/сервер

Почему именно клиент/сервер? 605

Архитектура клиент/сервер 606

Модели клиент/сервер 610

Архитектура клиент/сервер и локальные базы данных 611

Язык SQL и его роль в технологии клиент/сервер 614

Разработка приложений клиент/сервер в Delphi 614

Разработка серверной части 615

Разработка клиентской части 627

Резюме 649

Что же стоит за этим модным словосочетанием “клиент/сервер”? Складывается впечатление, что сегодня все или уже используют, или разрабатывают какую-либо систему этого типа, охватывающую всю организацию. Поэтому стоит уделить время изучению технологии клиент/сервер и узнать, что она собой представляет и какие новые возможности предоставляет разработчикам и приложениям.

Если вы являетесь разработчиком на Delphi, то углубленные рассуждения на тему “клиент/сервер” едва ли покажутся вам излишними. Ведь Delphi 5 — это, прежде всего, среда разработки приложений клиент/сервер. Однако это вовсе не означает, что любые приложения, разрабатываемые с помощью Delphi, относятся к этому типу. Более того, даже те приложения, которые получают доступ к данным, расположенным в базах данных архитектуры клиент/сервер, например, такие как Oracle, Microsoft SQL или InterBase, совсем не обязательно принадлежат к типу клиент/сервер.

В этой главе речь пойдет о тех элементах, которые составляют систему клиент/сервер. Мы проведем сравнение методов разработки приложений архитектуры клиент/сервер и традиционной архитектуры для баз данных на персональных компьютерах или мейнфреймах. Здесь же приводится упрощенный пример, с помощью которого обосновывается необходимость использования решений архитектуры клиент/сервер. Кроме того, мы рассмотрим процесс разработки трехуровневых приложений клиент/сервер в Delphi 5 и обсудим некоторые типичные ошибки, совершаемые разработчиками приложений локальных баз данных при переходе в распределенную среду приложений клиент/сервер.

Почему именно клиент/сервер?

Вот типичный пример, когда использование технологии “клиент/сервер” может оказаться оптимальным. Предположим, что вы ответственны за создание приложения масштаба подразделения, которое обращается к данным, размещенным в локальной сети или на файловом сервере. В пределах подразделения с этим приложением будут работать многие пользователи. Поскольку обрабатываемые данные находят в подразделении все более широкое применение, для работы с ними будут создаваться и другие приложения.

Предположим, что по мере накопления данные начинают представлять интерес и для других подразделений в пределах вашей организации. В этих подразделениях понадобится создать свои дополнительные приложения. Это, скорее всего, потребует переместить информацию на сервер базы данных, что позволит сделать их доступными более широкому кругу пользователей. С течением времени интерес к обрабатываемой информации возрастает и в масштабах всей организации в целом. Она привлекает внимание создателей приложений для поддержки принятия решений, и теперь важно, чтобы используемая архитектура обеспечивала не только быстрый доступ к данным, но и позволяла извлекать их в необходимом для подобных приложений виде.

Глобальная доступность подобных данных приводит к появлению некоторых проблем, унаследованных от классической архитектуры доступа к данным в сетях персональных компьютеров с локальными базами данных. Самые важные из них — чрезмерная сетевая нагрузка (трафик является узким местом при получении данных) и защита данных.

Приведенный выше пример хотя и весьма упрощен, но иллюстрирует ситуацию, в которой появляется реальная необходимость использования решения клиент/сервер. Такое решение обладает целым рядом преимуществ.

- Допускает независимый групповой доступ к данным, что позволяет отдельным подразделениям обрабатывать только ту часть данных, за которую они ответственны.
- Обеспечивает эффективный доступ к данным лицам, ответственным за принятие решений, позволяя извлекать информацию в наиболее удобном для них виде.

- Повышает возможности централизованного управления многоуровневой моделью с целью обеспечения целостности данных, что позволяет уменьшить остроту проблемы централизованного контроля и анализа использования данных.
- Устанавливает правила целостности данных для всей базы данных.
- Обеспечивает более высокий уровень разделения труда между клиентом и сервером (каждый выполняет задачи, для решения которых он более приспособлен).
- Позволяет использовать улучшенные средства поддержки целостности данных, предоставляемые большинством типов серверов баз данных.
- Уменьшает нагрузку на сеть, поскольку клиенту возвращаются подмножества данных, а не целиком все таблицы, как в случае локальных баз данных.

Этот список можно было бы продолжить. По мере чтения этой главы вы узнаете и о других преимуществах, достигаемых за счет перехода к архитектуре клиент/сервер.

Следует отметить, что использование технологии клиент/сервер не всегда бывает оправданным. Разработчик должен провести полный анализ требований к системе, прежде чем решить, является ли технология клиент/сервер именно тем средством, которое необходимо для решения поставленных задач. Кроме того, следует учесть, что системы клиент/сервер весьма дорогостоящи. В эту стоимость входит сетевое программное обеспечение, серверная операционная система, сервер базы данных и аппаратные средства, соответствующие программному обеспечению этого уровня. Не стоит забывать и о затратах времени и средств на обучение пользователей работе с серверной операционной системой и программным обеспечением базы данных.

Архитектура клиент/сервер

Типичная архитектура клиент/сервер предусматривает наличие конечного пользователя (*клиента*), который имеет доступ и возможность обрабатывать данные, сохраняемые на удаленном компьютере — *сервере*. Не существует никакого стандартного определения того, что такое клиент и чем занимается сервер. Однако можно с уверенностью полагать, что сервер предоставляет некоторый сервис, а клиент запрашивает его у сервера. К одному и тому же серверу может обращаться множество клиентов с требованием предоставить им какой-либо сервис, и именно сервер решает, как обработать подобные запросы. Кроме того, в системе клиент/сервер может существовать и третий элемент. Об этом речь пойдет далее в этой главе при рассмотрении трехуровневых систем.

В среде клиент/сервер последний играет намного б*о*льшую роль, чем роль простого распределителя данных. Фактически именно сервер выполняет основную часть работы системы. Он управляет тем, как клиент будет получать доступ и обрабатывать данные. Реально клиентские приложения являются лишь средством представления данных пользователю и получения их от него. Далее в этой главе обязанности клиента и сервера рассматриваются подробнее. Кроме того, речь пойдет о *бизнес-правилах*, которые определяют способ доступа клиента к данным сервера.

Клиент

Клиенты представляют собой приложения, обеспечивающие графический или не графический интерфейс с пользователем. Delphi 5 позволяет разрабатывать как чисто клиентские приложения, так и серверы среднего уровня в трехуровневых моделях. Приложения-серверы баз данных предпочтительнее разрабатывать с использованием мощных распределенных СУРБД, подобных Oracle или InterBase.

Клиентские приложения предоставляют пользователю интерфейс для управления данными на сервере. Именно через клиентское приложение пользователь получает доступ к функциональным возможностям сервера. Примером запрашиваемых действий может быть добавление заказчика, счета или печать отчета. В этом случае клиент просто посылает запрос и предоставляет необходимые для его выполнения данные. Сервер же несет ответственность за обработку запроса. Это не означает, что клиент не может выполнять каких-либо логических действий самостоятельно. Вполне возможно, что клиент реализует бо́льшую часть (если не всю) поддержки бизнес-логики приложения. Такое приложение называется *толстым клиентом* (fat client).

Масштабируемые приложения

Возможно, вам не раз приходилось встречаться с термином *масштабируемость*, применяемом в отношении разработки приложений архитектуры клиент/сервер в Delphi. Что же такое масштабируемость? Иногда под этим подразумевается всего лишь возможность простого доступа к базе данных на сервере с помощью мощных средств доступа к данным, используемым в Delphi. Также это означает, что при стремительном увеличении числа пользователей и количества их обращений к серверам потери производительности распределенной системы будут минимальны. В других случаях подразумевается, что локальное приложение можно превратить в клиентское путем простого изменения в нем свойства `Alias`. К сожалению, последнее не совсем верно. Естественно, можно изменить свойство `Alias` и тут же получить доступ к данным в базе на сервере. Однако этого не достаточно для превращения приложения в настоящее клиентское приложение и действие это ни как не связано с понятием масштабируемости. Основное преимущество систем клиент/сервер состоит в том, что они предоставляют доступ к мощным функциональным возможностям сервера. Однако невозможно получить какие-либо преимущества от использования этих функций в тех приложениях, которые проектировались для работы с локальными базами данных.

Сервер

Сервер предоставляет сервис клиенту. Он, по существу, ждет, пока клиент сделает запрос, а затем обрабатывает этот запрос. Сервер должен обладать способностью обрабатывать одновременно несколько запросов от нескольких клиентов, а также уметь распределять эти запросы по приоритетам. Чаще всего серверная программа работает постоянно, обеспечивая не прекращающийся доступ к ее услугам.

На заметку

Клиент и сервер не обязательно должны располагаться на разных компьютерах. Зачастую фоновые задачи обработки данных для клиентов выполняются на том же компьютере, где работает сервер.

Бизнес-правила

Бизнес-правила — это процедуры управления, которые определяют, как клиент должен получать доступ к данным на сервере. Эти правила реализуются программным текстом клиента, сервера или ими обоими. В Delphi 5 бизнес-правила реализуются в программах на языке Object Pascal. На стороне сервера бизнес-правила реализуются в виде хранимых процедур SQL, триггеров и других объектов, присущих серверным базам данных. В трехуровневой модели бизнес-правила могут быть реализованы на среднем уровне. Эти объекты рассматриваются ниже в данной главе.

Важно понимать, что бизнес-правила определяют поведение всей системы. При их отсутствии у вас есть просто данные на одном компьютере и приложение с графическим интерфейсом на другом, однако нет метода их соединения.

На определенном этапе разработки вашей системы необходимо решить, какие процессы должны будут в ней существовать. Например, рассмотрим систему складского учета. Ей свойственны следующие типичные процессы: прием заказа, печать накладной, добавление заказчика и т.п. Как указывалось выше, все правила выполнения этих операций реализуются в коде Object Pascal на стороне клиента или на среднем уровне. Эти бизнес-правила также могут располагаться в виде SQL-кода на сервере или представлять собой комбинацию всех трех вариантов. Если бо́льшая часть правил реализована на сервере, то его называют "толстым сервером". Если правила реализуются в основном на стороне клиента, он называется "толстым клиентом". Если правила реализованы на среднем уровне, то сервер также называют "толстым". То, на какой именно стороне реализуются бизнес-правила приложения, определяется объемом и типом требуемых операций управления данными.

На заметку

В литературе наряду с термином *трехуровневая* система иногда встречаются термины *n-уровневая* или *многоуровневая* (multitier) система, которые порой употребляются некорректно. В трехуровневой модели обычно существует один или более клиентов, бизнес-логика и сервер базы данных. Поддержка бизнес-логики может быть разделена на несколько частей, выполняемых на различных компьютерах или даже на нескольких серверах приложений. Не кажется ли вам абсурдом упоминание 10-, 15- или даже 25-уровневых систем? Мы предпочитаем рассматривать всю поддержку бизнес-логики как один средний уровень, независимо от количества подпрограмм и серверных приложений, потребовавшихся для ее реализации.

Размещение программ бизнес-правил

Решение о том, где должны выполняться программы бизнес-правил или как распределить их реализацию между сервером и клиентами, зависит от нескольких факторов. В частности, примерами этих факторов могут быть требования защиты данных и обеспечения их целостности, необходимость централизованного управления и правильное распределение работы.

Защита данных

Защита данных приобретает важное значение в том случае, если необходимо обеспечить ограниченный доступ к различным частям данных или к различным операциям, которые могут быть выполнены над ними. Это реализуется посредством назначения привилегий доступа на уровне пользователей к различным объектам базы данных, таким как представления и хранимые процедуры. Более подробно эти объекты рассматриваются несколько ниже. Используя привилегии доступа к объектам базы данных, можно ограничить возможности доступа пользователя только теми элементами данных, которые ему действительно необходимы. Сведения о привилегиях и хранимые процедуры хранятся на сервере.

Существует одна очень важная концепция, которую необходимо запомнить, — базы данных архитектуры клиент/сервер должны разрабатываться таким образом, чтобы к ним могли иметь доступ многие клиентские приложения и инструментальные средства. Хотя доступ к данным может быть ограничен непосредственно в логике используемых клиентских приложений, ничто не помешает пользователю прибегнуть к другим средствам просмотра или редактирования таблицы внутри вашей базы данных. Сделав доступ к базе данных возможным только с помощью представлений и хранимых процедур, можно предотвратить несанкционированный доступ к данным. Это ограничение доступа играет важную роль также в поддержке целостности данных, рассматриваемой в следующем разделе.

Целостность данных

Под *целостностью данных* (data integrity) понимается корректность и согласованность данных на сервере. Если не принять необходимых мер защиты, данные могут быть повреждены. Примерами нарушения целостности данных может быть добавление в заказ несуществующих или проданных товаров, изменение количества товаров в заказе без пересчета их общей стоимости или удаление заказчика с несогласованным балансом.

Как же обеспечить целостность данных? Один из способов — ограничение действий, которые могут выполняться над данными, за счет применения хранимых процедур. Другой способ — размещение бо́льшей части бизнес-логики на сервере или на среднем уровне. Предположим, например, что в системе складского учета существует клиентское приложение, которое содержит основную часть бизнес-логики. В этом клиентском приложении процедура удаления заказчика достаточно интеллектуальна, чтобы обратиться к данным сервера с запросом, не имеется ли у заказчика отрицательного баланса. Это очень полезная функция клиентского приложения. Однако, поскольку она присуща только данному клиенту, а не серверу, не существует преград для запуска другого приложения или стандартного инструмента доступа к базе данных, позволяющего удалить заказчика непосредственно из таблицы. Чтобы предотвратить такую ситуацию, следует запретить всем пользователям доступ к таблице заказчиков, а затем создать хранимую процедуру на сервере. Она обеспечит выполнение операции удаления только после выполнения необходимых проверок. В результате, поскольку прямой доступ к таблице запрещен, все пользователи вынуждены будут обращаться к этой хранимой процедуре.

Это только один из способов поддержки целостности данных, возможных при реализации бизнес-правил на сервере. Подобные действия по защите целостности данных могут выполняться путем размещения операций проверки в триггерах или с помощью предоставления пользователю только необходимых ему данных. Важно не забывать, что данные должны располагаться на сервере таким образом, чтобы различные подразделения могли получать к ним доступ посредством различных приложений. Чем больше бизнес-правил реализуется на сервере, тем шире возможности по управлению защитой данных.

Централизованное управление данными

Еще одно преимущество размещения бизнес-логики на сервере или в отдельном уровне при трехуровневой модели заключается в том, что многоуровневая система позволяет изменять бизнес-логику, не оказывая влияния на работу клиентских приложений. Это означает, что добавление дополнительного кода к хранимым процедурам будет прозрачно для программы-клиента во всех случаях, когда оно не вносит прямых изменений в интерфейс между клиентом и сервером. Это упрощает обслуживание многоуровневой модели и дает дополнительные преимущества на уровне всей компании.

Распределение работ

Размещая бизнес-правила на сервере или на различных средних уровнях, многоуровневая система упрощает выполнение задач разделения обязанностей между подразделениями, одновременно гарантируя необходимую целостность и защищенность данных сервера. Это позволяет разным подразделениям совместно использовать одни и те же данные, но ограничивает их доступ к информации только теми данными, которые необходимы этим подразделениям для выполнения их специфических задач. Такое распределение работы осуществляется благодаря предоставлению доступа только к тем хранимым процедурам и другим объектам базы данных, которые необходимы конкретному подразделению.

Обратимся к системе складского учета еще раз. Для бо́льшей определенности предположим, что это — система учета склада автомобильных запасных частей. В данной системе некоторым сотрудникам необходимо обращаться к одним и тем же данным, но с различными целями. Кассир

должен иметь возможность обрабатывать заказы, добавлять и удалять заказчиков и изменять сведения о заказчике. Складской персонал должен иметь возможность добавлять в базу данных наименования новых запасных частей, а также заказывать новые партии запасных частей. Сотрудники бухгалтерии должны иметь возможность выполнять свою часть работы с системой. Мало вероятно, что складскому персоналу потребуется выдавать финансовый отчет за месяц. Точно так же мало вероятно, что бухгалтеру потребуется изменять личные данные заказчика. Создавая все необходимые бизнес-правила на сервере, можно предоставить доступ каждому работнику и подразделению только к определенным данным. В этом случае доступ кассира к данным будет определяться одними правилам, доступ складского персонала — другими, а сотрудники бухгалтерии получат доступ только к данным, связанным с учетом денежных средств.

Распределение работ относится не только к их разделению между различными клиентами, но и к определению того, какую работу лучше выполнять на стороне клиента, а не на сервере или на среднем уровне. Как разработчик, вы должны оценить различные варианты распределения работы. Например, на быстрых клиентских компьютерах можно выполнять операции, интенсивно использующие процессор, что позволит несколько разгрузить сервер. Естественно, прежде чем принимать решение, необходимо продумать, какие бизнес-правила могут оказаться при этом нарушенными и не появятся ли при таком подходе бреши в системе защиты.

Модели клиент/сервер

Вероятно, вы часто слышали о системах клиент/сервер, относящихся к одной из двух моделей. Это двухуровневая и трехуровневая модели, показанные на рис. 29.1 и 29.2 соответственно.

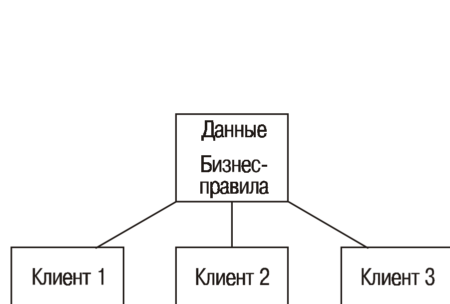


Рис. 29.1. Двухуровневая модель клиент/сервер

Рис. 29.2. Трехуровневая модель клиент/сервер

Двухуровневая модель

На рис. 29.1 представлена схема *двухуровневой модели* клиент/сервер. Эта модель, возможно, наиболее общая, поскольку она следует схеме построения локальных баз данных. Многие из уже существующих ныне систем клиент/сервер развивались из приложений, использовавших локальные базы данных, размещенных в сетях персональных компьютеров на совместно используемых файловых серверах. Перенос на SQL-серверы систем, основанных на совместно используемых файлах Paradox или dBASE, осуществлялся с целью повышения эффективности их работы, защищенности и надежности базы данных.

В такой модели данные постоянно находятся на сервере, а клиентские приложения — на компьютере пользователя. Бизнес-правила при этом могут располагаться на любом из компьютеров (или даже на обоих одновременно).

Трехуровневая модель

На рис. 29.2 показана *трехуровневая модель* клиент/сервер. Здесь клиент — это пользовательский интерфейс доступа к данным. Данные находятся на удаленном сервере. Клиентское приложение делает запросы для получения доступа или изменения данных через сервер приложений или сервер Remote Data Broker (Удаленный брокер-сервер данных) — RDB. Обычно бизнес-правила располагаются именно на сервере RDB.

Если клиент, сервер и бизнес-правила распределены по отдельным компьютерам, разработчик может оптимизировать доступ к данным и поддерживать их целостность при обращении к ним из любых приложений во всей системе. Благодаря поддержке технологии MIDAS, Delphi 5 предоставляет мощные средства разработки трехуровневой архитектуры.

MIDAS — Пакет служб поддержки многоуровневых распределенных приложений

Технология MIDAS (Multitier Distributed Application Services Suite) компании Borland включена лишь в комплект поставки Delphi 5 Enterprise. Она представляет собой набор специализированных компонентов, серверов и технологий для разработки трехуровневых приложений. В главе 32, “Разработка приложений MIDAS”, эта технология рассматривается более подробно.

Архитектура клиент/сервер и локальные базы данных

Если ранее вы занимались созданием приложений, работающих с локальными базами данных, то для вас очень важно понять различия в подходе, существующие между работой с локальными базами данных и базами данных в среде клиент/сервер. В этом разделе рассматриваются наиболее существенные отличия между этими двумя типами разработки.

Работа с наборами данных и с отдельными записями

Чаще всего неправильно понимается утверждение, что работа с базой данных в среде клиент/сервер строится на обработке *наборов данных*, а не отдельных *записей данных*. Имеется в виду, что клиентские приложения работают не с самими таблицами данных, как в случае локальных баз данных, а с некоторыми подмножествами (наборами) данных.

Суть заключается в том, что клиентское приложение запрашивает у сервера строки, которые состоят из полей одной или нескольких таблиц. Эти запросы создаются с помощью языка SQL (Structured Query Language — Структурированный язык запросов).

Используя команды языка SQL, клиенты тем самым ограничивают число записей, которые возвращает им сервер. Клиенты используют операторы SQL для выполнения запроса к серверу с целью получения набора результатов, который может состоять из весьма ограниченного подмножества данных сервера. Обратите особое внимание на этот важный момент — при доступе к локальным базам данных вызываемому приложению по сети отправляется вся таблица. Чем больше таблица, тем больше нагрузка на сеть. Это существенно отличается от среды клиент/сервер, в которой по сети пересылаются только затребованные записи, что снижает нагрузку на сеть.

Это отличие влияет на возможности перемещения по полученному набору данных SQL. Такие понятия, как первая, последняя, следующая или предыдущая запись не применимы к наборам данных SQL. Особенно справедливо это замечание в случае наборов данных, записи

которых составлены из полей нескольких таблиц. Многие SQL-серверы предоставляют “прокручиваемые курсоры”, которые позволяют перемещать указатель текущей записи в результирующем наборе данных SQL. Однако это далеко не те возможности перемещения, которые доступны в обычной таблице локальной базы данных. Ниже в этой главе, в разделе “Какой компонент использовать — TTable или TQuery?”, будет показано, как эти различия в концепциях влияют на разработку клиентских приложений в Delphi 5.

Защита данных

Организация защиты данных SQL-серверов отличается от защиты локальных баз данных. Хотя при этом для общего доступа к базе данных могут применяться одинаковые способы защиты с помощью пароля, в среде SQL существует также механизм ограничения доступа пользователя к специфическим объектам базы данных, например к таблицам, представлениям, хранимым процедурам и т.д. Более детально эти объекты рассматриваются несколько позже, а пока важно понять, что возможности доступа к данным могут быть определены на сервере исходя из потребностей конкретного пользователя.

Обычно базы данных SQL позволяют предоставлять (grant) или отменять (revoke) привилегии отдельным пользователям или группам пользователей. Следовательно, в таких базах данных можно определять группы пользователей и предоставлять им привилегии по отношению к любому из рассмотренных выше объектов базы данных.

Методы блокировки записей

Блокировка — это механизм, используемый для обеспечения одновременного доступа нескольких SQL-транзакций различных пользователей к одной и той же базе данных. Существует несколько уровней блокировки и разные серверы отличаются используемыми уровнями блокировки данных.

Блокировка *на уровне таблицы* запрещает модифицировать таблицы, участвующие в выполняющихся в настоящее время транзакциях. Хотя этот метод допускает параллельную обработку, он достаточно медленный, поскольку пользователям чаще всего приходится обращаться к одним и тем же таблицам.

Более эффективная методика — блокировка *на уровне страницы*. В этом случае сервер блокирует некоторые блоки данных на диске — так называемые *страницы*. В то время, когда одна транзакция выполняет операцию на некоторой странице, для других транзакций возможности модификации данных этой же страницы ограничиваются. Обычно данные располагаются на сотнях страниц, так что обращение нескольких транзакций к одной и той же странице случается нечасто.

Некоторые серверы предлагают блокировку *на уровне записи*, т.е. отдельной строки в таблице базы данных. Однако этот метод связан с большими затратам ресурсов для хранения информации о блокировках.

Локальные базы данных используют *пессимистическую*, или *детерминированную* блокировку. Это означает, что существует ограничение на внесение изменений в записи таблицы, которые в настоящее время изменяются другим пользователем. При попытке обратиться к такой записи появится сообщение об ошибке, указывающее, что вы не можете обрабатывать запись, пока предыдущий пользователь не освободит ее.

В базах данных SQL-серверов используется концепция, известная как *оптимистическая* блокировка. При этом доступ пользователя к записи, к которой уже обратился другой пользователь, не ограничивается. Он может редактировать ее и затем обратиться к серверу для сохранения этой записи. Однако, прежде чем запись будет сохранена, происходит сравнение ее

с копией на сервере, которая, возможно, модифицировалась другим пользователем в то самое время, когда первый пользователь просматривал или редактировал ее у себя. В этом случае первый пользователь получит сообщение об ошибке, указывающее, что запись изменялась с тех пор, как он ее получил. При проектировании клиентского приложения разработчик должен всегда помнить о таких ситуациях и корректно реагировать на них (заметьте, что в локальной базе данных подобная ситуация невозможна).

Целостность данных

При работе с базами данных SQL-серверов можно использовать более жесткие ограничения на целостность данных сервера. Хотя локальные базы данных и имеют встроенные средства поддержки ограничений целостности данных, все бизнес-правила приходится определять в коде клиентского приложения. Базы данных SQL позволяют устанавливать эти правила на стороне сервера. При этом основное преимущество заключается в том, что все клиентские приложения будут использовать один и тот же набор бизнес-правил, а кроме того, они будут централизованы.

Ограничения на целостность данных определяются при создании таблицы на сервере. Некоторые примеры ограничений, такие как проверка корректности, уникальность и ссылочные ограничения, будут рассматриваться ниже в этой главе.

Как отмечалось выше, ограничения целостности можно определить и в хранимой процедуре языка SQL. Например, перед обработкой заказа можно выяснить, установлена ли для заказчика максимальная сумма кредита. Далее мы покажем, как такие правила способствуют сохранению целостности данных.

Транзакции

Базы данных SQL-серверов ориентированы на использование *транзакций*. Это означает, что изменения данных не заносятся прямо в таблицы, как в локальных базах данных. Вместо этого клиентское приложение обращается к серверу с запросом на выполнение требуемых изменений, а сервер реализует запрошенный пакет операций как одну транзакцию.

Для того чтобы любое изменение данных могло быть зафиксировано, транзакция должна быть завершена целиком. Если любая из операций внутри транзакции не будет выполнена, происходит *откат* всей этой транзакции, т.е. транзакция прерывается и восстанавливается состояние данных, в котором они находились до ее начала.

Транзакции обеспечивают согласованность данных на сервере. Вернемся к нашему примеру складского учета. Когда заказ принят, в таблицу `ORDER` должны быть внесены изменения, отражающие этот заказ. Кроме того, в таблице `PARTS` количество заказанных запасных частей должно уменьшиться на число, указанное в заказе. Если по некоторым причинам между модификациями таблиц `ORDER` и `PARTS` произошел сбой системы, данные будут неправильно отражать фактическое число имеющихся запасных частей. При инкапсуляции этой операции внутри транзакции ни одна из таблиц не будет изменена, пока вся транзакция не будет выполнена.

В приложениях Delphi 5 транзакциями можно управлять на уровне сервера или на уровне клиента. Это положение будет проиллюстрировано ниже в этой главе, в разделе “Управление транзакциями”.

На заметку

Некоторые локальные базы данных, такие как Paradox 9, также могут поддерживать транзакции.

Язык SQL и его роль в технологии клиент/сервер

Язык SQL — это промышленный стандарт набора команд для манипулирования базой данных, который используется средой разработки приложений типа Delphi. SQL — это не язык программирования в обычном понимании, т.е. вы не сможете зайти в компьютерный магазин и купить коробку с транслятором с языка SQL. Поддержка языка SQL является частью серверной базы данных.

Язык SQL получил большое распространение как язык запросов баз данных в 80-х и 90-х годах, а сегодня он стал стандартом для работы с базами данных клиент/сервер в сетевой среде. Delphi позволяет работать с языком SQL посредством предоставляемых им компонентов. К преимуществам языка SQL можно отнести широкие возможности просмотра данных и гибкие средства их обработки, не доступные обычным интерфейсам, ориентированным на записи.

Язык SQL позволяет управлять данными сервера посредством следующих предоставляемых им функциональных возможностей.

- *Определение данных*. С помощью команд SQL определяется структура таблиц, т.е. типы данных полей внутри таблиц, а кроме того, ссылочные отношения определенных полей к полям в других таблицах.
- *Выборка данных*. Клиентские приложения используют команды SQL для запроса у сервера необходимых им данных. Кроме того, язык SQL позволяет клиентам определять, какие данные выбрать и в каком виде их предоставить, как эти данные отсортировать и какие поля включить в возвращаемую информацию.
- *Целостность данных*. Язык SQL позволяет поддерживать целостность данных с помощью различных ограничений, либо определенных как часть таблицы, либо сохраняемых независимо от таблицы — в виде хранимых процедур или других объектов базы данных.
- *Обработка данных*. Язык SQL позволяет клиентским приложениям модифицировать, добавлять или удалять данные с сервера. Эти действия могут либо описываться отдельными операторами SQL, передаваемыми серверу, либо осуществляться хранимой процедурой, сохраняемой на сервере.
- *Защита*. Язык SQL позволяет защищать данные посредством определения привилегий доступа пользователя, создания представлений и организации ограниченного доступа к различным объектам базы данных.
- *Конкурентный доступ*. Язык SQL управляет конкурентным доступом к данным таким образом, что пользователи, одновременно работающие с системой, не оказывают влияния на работу друг друга.

Такое обилие возможностей позволяет сделать вывод, что язык SQL является главным инструментом разработки и манипулирования данными в приложениях клиент/сервер.

Разработка приложений клиент/сервер в Delphi

Как же Delphi 5 вписывается в концепции архитектуры клиент/сервер? Delphi 5 предлагает компоненты объектов баз данных, инкапсулирующие функциональные возможности Borland Database Engine (BDE). Благодаря чему стало возможным создавать приложения

баз данных, не зная всех функций BDE. Кроме того, компоненты доступа к базе данных связываются с компонентами для работы с данными. Это упрощает создание пользовательского интерфейса приложений баз данных. SQL-связи обеспечиваются с помощью драйверов таких серверов, как Oracle, Sybase, Informix, Microsoft SQL Server, DB2 и InterBase. Кроме того, доступ к данным из других баз данных можно получить также с помощью средств ODBC и ADO. В последующих разделах для иллюстрации различных методов создания приложений клиент/сервер используются база данных клиент/сервер InterBase и компоненты баз данных Delphi 5.

Delphi 5 поддерживает технологию MIDAS. Ранее в данной главе уже упоминалась эта технология, а в главе 34, “Диспетчер клиента: разработка приложения по технологии MIDAS”, она описана более подробно. Delphi допускает также создание распределенных приложений с использованием технологии *CORBA* — Common Object Request Broker Architecture (Архитектура типовых объектов брокера запросов). Спецификация CORBA была принята организацией Object Management Group. Эта технология позволяет создавать объектно-ориентированные распределенные приложения. Информация о том, как использовать технологию CORBA в приложениях Delphi 5, содержится в главе 27, “Разработка приложений CORBA в Delphi”.

Разработка серверной части

При разработке приложения, предназначенного для среды клиент/сервер, прежде чем приступать непосредственно к кодированию, обязательно выделите некоторое временя на планирование. Планирование включает определение бизнес-правил для приложения. Здесь необходимо решить, какие задачи будут выполняться сервером, а какие — клиентом. Затем следует установить структуры таблиц, связи между полями, типы используемых данных и требуемые механизмы защиты. Чтобы ответить на все эти вопросы, необходимо познакомиться с объектами баз данных, расположенными на стороне сервера.

Проиллюстрируем эти понятия на примере СУБД InterBase. Это — сервер баз данных, который входит в комплект поставки Delphi. С его помощью можно создавать автономные приложения клиент/сервер, соответствующие стандарту ANSI SQL-92. Для успешной работы с сервером InterBase необходимо освоить программу Windows ISQL, которая также поставляется вместе с Delphi.

На заметку

Рассмотрение особенностей реализации языка SQL в СУБД InterBase или любых других аспектов этого приложения выходит за рамки вопросов, освещаемых в этой книге. При обсуждении методов разработки приложений архитектуры клиент/сервер мы упоминаем о сервере InterBase и его реализации языка SQL только потому, что он входит в поставку Delphi. Практически все, о чем идет речь далее в этой главе, применимо и к другим реализациям SQL на других серверах баз данных, за исключением некоторых специфических свойств, зависящих от конкретных типов серверов.

Объекты базы данных

Сервер InterBase использует язык описания данных (Data Definition Language — DDL) для определения различных объектов базы данных, сохраняющих информацию о структуре самой базы данных и содержащихся в ней данных. Эти объекты иначе называются *метаданными*. Ниже в этой главе будут описаны различные объекты, входящие в состав метаданных, и приведены примеры их определения. Имейте в виду, что большинство баз данных SQL состоит из подобных объектов, с помощью которых обеспечивается сохранение информации о данных.

На заметку Такие мощные инструменты моделирования данных, как пакеты Erwin, xCase и RoboCase, предоставляют графическую среду, предназначенную для проектирования баз данных с помощью стандартных методологий моделирования данных. Стоит подумать о применении этих средств, прежде чем приступать к ручному проектированию системы, включающей одну–две сотни таблиц.

Определение таблиц

По структуре и функциональным особенностям использования таблицы InterBase мало чем отличаются от таблиц, рассмотренных в главе 28, “Создание локальных приложений баз данных”. Они также представляют собой неупорядоченный набор строк, каждая из которых включает некоторое число *столбцов*.

Типы данных

Столбцы могут иметь любой из допустимых типов данных, описание которых приведено в табл. 29.1.

Таблица 29.1. Типы данных СУБД InterBase

Имя	Размер	Диапазон/точность
`BLOB`	Переменный	Без ограничений, размер сегмента — 64 Кбайт
`CHAR(n)`	n символов	От 1 до 32 767 байт
`DATE`	64 бита	1 января 100 года — 11 декабря 5941 года
`DECIMAL` (длина, дробная часть)	Переменный	Длина — от 1 до 15;
		дробная часть — от 1 до 15
`DOUBLE PRECISION`	64 бита	От $1,7x10^{-308}$ до $1,7x10^{308}$
	(зависит от платформы)	
`FLOAT`	32 бита	От $3,4x10^{-38}$ до $3,4x10^{38}$
`INTEGER`	32 бита	От -2147483648 до 2147483648
`NUMERIC` (длина, дроб. часть)	Переменный	От -32 768 до 32 767
`SMALLINT`	16 бит	От 1 до 32 767
`VARCHAR(n)`	n символов	От 1 до 32 765

Типы поля могут быть также определены с помощью доменов (эта возможность рассматривается ниже, в разделе “Использование доменов”).

Создание таблицы

Для создания таблицы, ее столбцов и любых ограничений целостности данных, наложенных на каждый столбец, используется оператор `CREATE TABLE`. В листинге 29.1 приведен пример создания таблицы InterBase.

Листинг 29.1. Создание таблицы в InterBase

```
/* Определения домена */
CREATE DOMAIN FIRSTNAME AS VARCHAR(15);
CREATE DOMAIN LASTNAME AS VARCHAR(20);
CREATE DOMAIN DEPTNO AS CHAR(3)
        CHECK (VALUE = '000' OR (VALUE > '0' AND VALUE <= '999')
            OR VALUE IS NULL);
CREATE DOMAIN JOBCODE AS VARCHAR(5)
        CHECK (VALUE > '99999');
CREATE DOMAIN JOBGRADE AS SMALLINT
        CHECK (VALUE BETWEEN 0 AND 6);
CREATE DOMAIN SALARY AS NUMERIC(15, 2)
        DEFAULT 0 CHECK (VALUE > 0);

/* Таблица: EMPLOYEE, Владелец: SYSDBA */
CREATE TABLE EMPLOYEE (
        EMP_NO EMPNO NOT NULL,
        FIRST_NAME FIRSTNAME NOT NULL,
        LAST_NAME LASTNAME NOT NULL,
        PHONE_EXT VARCHAR(4),
        HIRE_DATE DATE DEFAULT 'NOW' NOT NULL,
        DEPT_NO DEPTNO NOT NULL,
        JOB_CODE JOBCODE NOT NULL,
        JOB_GRADE JOBGRADE NOT NULL,
        JOB_COUNTRY COUNTRYNAME NOT NULL,
        SALARY SALARY NOT NULL,
        FULL_NAME COMPUTED BY (last_name || ', ' || first_name),
PRIMARY KEY (EMP_NO));
```

В первом разделе листинга 29.1 приведено несколько операторов CREATE DOMAIN (их назначение вкратце разъясняется чуть позже). Во втором разделе листинга 29.1 создается таблица EMPLOYEE и определяются ее поля. Каждое определение поля состоит из типа поля и, возможно, фразы NOT NULL. Фраза NOT NULL указывает, что для этого поля значение должно задаваться обязательно. Кроме того, с помощью предложения PRIMARY KEY определен первичный ключ таблицы по полю EMP_NO. Определение первичного ключа не только гарантирует уникальность значений поля, но и создает для этого поля *индекс*, ускоряющий поиск данных.

Индексы

Индексы могут быть созданы явно — с помощью оператора CREATE INDEX. Они могут быть построены на базе одного или нескольких столбцов таблицы. Например, следующий SQL-оператор создает индекс по фамилии и имени служащего:

```
CREATE INDEX IDX_EMPNAME ON EMPLOYEE (LAST_NAME, FIRST_NAME);
```

Вычисляемые столбцы

Поле FULL_NAME — вычисляемое. Вычисляемые поля стоятся на некотором выражении, заданном в операторе COMPUTED BY. В листинге 29.1 оператор COMPUTED BY используется для объединения фамилии и имени, разделенных запятой. Можно создавать множество различных вычисляемых столбцов, отвечающих существующим потребностям. Возможности построения вычисляемых столбцов описываются в документации к используемому серверу.

Внешние ключи

Для некоторых полей можно устанавливать ограничение, называемое *внешним ключом*. Например, поле DEPT_NO определено следующим образом:

```
DEPT_NO DEPTNO NOT NULL
```

Тип DEPT_NO определен собственным доменом. Если вы не поняли приведенного выше определения, — не страшно, пока просто примите к сведению, что поле было определено правильно, подобно CHAR(3). Для гарантии того, что это поле ссылается на другое поле в другой таблице, к определению таблицы добавляется оператор FOREIGN KEY, как показано в следующем фрагменте кода:

```
CREATE TABLE EMPLOYEE (
        EMP_NO EMPNO NOT NULL,
        DEPT_NO DEPTNO NOT NULL
        FIRST_NAME FIRSTNAME NOT NULL,
        LAST_NAME LASTNAME NOT NULL,
PRIMARY KEY (EMP_NO),
FOREIGN KEY (DEPT_NO) REFERENCES DEPARTMENT (DEPT_NO));
```

В данном случае оператор FOREIGN KEY гарантирует, что значение в поле DEPT_NO таблицы EMPLOYEE будет одним из тех, которые присутствуют в столбце DEPT_NO таблицы DEPARTMENT. Кроме того, внешние ключи вызывают создание соответствующего индекса.

Значения по умолчанию

Чтобы определить значение по умолчанию для некоторого поля, можно использовать оператор DEFAULT. Так, обратите внимание на определение HIRE_DATE, в котором оператор DEFAULT используется для того, чтобы установить значение по умолчанию для этого поля:

```
HIRE_DATE DATE DEFAULT 'NOW' NOT NULL,
```

Здесь значением по умолчанию, которое присваивается этому полю, является результат вычисления функции InterBase NOW, возвращающей текущую дату.

Использование доменов

Обратите внимание на список определений доменов, который в приведенном примере расположен перед оператором CREATE TABLE. Домены — это пользовательские определения типа столбца. С их помощью можно определять столбцы таблицы со сложными параметрами, которые могут использоваться другими таблицами в той же базе данных. Например, в листинге 29.1 приведено следующее определение домена FIRSTNAME:

```
CREATE DOMAIN FIRSTNAME VARCHAR (15);
```

Любая другая таблица, которая использует FIRSTNAME как одно из определений поля, наследует тот же тип данных — VARCHAR(15). Если в будущем вы переопределите FIRSTNAME, любая таблица с полем этого типа унаследует новое определение.

К определению домена можно добавить ограничения — точно так же, как и к определению столбца. Например, определение домена JOBCODE, гарантирующее, что любое его значение будет больше 99999, выглядит следующим образом:

```
CREATE DOMAIN JOBCODE AS VARCHAR(5)
       CHECK (VALUE > '99999');
```

Определение домена JOBGRADE гарантирует, что его значение лежит между 0 и 6:

```
CREATE DOMAIN JOBGRADE AS SMALLINT
       CHECK (VALUE BETWEEN 0 AND 6);
```

Приведенные выше примеры дают лишь некоторое представление о тех типах ограничений целостности данных, которые можно помещать в определения таблицы. Однако эти возможности существенно меняются в зависимости от типа используемого SQL-сервера. В ваших же интересах тщательно изучить различные методы, предоставляемые этим сервером.

Определение бизнес-правил с помощью представлений, хранимых процедур и триггеров

В этой главе уже шла речь о бизнес-правилах — логике базы данных, которая определяет способ доступа и обработки данных. Существует три категории объектов базы данных, позволяющие определять правила, — *представления* (views), *хранимые процедуры* (stored procedures) и *триггеры* (triggers). Они рассматриваются в последующих разделах.

Определение представлений

Представление — это важный объект базы данных, позволяющий создавать произвольный результирующий набор данных, состоящий из кластеров столбцов одной или нескольких таблиц в базе данных. Над этой "виртуальной таблицей", как и над реальной, могут выполняться различные действия. Это позволяет определять подмножество данных, необходимых конкретному пользователю (или группе пользователей), дополненное ограничениями доступа к остальной части данных.

Чтобы создать представление, необходимо использовать оператор CREATE VIEW. В СУБД InterBase существует три основных способа создания представления.

- Горизонтальное подмножество строк одиночной таблицы. Например, следующее представление отображает все поля таблицы EMPLOYEE за исключением столбца SALARY:

  ```
  CREATE VIEW EMPLOYEE_LIST AS
      SELECT EMP_NO, FIRST_NAME, LAST_NAME, PHONE_EXT, FULL_NAME
      FROM EMPLOYEE;
  ```

- Подмножество строк и столбцов одиночной таблицы. В следующем примере показано представление служащих, у которых зарплата больше 100 000 долларов:

  ```
  CREATE VIEW EXECUTIVE_LIST AS
      SELECT EMP_NO, FIRST_NAME, LAST_NAME, PHONE_EXT, FULL_NAME
      FROM EMPLOYEE WHERE SALARY >= 100,000;
  ```

- Подмножество строк и столбцов из нескольких таблиц. В следующем примере показано подмножество таблицы EMPLOYEE наряду с двумя столбцами из таблицы JOB. Возвращенные строки и столбцы принадлежат одной таблице:

```
CREATE VIEW ENTRY_LEVEL_EMPL AS
    SELECT JOB_CODE, JOB_TITLE, FIRST_NAME, LAST_NAME.
    FROM JOB, EMPLOYEE
    WHERE JOB.JOB_CODE = EMPLOYEE.JOB_CODE AND SALARY < 15000;
```

Над представлениями можно выполнять множество действий. Одни представления существуют в режиме "только для чтения", в то время как другие допускают внесение изменений. Свойства представлений зависят от конкретных критериев, специфических для используемого сервера.

Определение хранимых процедур

Хранимую процедуру можно представить как автономную подпрограмму, которая выполняется на сервере и вызывается из клиентских приложений. Хранимые процедуры создаются с помощью оператора CREATE PROCEDURE. Существует два основных типа хранимых процедур.

- *Процедуры выбора* возвращают результирующий набор строк, состоящих из столбцов, выбранных из одной или нескольких таблиц или представления.
- *Выполняемые процедуры* не возвращают результирующего набора, но выполняют некоторый тип действий над данными на сервере.

Синтаксис для определения каждого типа процедуры одинаков и состоит из заголовка и тела.

Заголовок хранимой процедуры включает имя процедуры, необязательный список входных параметров и необязательный список выходных параметров. *Тело* состоит из необязательного списка локальных переменных и блока операторов SQL, которые и выполняют требуемые действия (этот блок расположен внутри блока BEGIN..END). В блок может быть вложена другая хранимая процедура.

Хранимая процедура типа SELECT

В листинге 29.2 приведен код хранимой процедуры типа SELECT.

Листинг 29.2. Хранимая процедура типа SELECT

```
CREATE PROCEDURE CUSTOMER_SELECT(
iCOUNTRY        VARCHAR(15)
)
RETURNS(
CUST_NO          INTEGER,
CUSTOMER         VARCHAR(25),
STATE_PROVINCE   VARCHAR(15),
COUNTRY          VARCHAR(15),
POSTAL_CODE      VARCHAR(12)
)
AS
BEGIN
    FOR SELECT
        CUST_NO,
        CUSTOMER,
        STATE_PROVINCE,
```

```
        COUNTRY,
        POSTAL_CODE
    FROM customer WHERE COUNTRY = :iCOUNTRY
    INTO
        :CUST_NO,
        :CUSTOMER,
        :STATE_PROVINCE,
        :COUNTRY,
        :POSTAL_CODE
    DO
        SUSPEND;
END
^
```

Этой процедуре в качестве параметра передается строка iCOUNTRY, а возвращает она те строки таблицы CUSTOMER, в которых страна соответствует параметру iCOUNTRY. Выполняемый код использует операторы FOR SELECT..DO, которые возвращают несколько строк. Эти операторы функционируют точно так же, как и обычный оператор SELECT, за исключением того, что они возвращают по одной строке за каждый проход цикла и помещают значения указанных столбцов в переменные, определенные в операторе INTO. Следовательно, результат выполнения этого оператора будет аналогичен выполнению в программе Windows ISQL следующего оператора:

```
SELECT * FROM CUSTOMER_SELECT("USA");
```

Позже будет рассмотрено, как вызвать эту хранимую процедуру на выполнение из приложения Delphi 5.

Выполняемая хранимая процедура

В листинге 29.3 приведен текст выполняемой хранимой процедуры.

Листинг 29.3. Пример выполняемой хранимой процедуры

```
CREATE PROCEDURE ADD_COUNTRY(
iCOUNTRY        VARCHAR(15),
iCURRENCY       VARCHAR(10)
)
AS
BEGIN
    INSERT INTO COUNTRY(COUNTRY, CURRENCY)
    VALUES (:iCOUNTRY, :iCURRENCY);
    SUSPEND;
END
^
```

Эта процедура добавляет новую запись в таблицу COUNTRY, используя оператор INSERT с данными, переданными в процедуру в качестве параметров. Эта процедура не возвращает результирующий набор данных и может быть выполнена с помощью оператора EXECUTE PROCEDURE в программе Windows ISQL следующим образом:

```
EXECUTE PROCEDURE ADD_COUNTRY("Mexico", "Peso");
```

Поддержка целостности данных с помощью хранимых процедур

Выше в этой главе отмечалось, что хранимые процедуры являются одним из способов поддержки целостности данных на стороне сервера, а не клиента. В логике хранимой процедуры можно проверять правила целостности и генерировать ошибку, если клиент запрашивает выполнение запрещенной операции. Например, в листинге 29.4 показан текст хранимой процедуры, выполняющей операции по размещению заказа на поставку. Она содержит и все необходимые проверки допустимости этой операции. Если операция недопустима, после генерации соответствующей исключительной ситуации выполнение процедуры прекращается.

Листинг 29.4. Хранимая процедура обработки заказа на поставку

```
CREATE EXCEPTION ORDER_ALREADY_SHIPPED "Order status is 'shipped.'";
CREATE EXCEPTION CUSTOMER_ON_HOLD "This customer is on hold.";
CREATE EXCEPTION CUSTOMER_CHECK "Overdue balance -- can't ship.";

CREATE PROCEDURE SHIP_ORDER (PO_NUM CHAR(8))
AS

  DECLARE VARIABLE ord_stat CHAR(7);
  DECLARE VARIABLE hold_stat CHAR(1);
  DECLARE VARIABLE cust_no INTEGER;
  DECLARE VARIABLE any_po CHAR(8);
BEGIN
/*  Сначала получим состояние заказа, информацию о заказчике и его
    номер, который будет использоваться далее для проверки в процедуре.
    Эти значения сохраняются в локальных переменных, определенных выше. */

  SELECT s.order_status, c.on_hold, c.cust_no
  FROM sales s, customer c
  WHERE po_number = :po_num
  AND s.cust_no = c.cust_no
  INTO :ord_stat, :hold_stat, :cust_no;

 /* Проверка, был ли заказ на товары уже выполнен. Если это так, генерируется
    исключительная ситуация и выполнение процедуры прерывается. */

  IF (ord_stat = "shipped") THEN
  BEGIN
    EXCEPTION order_already_shipped;
    SUSPEND;
  END

 /* Проверим, не числится ли заказчик в должниках. Если это так, генерируется
    исключительная ситуация и выполнение процедуры прерывается. */

  ELSE IF (hold_stat = "*") THEN
  BEGIN
    EXCEPTION customer_on_hold;
    SUSPEND;
```

```
  END

 /* Если баланс поставленных более чем за 2 месяца заказов отрицательный,
    поместим заказчика в список должников и сгенерируем исключительную
    ситуацию, после чего прервем выполнение процедуры. */

  FOR SELECT po_number
    FROM sales
    WHERE cust_no = :cust_no
    AND order_status = "shipped"
    AND paid = "n"
    AND ship_date < 'NOW' - 60
    INTO :any_po
  DO
  BEGIN
    EXCEPTION customer_check;

    UPDATE customer
    SET on_hold = "*"
    WHERE cust_no = :cust_no;

    SUSPEND;
  END

 /* Если все проверки успешно выполнены, заказ на поставку принимается. */
  UPDATE sales
  SET order_status = "shipped", ship_date = 'NOW'
  WHERE po_number = :po_num;

  SUSPEND;
END
^
```

Просматривая листинг 29.4, обратите внимание на еще одну функцию языка описания данных сервера InterBase — поддержку *исключительных ситуации* (exceptions). Они весьма напоминают исключительные ситуации в Delphi 5 и представляют собой поименованные сообщения об ошибках, генерируемые в хранимой процедуре при обнаружении ошибочной ситуации. При возникновении исключительной ситуации в вызывающее приложение возвращается сообщение об ошибке и выполнение хранимой процедуры завершается. Однако исключительную ситуацию можно обработать и внутри хранимой процедуры, после чего продолжить ее выполнение.

Исключительные ситуации определяются с помощью оператора `CREATE EXCEPTION`, как показано в листинге 29.4. Для генерации исключительной ситуации внутри хранимой процедуры используется следующий синтаксис:

```
EXCEPTION ExceptionName;
```

В листинге 29.4 определены три исключительные ситуации, которые могут генерироваться хранимой процедурой в различных ситуациях. Условия их генерации описаны в комментариях. Не забывайте, что все требуемые проверки должны выполняться внутри данной хранимой процедуры. В результате, любое использующее эту хранимую процедуру клиентское приложение будет придерживаться одних и тех же ограничений целостности данных.

Определение триггеров

Триггеры представляют собой хранимые процедуры, отличающиеся тем, что они автоматически выполняются в ответ на некоторое событие и не вызываются непосредственно ни из клиентского приложения, ни из другой хранимой процедуры. События триггеров генерируются при выполнении в таблице операций *обновления*, *вставки* и *удаления*.

Подобно хранимым процедурам, триггеры могут использовать исключительные ситуации. Поэтому при выполнении в определенной таблице любой из упомянутых выше операций могут осуществляться любые требуемые проверки целостности данных. Использование триггеров обеспечивает ряд преимуществ.

- *Обеспечение целостности данных*. Только достоверные данные могут быть вставлены в таблицу.
- *Упрощение сопровождения*. Любые внесенные в триггер изменения отображаются на работе всех приложений, использующих данную таблицу.
- *Автоматическое отслеживание изменений в таблице*. Триггер может регистрировать в журналах любые события, происходящие в таблице.
- *Автоматическое уведомление об изменениях в таблице*. С помощью системы предупреждающих событий триггер может автоматически посылать уведомления об имевших место изменениях.

Как и хранимые процедуры, триггеры состоят из заголовка и тела. Заголовок триггера содержит его имя, имя таблицы, к которой он применяется, и оператор, указывающий, когда этот триггер вызывается. Тело триггера включает необязательный список локальных переменных и блок операторов SQL в операторных скобках `begin ... end`, работающий так же, как и в хранимой процедуре.

Триггеры определяются с помощью оператора `CREATE TRIGGER`. В листинге 29.5 показан триггер сервера InterBase, предназначенный для сохранения хронологических сведений об изменении зарплаты служащих.

Листинг 29.5. Пример определения триггера

```
CREATE TRIGGER SALARY_CHANGE_HISTORY FOR EMPLOYEE
AFTER UPDATE AS
BEGIN
    IF (old.SALARY <> new.SALARY) THEN
        INSERT INTO SALARY_HISTORY (
            EMP_NO,
            CHANGE_DATE,
            UPDATER_ID,
            OLD_SALARY,
            PERCENT_CHANGE)
        VALUES
            old.EMP_NO,
            "now",
            USER,
            old.SALARY,
            (new.SALARY - old.SALARY) * 100 / old.SALARY);
END
```

Рассмотрим этот пример более подробно. Заголовок содержит оператор

```
CREATE TRIGGER SALARY_CHANGE_HISTORY FOR EMPLOYEE
AFTER UPDATE AS
```

Прежде всего оператор CREATE TRIGGER создает триггер с именем SALARY_CHANGE_HISTORY. Затем оператор FOR EMPLOYEE указывает, к какой таблице этот триггер должен быть приложен (в данном случае — к таблице EMPLOYEE). Оператор AFTER UPDATE показывает, что триггер должен запускаться после обновления данных таблицы EMPLOYEE. Здесь может также использоваться оператор BEFORE UPDATE, который указывает, что триггер должен запускаться *до* внесения изменений в таблицу.

Триггеры применяются не только при модификации таблиц. Вот элементы заголовка, которые могут использоваться в определении триггеров:

- AFTER UPDATE — запуск триггера после модификации таблицы;
- AFTER INSERT — запуск триггера после вставки записи в таблицу;
- AFTER DELETE — запуск триггера после удаления записи из таблицы;
- BEFORE UPDATE — запуск триггера перед модификацией таблицы;
- BEFORE INSERT — запуск триггера перед вставкой новой записи в таблицу;
- BEFORE DELETE — запуск триггера перед удалением записи из таблицы.

В определении триггера следующий за оператором AS код представляет собой выполняемое тело триггера. В листинге 29.5 в нем сравнивается старая и новая заработная плата. Если между ними имеется различие, добавляется новая запись в таблицу SALARY_HISTORY, предназначенную для регистрации подобных изменений.

Обратите внимание: в примере используются идентификаторы old и new. Эти контекстные переменные указывают на текущее и предыдущее значения модифицируемой строки. Идентификатор old не используется при вставке записи, а new — при удалении записи.

Более подробно использование триггеров будет описано в главе 32, “Разработка приложений MIDAS”, в которой рассматривается процесс разработки конкретного приложения архитектуры клиент сервер в среде InterBase.

Привилегии и права доступа к объектам базы данных

Пользователям в базах данных клиент/сервер доступ к данным на сервере может как предоставляться, так и запрещаться. Пользователю могут предоставляться *права доступа* к таблицам, хранимым процедурам и представлениям. Они назначаются посредством указания в операторе GRANT предоставляемых привилегий. В табл. 29.2 приведены различные типы привилегий, существующие в InterBase и большинстве других типов SQL-серверов.

Таблица 29.2. Привилегии доступа в языке SQL

Привилегия	Тип доступа
ALL	Пользователь может выбирать, вставлять, модифицировать и удалять данные (см. другие привилегии доступа). Предоставляется также право выполнения хранимых процедур
SELECT	Пользователь может читать данные
DELETE	Пользователь может удалять данные

Окончание табл. 29.2

Привилегия	Тип доступа
INSERT	Пользователь может вставлять новые данные
UPDATE	Пользователь может редактировать данные
EXECUTE	Пользователь может выполнять или вызывать хранимую процедуру

Предоставление доступа к таблицам

Чтобы предоставить пользователю доступ к таблице, необходимо использовать оператор GRANT, который должен включать следующую информацию:

- привилегию доступа;
- имя таблицы, хранимой процедуры или представления, к которым применяется эта привилегия;
- имя пользователя, которому предоставляется доступ.

По умолчанию в InterBase только создатель таблицы имеет к ней доступ и может предоставлять права доступа к ней другим пользователям. Примеры предоставления прав доступа приведены ниже. Дополнительную информацию по этому вопросу можно найти в документации к серверу InterBase.

Следующий оператор предоставляет привилегию UPDATE к таблице EMPLOYEE пользователю с именем JOHN:

```
GRANT UPDATE ON EMPLOYEE TO JOHN;
```

Следующий оператор предоставляет доступ к таблице EMPLOYEE с привилегиями чтения и редактирования пользователям JOHN и JANE:

```
GRANT SELECT, UPDATE ON EMPLOYEE TO JOHN, JANE;
```

Как видите, существует возможность предоставить доступ списку пользователей. Если требуется предоставлять пользователю все права, укажите в операторе GRANT привилегию ALL:

```
GRANT ALL ON EMLPOYEE TO JANE;
```

В этом примере пользователю JANE предоставляется право доступа к таблице EMPLOYEE с привилегиями SELECT, UPDATE и DELETE.

Кроме того, можно предоставить права доступа к отдельным столбцам в таблице:

```
GRANT SELECT, UPDATE (CONTACT, PHONE) ON CUSTOMERS TO PUBLIC;
```

Этот оператор предоставляет доступ с правом чтения и редактирования полей CONTACT и PHONE в таблице CUSTOMERS. Ключевое слово PUBLIC указывает, что это право доступа предоставляется всем пользователям.

Права доступа необходимо также предоставлять и хранимым процедурам, которые требуют обращения к определенным таблицам. Например, в следующем примере доступ с правом чтения и модификации таблицы CUSTOMER предоставляется хранимой процедуре UPDATE_CUSTOMER:

```
GRANT SELECT, UPDATE ON CUSTOMER TO PROCEDURE UPDATE_CUSTOMER;
```

Все упоминавшиеся варианты оператора GRANT применимы также и к хранимым процедурам.

Предоставление доступа к представлениям

В большинстве случаев использование оператора GRANT для представления воспринимается системой аналогично применению этого оператора к таблице. Однако следует убедиться, что пользователь, которому предоставляются права UPDATE, INSERT или DELETE, имеет те же права в отношении базовых таблиц, на которые ссылается представление. Использование оператора WITH CHECK OPTION при создании представления гарантирует, что редактируемые поля могут быть отображены в представлении до выполнения операции. Рекомендуется создавать модифицируемые представления именно с использованием этого параметра.

Предоставление доступа к хранимым процедурам

Чтобы пользователь или хранимая процедура могли выполнить некоторую хранимую процедуру, им должно быть предоставлено право доступа к этой процедуре с привилегией EXECUTE. Ниже приведен пример предоставления перечисленным в списке пользователям и хранимым процедурам права доступа с привилегией EXECUTE к процедуре EDIT_CUSTOMER:

```
GRANT EXECUTE ON EDIT_CUSTOMER TO MIKE, KIM, SALLY, PROCEDURE ADD_CUSTOMER;
```

Пользователи MIKE, KIM и SALLY, а также хранимая процедура ADD_CUSTOMER, смогут выполнять хранимую процедуру EDIT_CUSTOMER.

Отмена прав доступа пользователей

Чтобы отменить доступ пользователя к таблице или хранимой процедуре, необходимо использовать оператор REVOKE, который должен включать следующие элементы:

- отменяемые привилегии доступа;
- имя таблицы или хранимой процедуры, права доступа к которой нужно отменить;
- имя пользователя, чьи права будут отменены.

Оператор REVOKE аналогичен оператору GRANT. В приведенном ниже примере показано, как отменить право доступа к таблице:

```
REVOKE UPDATE, DELETE ON EMPLOYEE TO JANE, TOM;
```

Разработка клиентской части

В следующих разделах рассматриваются компоненты Delphi 5 для работы с базами данных и способы их использования для получения доступа к базам данных в среде клиент/сервер. Речь пойдет также о различных методах использования этих компонентов для решения типичных задач обработки данных.

Использование компонента TDatabase

Компонент TDatabase предоставляет широкие возможности управления соединениями с базами данных:

- создание постоянного подключения к базе данных;
- переопределение установленной по умолчанию процедуры регистрации на сервере;

- создание BDE-псевдонима на уровне приложения;
- управление транзакциями и определение уровня их изоляции.

В табл. 29.3 и 29.4 приведено краткое описание свойств и методов компонента TDatabase. За более подробной информацией обращайтесь к справочной системе Delphi или к документации. Об использовании описанных свойств и методов речь идет в этой и следующих главах.

Таблица 29.3. Свойства компонента TDatabase

Свойство	Назначение
AliasName	Определяет существующий BDE-псевдоним, созданный с помощью утилиты BDE Configuration. Не может использоваться вместе со свойством DriverName
Connected	Свойство типа Boolean определяет, связан ли компонент TDatabase с базой данных
DatabaseName	Определяет специфический для приложения псевдоним. Другие компоненты, производные от класса TDataSet (TTable, TQuery, TStoredProc), используют значение этого свойства для установки значения собственного свойства AliasName
DatasetCount	Содержит число компонентов, производных от класса TDataSet, которые связаны с данным компонентом TDatabase
Datasets	Массив, содержащий ссылки на все компоненты, производные от класса TDataSet, связанные с данным компонентом Tdatabase
Directory	Рабочий каталог базы данных Paradox или dBase
DriverName	Содержит имя BDE-драйвера, такого как Oracle, dBASE, InterBase и т.д. Это свойство не может использоваться вместе со свойством AliasName
Exclusive	Предоставляет приложению монопольные права доступа к базе данных
Handle	Используется для прямых вызовов Borland Database Engine (BDE) API
InTransaction	Определяет, выполняется ли в настоящий момент транзакция
IsSQLBased	Свойство типа Boolean определяет, относится ли подключенная база данных к типу SQL. Принимает значение False, если свойство Driver имеет значение STANDARD
KeepConnection	Свойство типа Boolean, определяет, сохраняет ли компонент TDatabase подключение к базе данных, когда нет открытых компонентов, производных от класса TDataSet. Используется для повышения эффективности работы, так как установка соединения с некоторыми SQL-серверами может занимать длительное время
Locale	Идентифицирует драйвер языка, используемый с компонентом TDatabase. Используется преимущественно при прямых вызовах BDE API
LoginPrompt	Определяет, как компонент TDatabase обрабатывает вход пользователя в систему. Если значение этого свойство равно True, используется процедура регистрации, принимаемая в системе по умолчанию. Если значение этого свойство равно False, параметры регистрации должны находиться в коде обработчика события TDatabase.OnLogin
Name	Содержит имя компонента, используемое для ссылок со стороны других компонентов

Окончание табл. 29.3

Свойство	Назначение
`Owner`	Имя объекта, владельца компонента `TDatabase`
`Params`	Содержит параметры, необходимые для подключения к серверу базы данных. Используемые по умолчанию параметры устанавливаются с помощью утилиты настройки BDE, но при необходимости они могут быть переопределены
`Session`	Указывает на компонент сеанса, с которым связан данный компонент базы данных
`SessionAlias`	Определяет, использует ли данный компонент базы данных псевдоним сеанса
`Tag`	Свойство типа `longint`, предназначенное для хранения произвольного целого значения
`Temporary`	Свойство типа `Boolean`. Определяет, был ли данный компонент `TDatabase` создан в результате открытия компонента `TTable`, `TQuery` или `TStoredProc`, когда этого компонента еще не существовало
`TraceFlags`	Определяет операции базы данных, которые будут отслеживаться средством SQL Monitor во время выполнения приложения
`TransIsolation`	Определяет уровень изоляции транзакций для сервера

В табл. 29.4 представлено описание методов компонента `TDatabase`.

Таблица 29.4. Методы компонента TDatabase

Метод	Назначение
`ApplyUpdates()`	Отправляет ожидающие (pending) кэшированные обновления для определенных наборов данных на сервер базы данных
`Close()`	Закрывает соединение `TDatabase` и все связанные с ним компоненты типа `TDataSet`
`CloseDatasets()`	Закрывает все связанные с `TDatabase` компоненты типа `TDataSet`. При этом соединение `TDatabase` не обязательно должно быть закрыто
`Commit()`	Фиксирует все изменения в базе данных, выполненные в транзакции. Транзакция должна быть установлена посредством вызова метода `StartTransaction`
`Create()`	Распределяет память и создает экземпляр компонента `TDatabase`
`Destroy()`	Освобождает память и удаляет экземпляр компонента `TDatabase`
`Execute()`	Выполняет оператор SQL без использования компонента `TQuery`
`FlushSchemaCache()`	Очистка данных кэшированной структуры для таблицы
`Free()`	Аналогичен методу `Destroy` во всем, за исключением того, что сначала определяет, не равен ли данный компонент `Tdatabase` значению `nil`
`Open()`	Подключает компонент `TDatabase` к серверу базы данных. При установке свойства `Connected` в `True` этот метод вызывается автоматически

Окончание табл. 29.4

Метод	Назначение
`RollBack()`	Осуществляет откат или отмену выполнения транзакции. В результате отменяются любые изменения, внесенные с момента последнего вызова метода `StartTransaction`
`StartTransaction()`	Начинается выполнение транзакции с уровнем изоляции, заданным значением свойства `TransIsolation`. Изменения в базу данных вносятся, но не фиксируются до тех пор, пока не будет выполнен метод `Commit`. Для отмены изменений следует вызвать метод `RollBack`
`ValidateName()`	Генерирует исключительную ситуацию, если указанная база данных в активном сеансе уже открыта

Подключение на уровне приложения

Одна из причин использования компонента `TDatabase` в проекте связана с возможностью предоставить псевдоним уровня приложения всему проекту. В отличие от псевдонима уровня BDE, псевдоним, предоставляемый компонентом `TDatabase`, будет доступен только в данном проекте. Псевдоним уровня приложения может быть использован другими проектами лишь при условии размещения компонента `TDatabase` в разделяемом компоненте `TDataModule`. Совместное использование компонента `TDataModule` возможно в том случае, если он находится там, откуда другие разработчики смогут скопировать его в свои проекты, либо когда он помещен в хранилище объектов (Object Repository).

Псевдоним уровня приложения определяется посредством присвоения требуемого значения свойству `TDatabase.DatabaseName`. Псевдоним BDE, определяющий базу данных, к которой подключается компонент `TDatabase`, указывается в свойстве `TDatabase.AliasName`.

Управление защитой

Компонент `TDatabase` позволяет управлять доступом пользователей к данным сервера посредством задания способа проведения процедуры регистрации. Чтобы получить доступ к данным, пользователь при регистрации должен ввести корректное имя пользователя и пароль. Существует стандартное диалоговое окно регистрации в системе, которое по умолчанию используется при подключении к серверам базы данных.

Имеется несколько способов выполнения процедуры регистрации в системе. Во-первых, можно вообще отменить процедуру регистрации и предоставить всем пользователям свободный доступ к данным. Во-вторых, можно использовать различные типы диалога регистрации, предусматривающие при необходимости выполнение собственных проверок до того, как имя пользователя и его пароль будут направлены серверу для проведения стандартной регистрации. В-третьих, можно разрешить пользователям заканчивать сеанс и заново регистрироваться на сервере без перезагрузки приложения. В следующих разделах речь пойдет о реализации всех этих вариантов регистрации.

Автоматическая регистрация

Чтобы предотвратить вывод диалогового окна регистрации на сервере при запуске приложения, необходимо указанным образом установить перечисленные ниже свойства компонента `TDatabase`.

После установки свойств компонента `TDatabase` необходимо связать компоненты `TTable`, `TQuery` и `TStoredProc` с данным компонентом `TDatabase`. Для этого установите значение

свойства TDatabase.DatabaseName равным значению свойства Alias этих компонентов (это значение появляется в раскрывающемся списке псевдонимов при выборе данного свойства в окне Object Inspector).

Свойство	Назначение
AliasName	Установите равным существующему псевдониму BDE, который был определен утилитой настройки BDE. Это то же значение, которое обычно используется как значение свойства Alias компонентов TTable и TQuery
DatabaseName	Установите равным псевдониму уровня приложения, который будет доступен потомкам компонента TDataSet (компоненты TTable, TQuery и TStoredProc) в текущем приложении. Эти компоненты будут использовать его в качестве значения свойства Alias
LoginPrompt	Установите равным False. При этом компонент TDatabase будет просматривать свое свойство Params, чтобы найти имя пользователя и пароль
Params	Определите имя пользователя и пароль. Чтобы установить эти значения, откройте для этого свойства окно String List Editor

Теперь, после установки значения свойства TDatabase.Connected равным True, приложение подключится к серверу, не запрашивая значения имени учетной записи и пароля, а используя предопределенные значения в своем свойстве Params. То же самое будет происходить и при запуске приложения во время выполнения.

На прилагаемом компакт-диске содержится небольшой проект NoLogin.dpr, иллюстрирующий все вышесказанное.

Собственное диалоговое окно регистрации

В некоторых случаях может потребоваться вывести пользователю нестандартное окно диалога регистрации. Возможно, для работы вашего приложения необходима дополнительная информация, помимо имени пользователя и пароля. А может, вам просто желательно в начале работы приложения вывести на экран нечто более привлекательное, чем стандартное окно регистрации. В любом случае это очень просто сделать.

Прежде всего отключите вывод стандартного диалогового окна регистрации, установив значение свойства TDatabase.LoginPrompt равным True. Однако на этот раз не следует указывать имя пользователя и пароль в свойстве Params. Вместо этого создайте обработчик события TDatabase.OnLogin. Этот обработчик вызывается всякий раз, когда принимает значение True свойство TDatabase.Connected или TDatabase.LoginPrompt.

В приведенной ниже функции выводится пользовательская форма регистрации, а затем введенное имя пользователя и пароль возвращаются вызывающему приложению:

```
function GetLoginParams(ALoginParams: TStrings): word;
var
  LoginForm: TLoginForm;
begin
  LoginForm := TLoginForm.Create(Application);
  try
    Result := LoginForm.ShowModal;
    if Result = mrOK then
    begin
      ALoginParams.Values['USER NAME'] := LoginForm.edtUserName.Text;
      ALoginParams.Values['PASSWORD'] := LoginForm.edtPassWord.Text;
```

```
    end;
  finally
    LoginForm.Free;
  end;
end;
```

А вот как эта функция вызывается в обработчике TDatabase.OnLogin (данный пример содержится также в проекте LOGIN.DPR на прилагаемом компакт-диске):

```
procedure TMainForm.dbMainLogin(Database: TDatabase; LoginParams: TStrings);
begin
  GetLoginParams(LoginParams);
end;
```

Выход из текущего сеанса

Можно также предоставить пользователю возможность выхода и повторной регистрации (вероятно, уже с другим именем пользователя) без закрытия приложения. Для этого, пользуясь приведенным выше способом, запретите вывод системного диалогового окна регистрации и вновь создайте обработчик события OnLogin. Чтобы этот обработчик был вызван, следует установить значение свойства TDatabase.LoginPrompt равным True. В данном обработчике вам потребуется несколько переменных — для сохранения имени пользователя и его пароля, а также булева переменная, предназначенная для указания, успешно ли завершена регистрация пользователя в системе. Кроме того, потребуется написать не один, а два метода — для выполнения входа в систему и выхода из нее. В листинге 29.6 приведен соответствующий код.

Листинг 29.6. Пример поддержки входа и выхода из системы

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Grids, DBGrids, DB, DBTables;

type
  TMainForm = class(TForm)
    dbMain: TDatabase;
    tblEmployee: TTable;
    dsEmployee: TDataSource;
    dgbEmployee: TDBGrid;
    btnLogon: TButton;
    btnLogOff: TButton;
    procedure btnLogonClick(Sender: TObject);
    procedure dbMainLogin(Database: TDatabase; LoginParams: TStrings);
    procedure btnLogOffClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  public
    TempLoginParams: TStringList;
```

```
    LoginSuccess: Boolean;
  end;

var
  MainForm: TMainForm;

implementation
uses LoginFrm;

{$R *.DFM}

procedure TMainForm.btnLogonClick(Sender: TObject);
begin
  // Получение новых параметров регистрации
  if GetLoginParams(TempLoginParams) = mrOk then
  begin
    // Отключение компонента TDatabase
    dbMain.Connected := False;
    try
      { Попытка повторного подключения к компоненту TDatabase. При этом
        будет вызван обработчик события DataBase1Login, который внесет
        в LoginParams текущие имя пользователя и пароль. }
      dbMain.Connected := True;
      tblEmployee.Active := True;
      LoginSuccess := True;
    except
      on EDBEngineError do
      begin
        { Если регистрация в системе не выполнена, генерируется
          исключительная ситуация. }
        LoginSuccess := False;
        Raise;
      end;
    end;
  end;
end;

procedure TMainForm.dbMainLogin(Database: TDatabase; LoginParams: TStrings);
begin
  LoginParams.Assign(TempLoginParams);
end;

procedure TMainForm.btnLogOffClick(Sender: TObject);
begin
  { Отключение компонента TDatabase и сброс значений переменных. }
  dbMain.Connected := False;
  TempLoginParams.Clear;
end;

procedure TMainForm.FormCreate(Sender: TObject);
```

```
begin
  TempLoginParams := TStringList.Create;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  TempLoginParams.Free;
end;

end.
```

Вы, очевидно, заметили, что в главной форме есть два поля: TempLoginParams и LoginSuccess. Поле TempLoginParams содержит имя пользователя и пароль. Метод btnLogonClick() — это процесс входа в систему, а обработчик события btnLogOffClick() — это процесс выхода из системы. Метод dbMainLogin() представляет собой обработчик события OnLogin для объекта dbMain. Работа кода поясняется в комментариях листинга. Обратите внимание на то, что в этом проекте используется тот же компонент TLoginForm, что и в предыдущем примере. Приведенный пример находится в проекте LogOnOff.dpr на прилагаемом компакт-диске.

Управление транзакциями

Мы уже познакомились с транзакциями в этой главе. Речь шла о том, что транзакции позволяют вносить несколько изменений в базу данных единовременно, как единое целое, что гарантирует ее целостность.

Обработка транзакций может контролироваться клиентским приложением Delphi 5, для чего используются специальные свойства и методы компонента TDatabase. В следующих разделах будет показано, как организовать обработку транзакций в приложении Delphi 5.

Неявное или явное управление транзакциями

В Delphi 5 транзакции могут обрабатываться явно или неявно. По умолчанию транзакции обрабатываются неявно.

Неявные транзакции — это такие транзакции, которые запускаются и завершаются для каждой из последовательно обрабатываемых строк. Это происходит всякий раз, когда метод Post вызывается вашей подпрограммой или подпрограммой компонента библиотеки VCL. Поскольку транзакции выполняются построчно, нагрузка в сети возрастает, что может привести к снижению производительности приложения.

Явные транзакции обрабатываются одним из двух способов. Первый способ заключается в использовании метода StartTransaction(), Commit() или RollBack() компонента TDatabase. Второй способ состоит в передаче на выполнение SQL-операторов в компоненте TQuery, о чем речь пойдет чуть ниже. Рекомендуем применять явное управление транзакцией, поскольку при этом уменьшается сетевая нагрузка, а используемый код более безопасен.

Управление транзакциями

В табл. 29.4 были приведены три метода компонента TDatabase, которые используются при работе с транзакциями: StartTransaction(), Commit() и RollBack().

Метод StartTransaction() начинает выполнение транзакции, используя уровень изоляции, определенный свойством TDatabase.TransIsolation. Любые изменения, выполненные на сервере после вызова метода StartTransaction(), будут попадать в текущую транзакцию.

Если все изменения на сервере выполнены успешно, необходимо вызвать метод TDatabase.Commit(), который вызовет фиксацию выполненных изменений в базе данных. В случае ошибки необходимо вызвать метод TDatabase.RollBack(), который отменит любые внесенные изменения.

Продемонстрируем использование транзакций на примере все той же базы данных складского учета. Допустим, что у нас имеются таблицы ORDER и INVENTORY. При приеме каждого нового заказа в таблицу ORDER следует добавить новую запись. Одновременно необходимо модифицировать таблицу INVENTORY, чтобы отобразить изменения остатка на складе заказанного товара или товаров. Предположим, пользователь вводит заказ в систему, не использующую транзакции. В таблице ORDER появляется новая запись, и вдруг перед модификацией таблицы INVENTORY происходит сбой питания. База данных оказывается в несогласованном состоянии, поскольку таблица INVENTORY неверно отражает остаток товара на складе. Использование транзакций решает эту проблему, гарантируя, что все изменения в таблицах будут зафиксированы в базе данных, только если все операции транзакции были выполнены успешно. В листинге 29.7 показано, как все эти операции могут быть реализованы в Delphi 5.

Листинг 29.7. Обработка транзакции

```
dbMain.StartTransaction;
  try
    spAddOrder.ParamByName('ORDER_NO').AsInteger := OrderNo;
    { Устанавливаем значение параметров и выполняем хранимую
      процедуру добавления нового заказа к таблице ORDER.}
    spAddOrder.ExecProc;
    { Итерация по всем заказанным товарам и обновление таблицы
      INVENTORY для отражения их остатка на складе. }
    for i := 0 to PartList.Count - 1 do
    begin
      spReduceParts.ParamByName('PART_NO').AsInteger :=
        PartRec(PartList.Objects[i]).PartNo;
      spReduceParts.ParamByName('NUM_SOLD').AsInteger :=
        PartRec(PartList.Objects[i]).NumSold;
      spReduceParts.ExecProc;
    end;
    // Внесение изменений в таблицы ORDER и INVENTORY.
    dbMain.Commit;
  except
    // При появлении ошибки отменить все изменения.
    dbMain.RollBack;
  raise;
  end;
```

Приведенный код — упрощенный пример того, как использовать транзакции для обеспечения согласованности базы данных. Здесь применялись две хранимые процедуры: одна для добавления новой записи заказа, и другая для модификации таблицы INVENTORY в соответствии с новыми данными. Однако помните, что здесь приведен только фрагмент кода, демонстрирующий использование транзакций в Delphi. Выполнять такую задачу лучше всего на сервере.

В некоторых случаях требуемый способ обработки транзакции может зависеть от специфических возможностей сервера. В этой ситуации необходимо использовать компонент TQuery для передачи на сервер специфического SQL-кода. Эти действия требуют установки соответствующего режима пересылки команд SQL.

Режим пересылки команд SQL

Режим пересылки команд SQL определяет, как приложения Delphi 5 и Borland Database Engine (BDE) будут совместно использовать соединения с серверами баз данных. Соединения BDE используются в тех методах Delphi, которые вызывают API BDE. Режим пересылки устанавливается с помощью утилиты настройки BDE и может принимать три различных значения.

Значение	Описание
SHARED AUTOCOMMIT	Транзакции обрабатываются построчно. Этот метод наиболее близок локальным базам данных, но в приложениях клиент/сервер такой тип обработки вызывает огромную нагрузку на сеть, поэтому его применять не рекомендуется. Однако это параметр используется для приложений Delphi 5 по умолчанию
SHARED NOAUTOCOMMIT	Приложения Delphi 5 должны явно начать, зафиксировать или (при необходимости) отменить транзакции, используя методы `StartTransaction()`, `Commit()` и `RollBack()`
NOT SHARED	Ядро BDE и компоненты `TQuery` выполняют пересылаемые операторы SQL, без совместного использования одних и тех же соединений. Это означает, что SQL-код не ограничен возможностями BDE и может использовать специфические возможности сервера

Если режим пересылаемых SQL-команд не используется, но требуется контролировать выполнение транзакций, установите режим `SHARED NOAUTOCOMMIT` и организуйте обработку транзакций самостоятельно. Вероятно, этот метод будет отвечать большинству ваших потребностей, хотя и может порождать конфликты в многопользовательских системах с высокой частотой обновлений.

Уровни изоляции

Уровни изоляции определяют, как транзакции "видят" данные, к которым обращаются из других транзакций. Значение свойства `TDatabase.TransIsolation` определяет, какой уровень изоляции использует та или иная транзакция. Существует три уровня изоляции, которые можно назначить свойству `TransIsolation`.

Уровень изоляции	Описание
tiDirtyRead	Самый низкий уровень. Транзакциям, использующим этот уровень изоляции, доступны незафиксированные изменения, вносимые другими транзакциями
tiReadCommitted	Уровень, установленный по умолчанию. Транзакции могут считывать изменения только после их фиксации другими транзакциями
tiRepeatableRead	Самый высокий уровень. Транзакции не могут считывать изменения в предварительно прочитанных ими данных, внесенные и зафиксированные другими транзакциями

Поддержка перечисленных выше уровней изоляции может изменяться на различных серверах. Если определенный уровень не поддерживается сервером, Delphi 5 будет использовать следующий, но обязательно более высокий уровень изоляции.

Какой компонент использовать — TTable или TQuery?

Бытует ошибочное мнение, что разработка клиентских приложений среды клиент/сервер не отличается или очень похожа на разработку приложений, работающих с локальными базами данных.

Этот идейный штамп проявляется себя в том, как и когда работник использует для доступа к базам данных компоненты TTable и TQuery. Ниже мы обсудим достоинства и недостатки использования компонента TTable, когда его нужно использовать и в каких случаях он неприменим. Кроме того, вы поймете, почему для работы с распределенной базой данных, как правило, лучше использовать компонент TQuery.

Компонент TTable и SQL-команды

Компоненты TTable большей частью используются для доступа к данным в локальной среде. Они разработаны специально для выполнения действий, которые требуются при работе с локальными базами данных (например, для манипулирования всей таблицей, перемещения по таблице вперед и назад, поиска в таблице требуемой записи). Однако эти операции не годятся для работы с базами данных SQL-серверов. Реляционные серверные базы данных разработаны таким образом, что доступ к ним осуществляется через наборы данных. Такие понятия, как следующая, предыдущая или последняя запись, нельзя применить к SQL-наборам данных (а это базовые понятия при работе с компонентом TTable). Хотя некоторые базы данных SQL поддерживают так называемые “прокручиваемые” (или скроллируемые) курсоры, эта операция обычно применяются только к полученному результирующему набору данных. Кроме того, некоторые серверы не поддерживают режим прокрутки в обоих направлениях.

При анализе возможности использования компонентов TTable для доступа к базам данных SQL особое внимание следует обратить на то, что команды, введенные с помощью компонента TTable, будут преобразованы в SQL-код, который должен быть правильно понят базой данных. А такое преобразование не только ограничивает возможности доступа к данным на сервере, но серьезно сказывается на общей производительности приложения.

Чтобы убедиться в неэффективности использования компонента TTable для доступа к большим наборам данных, рассмотрим процесс открытия компонента TTable с целью получения нескольких записей. Время, затрачиваемое на открытие SQL-таблицы с помощью компонента TTable, прямо пропорционально числу полей и объему метаданных (индексы и т.д.), связанных с этой SQL-таблицей. При выполнении команды Table1.Open для SQL-таблицы BDE генерирует несколько SQL-команд к серверу для получения сведений о столбцах таблицы, индексах и т.п. Затем выполняется оператор SELECT для формирования результирующего набора, состоящего из всех столбцов и строк таблицы. Поэтому время, которое затрачивается на открытие таблицы, прямо пропорционально размеру SQL-таблицы (количеству ее строк). И хотя в отображаемый на экране компонент требуется поместить лишь небольшое количество строк данных, в ответ на запрос в результирующий набор данных будут переданы все строки таблицы. Этот процесс происходит всякий раз, когда открывается компонент TTable. В результате в очень больших таблицах, типичных для баз данных клиент/сервер, только одна эта простая операция может выполняться около 20 секунд. Имейте также в виду, что некоторые SQL-серверы (например, Sybase и Microsoft SQL) не позволяют клиенту прерывать процесс возврата результирующего набора, и в этом случае размер открываемой таблицы *всегда* будет определять время ее открытия. Следует отметить, что СУБД Oracle, InterBase и Informix позволяют остановить процесс передачи результирующего набора данных.

Несмотря на общие недостатки использования компонента TTable в клиентском приложении, он прекрасно подходит для доступа к таблицам малого размера, пусть даже расположенным на сервере. Но и в этом случае следует обязательно протестировать готовое приложение, чтобы выяснить, соответствует ли его производительность поставленным целям.

На заметку

IDAS обрабатывает возвращаемые пакеты данных несколько иначе. Более подробно об этом можно прочесть в главе 34, "Диспетчер клиента: разработка приложения по технологии MIDAS".

Использование методов FindKey и FindNearest в таблицах баз данных SQL

Хотя компонент TTable позволяет выполнять поиск записей с помощью метода FindKey(), существуют определенные ограничения на его использование в базах данных SQL. Во-первых, компонент TTable может использовать этот метод для в индексированного поля (или полей, если выполняется поиск, основанный на значениях нескольких полей). Компонент TQuery не имеет таких ограничений, поскольку в этом случае поиск записи выполняется средствами SQL. Вызов метода TTable.FindKey() вызывает генерацию команды SELECT для выборки данных из таблицы на сервере. Однако результирующий набор будет состоять из всех полей таблицы, причем, даже в том случае, если в компонент TTable в окне редактора полей была помещена только часть полей таблицы.

Получить доступ к функциональности метода FindNearest() компонента TTable в SQL-коде не просто, однако возможно. Следующий SQL-оператор почти аналогичен по функциональным возможностям методу TTable.FindNearest():

```
SELECT * FROM EMPLOYEES
    WHERE NAME >= "CL"
    ORDER BY NOMENCLATURE
```

В данном примере результирующий набор возвращает запись либо на искомую позицию, либо на следующую за той позицией, на которой она могла бы быть. Проблема состоит в том, что этот результирующий набор возвращает *все* записи после искомой позиции. Ограничим поиск так, чтобы результирующий набор состоял только из одной записи:

```
SELECT * FROM EMPLOYEES
    WHERE NAME = (SELECT MIN(NAME) FROM EMPLOYEES
    WHERE NAME >= "CL")
```

В данном примере используется вложенный оператор SELECT. В нем внутренний оператор возвращает результирующий набор внешнему оператору SELECT. Внешний оператор затем использует этот результирующий набор для собственной обработки. Во внутреннем запросе в данном примере используется вычисляемая SQL-функция MIN() для возврата наименьшего значения столбца NAME таблицы EMPLOYEES. Этот результирующий набор из одной строки и одного столбца затем используется во внешнем запросе для получения остальных строк.

Благодаря применению компонента TQuery обеспечивается гибкость и эффективность вашего приложения, чего невозможно достичь с помощью компонента TTable.

Использование компонента TQuery

В предыдущей главе вы познакомились с компонентом TQuery и узнали, как его использовать для поиска результирующих наборов строк в таблицах. В этом разделе компонент TQuery рассматривается более подробно. Вы узнаете, как создавать динамические SQL-операторы в процессе работы, как передать параметры в запросы и как повысить эффективность компонента TQuery, изменяя определенные значения его свойств.

Существует два основных типа запросов, в которых используется компонент TQuery: возвращающие и не возвращающие результирующий набор. Для запросов, возвращающих результирующий набор, используйте метод TQuery.Open(). Метод TQuery.ExecSQL() используется в том случае, когда результирующий набор не возвращается.

Динамический SQL

Термин *динамический SQL* означает, что во время выполнения программы SQL-операторы можно изменять в зависимости от конкретных условий. Если открыть окно String List Editor для свойства TQuery.SQL и ввести любой оператор (например, приведенный ниже), то будет создан статический SQL-оператор:

```
SELECT * FROM EMPLOYEE WHERE COUNTRY = "USA"
```

Это оператор не будет изменяться, если только не заменить его полностью во время выполнения приложения.

Для превращения приведенного выше статического оператора в динамический необходимо ввести его следующим образом

```
SELECT * FROM CUSTOMER WHERE COUNTRY = :iCOUNTRY;
```

В этом операторе вместо жестко заданного значения критерия поиска используется параметр, значение которого может быть определено позже. Эта переменная называется iCountry и следует за двоеточием в операторе SELECT (ее имя было выбрано совершенно произвольно). Теперь вы можете выполнять поиск по любой стране, предварительно указав ее название.

Существует несколько способов ввода значений в параметрические запросы. Один из них заключается в использовании редактора свойств для свойства TQuery.Params, а другой — в изменении значения непосредственно во время работы программы. Кроме того, с помощью компонента TDataSource можно использовать значение, выбранное из некоторого другого набора данных.

Установка параметров с помощью редактора свойств

При открытии в окне инспектора объектов редактора свойства TQuery.Params, в списке Parameter Name отображаются параметры введенного запроса. Для каждого из этих параметров следует выбрать тип в раскрывающемся списке Data Type. Кроме того, при необходимости в соответствующем поле можно задать начальное значение параметра. Выбор флажка опции Null вызовет присвоение параметру этого значения. По щелчку на кнопке OK запрос подготовит параметры, связав их с заданными типами (читайте врезку "Подготовка запросов"). При вызове метода TQuery.Open() компоненту TQuery будет возвращен результирующий набор.

Подготовка запросов

Когда SQL-запрос отсылается серверу, последний должен проанализировать его, проверить корректность, откомпилировать и выполнить. Это происходит всякий раз при отправке SQL-оператора на сервер. Однако можно существенно увеличить производительность, позволив серверу выполнить предварительные действия по анализу, проверке и компиляции, т.е. выполнив всю предварительную подготовку SQL-оператора еще до того, как он будет отправлен серверу для выполнения. Подготовительные действия особенно уместны в случае циклического запроса. Подготовка SQL-оператора осуществляется с помощью вызова метода TQuery.Prepare(), как показано в следующем фрагменте кода:

```
Query1.Prepare;              // Предварительная подготовка запроса
try
  { Выполнение запроса в цикле }
  for i := 1 to 100 do begin
    { Определение параметров запроса }
    Query1.ParamByName('SomeParam').AsInteger := i;
    Query1.ParamByName('SomeOtherParam').AsString := SomeString;
    Query1.Open;         // Открытие запроса
```

```
    try
      { Здесь можно использовать полученный результат }
    finally
      Query1.Close;    // Закрытие запроса
    end;
  end;
finally
  Qiery1.Unprepare;   // Освобождение ресурсов
end;
```

Метод Prepare() должен быть вызван только один раз перед использованием запроса. Более того, можно изменить значения параметров запроса после первого вызова метода Prepare(); повторный вызов при этом не нужен. Однако, если изменить сам SQL-оператор в запросе, необходимо будет снова вызывать метод Prepare(). Вызов этого метода всегда должен сопровождаться вызовом метода TQuery.UnPrepare(), предназначенного для освобождения ресурсов, занятых при использовании метода Prepare().

Запросы можно подготовить, щелкнув на кнопке OK в редакторе свойства Params или вызвав метод TQuery.Prepare(), как показано в приведенном выше фрагменте кода. Для запросов, SQL-операторы которых не могут быть изменены, рекомендуем вызывать метод Prepare() в обработчике события формы OnCreate и метод UnPrepare() в обработчике ее события OnDestroy. SQL-операторы запросов можно и не подготавливать, однако выигрыш от подготовки слишком значительный, чтобы пренебрегать им.

Установка параметров запроса с помощью свойства Params

Компонент TQuery содержит массив (индексы которого начинаются с 0) объектов TParam, представляющих собой параметры SQL-оператора, помещенного в свойство TQuery.SQL. Рассмотрим следующий оператор SQL:

```
INSERT INTO COUNTRY (
  NAME,
  CAPITAL,
  POPULATION)
VALUES(
  :NAME,
  :CAPITAL,
  :POPULATION)
```

Для установки значений параметров :NAME, :CAPITAL и :POPULATION свойство Params используется следующим образом:

```
with Query1 do begin
  Params[0].AsString: = 'Peru';
  Params[1].AsString: = 'Lima';
  Params[2].AsInteger: = 22,000,000;
end;
```

Используемые значения будут связаны с параметрами в SQL-операторе. Имейте в виду, что порядок указания параметров в SQL-операторе определяет и их положение в массиве свойства Params.

Установка параметров запроса с помощью метода ParamByName

Кроме свойства Params, компонент TQuery содержит метод ParamByName(), который позволяет устанавливать значения параметров SQL-операторов по их именам, а не по порядку перечисления в операторе. Это делает текст программы более удобочитаемым, однако данный метод не так эффективен, как позиционный, поскольку при этом дополнительно осуществляется разрешение ссылок.

Использование метода ParamByName() в случае рассмотренного ранее запроса выглядит следующим образом:

```
with Query1 do begin
  ParamByName('COUNTRY').AsString: = 'Peru';
  ParamByName('CAPITAL').AsString: = 'Lima';
  ParamByName('POPULATION').AsInteger: = 22,000,000;
end;
```

Установка параметров запроса из другого набора данных

Параметры, передающиеся компоненту TQuery, могут быть получены из другого компонента типа TDataset, такого как TQuery или TTable. При этом между двумя наборами данных создается связь типа "главный–подчиненный". Вначале необходимо связать компонент TDataSource с главным набором данных. Имя компонента TDataSource назначается свойству DataSource подчиненного компонента TQuery. Когда запрос выполнен, Delphi определяет, существует ли значение, установленное для свойства TQuery.DataSource. Если да, осуществляется поиск имен столбцов в компоненте DataSource, которые совпадают с именами параметров в операторе SQL, и их связывание с параметрами.

В качестве примера рассмотрим следующий оператор SQL:

```
SELECT * FROM SALARY_HISTORY
    WHERE EMP_NO = :EMP_NO
```

В данном примере необходимо определить значение параметра с именем EMP_NO. Предварительно необходимо назначить объект TDataSource, ссылающийся на главный компонент TTable, свойству TQuery компонента DataSource. В этой таблице будет проведен поиск поля с именем EMP_NO, а затем значение этого поля в текущей записи будет использовано как значение параметра запроса. Этот прием иллюстрируется в проекте LnkQuery.dpr, содержащемся на прилагаемом компакт-диске.

Функция Format и динамический SQL

Теперь, когда вы познакомились с параметрическими запросами, следующие операторы могут показаться вам вполне корректными:

```
SELECT * FROM ORDER BY :ORDERVAL;
SELECT * FROM :TABLENAME
```

Но, к сожалению, некоторые элементы, например, такие как имена столбцов и таблиц, в SQL-операторе нельзя заменять параметрами. SQL-серверы не поддерживают такую возможность. Каким же образом придать динамическим SQL-операторам подобную дополнительную гибкость? Для этого на этапе разработки следует конструировать SQL-опeратры с помощью функции Format().

Если у вас имеется опыт программирования на языках C или C++, то вы можете заметить, что функция Format() очень похожа на функцию printf() из языка C.

Использование функции Format()

Функция Format() применяется для создания строк, значения которых изменяются в зависимости от значений предоставляемых *определителями формата*. Последние помещаются в качестве держателей места в места строки-шаблона, где преобразованные значения-параметры должны быть вставлены в результирующую строку. Эти определители состоят из символа процента (%) и *определителя типа*. Ниже приведено несколько определителей типа:

Спецификатор	Назначение
c	Определитель символьного типа
d	Определитель целого типа
f	Определитель вещественного типа
p	Определитель указателя
s	Определитель строки

Например, в строке "My name is %s and I'm %d years old." используется два определителя формата. Определитель %s показывает, что в данное место будет вставлена строка, а %d — целое число. Для получения результирующей строки функция Format() используется следующим образом:

```
S := Format('My name is %s and I'm %d years old.', ['Xavier', 32]);
```

Функции Format() в качестве параметров передается строка-шаблон и открытый массив аргументов, предназначенных для подстановки вместо определителей формата. Функция замещает определители формата значениями из массива параметров и возвращает полученную таким образом строку. Более подробную информацию о функции Format() можно получить в интерактивной справочной системе Delphi.

Таким образом, чтобы спроектировать операторы SQL и обеспечить возможность модификации имен полей или таблиц, необходимо использовать функцию Format() так, как показано в следующих примерах.

В листинге 29.8 показано, как использовать функцию Format(), чтобы разрешить пользователю выбирать поля сортировки результирующего набора. Список полей содержится в окне списка, а прелагаемый текст — это текст события OnClick данного окна списка. Этот демонстрационный пример содержится в проекте OrderBy.dpr на прилагаемом компакт-диске.

Листинг 29.8. Использование функции Format() для задания столбца сортировки

```
procedure TMainForm.lbFieldsClick(Sender: TObject);
{ Определение строки, из которой будет создана SQL-строка }
const
  SQLString = 'SELECT * FROM PARTS ORDER BY %s';
begin
  with qryParts do
  begin
    Close;           // Убедитесь, что запрос закрыт
    SQL.Clear;       // Удалите предыдущие операторы SQL
    { Теперь добавьте новый SQL-оператор, созданный функцией Format. }
    SQL.Add(Format(SQLString, [lbFields.Items[lbFields.ItemIndex]]));
    Open;      { Теперь откроем Query1 с новым SQL-оператором. }
  end;
end;
```

Для заполнения окна списка (листинг 29.8) именами полей поместите в обработчик события OnCreate формы следующий текст:

```
tblParts.Open;
try
    tblParts.GetFieldNames(lbFields.Items);
finally
    tblParts.Close;
end;
```

Предполагается, что объект tblParts связан с таблицей PARTS.DB.

В листинге 29.9 показано, как выбрать таблицу, к которой будет применен некоторый оператор SELECT. Код этого примера практически идентичен коду, приведенному в листинге 29.8, различаются лишь строки формата, а обработчик события OnCreate формы получает список имен таблиц вместо списка имен полей.

Вот как получается список имен таблиц:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
{ Сначала необходимо получить список имен таблиц }
  Session.GetTableNames(dbMain.DatabaseName, '', False, False, lbTables.Items);
end;
```

Для выбора таблицы, к которой будет применен запрос SELECT, используется обработчик события lbTables.OnClick, как показано в листинге 29.9.

Листинг 29.9. Использование функции Format() для выбора таблицы

```
procedure TMainForm.lbTablesClick(Sender: TObject);
{ Определение строки, из которой будет создана SQL-строка }
const
  SQLString = 'SELECT * FROM %s';
begin
  with qryMain do
  begin
    Close;           // Убедитесь, что запрос закрыт
    SQL.Clear;       // Удалите все предыдущие SQL-операторы
    { Теперь добавьте новый SQL-оператор, созданный с помощью функции Format }
    SQL.Add(Format(SQLString, [lbTables.Items[lbTables.ItemIndex]]));
    Open;       { Теперь откроем Query1 с новым оператором }
  end;
end;
```

Этот демонстрационный пример содержится в проекте SelTable.dpr на прилагаемом компакт-диске.

Возвращение значений из результирующего набора с помощью объекта TQuery

После завершения выполнения запроса и возвращения результирующего набора данных, можно обращаться к значениям столбцов, используя компонент TQuery так, как если бы это был массив, в котором имена полей являются индексами. Пусть, например, свойство SQL компонента TQuery содержит следующий SQL-оператор:

```
SELECT * FROM CUSTOMER
```

Получить значения столбцов после выполнения запроса можно так, как показано в листинге 29.10. Этот же код содержится в проекте ResltSet.dpr на прилагаемом компакт-диске.

Листинг 29.10. Выборка полей из результирующего набора данных компонента TQuery

```
procedure TMainForm.dsCustomerDataChange(Sender: TObject; Field: TField);
begin
  with lbCustomer.Items do
  begin
    Clear;
    Add(VarToStr(qryCustomer['CustNo']));
    Add(VarToStr(qryCustomer['Company']));
    Add(VarToStr(qryCustomer['Addr1']));
    Add(VarToStr(qryCustomer['City']));
    Add(VarToStr(qryCustomer['State']));
    Add(VarToStr(qryCustomer['Zip']));
    Add(VarToStr(qryCustomer['Country']));
    Add(VarToStr(qryCustomer['Phone']));
    Add(VarToStr(qryCustomer['Contact']));
  end;
end;
```

Для получения доступа к значениям полей результирующего набора запроса qryCustomer в приведенном примере применен метод FieldValues(). Поскольку FieldValues() — это метод, используемый по умолчанию, его не надо указывать явно, как в следующей строке кода:

```
Add(VarToStr(qryCustomer.FieldValues['Contact']));
```

Функция FieldValues() возвращает поле типа variant. Если поле содержит значение NULL, то при попытке получить доступ к его значению с помощью функции FieldValues() будет сгенерирована исключительная ситуация EVariantError. Поэтому в Delphi имеется функция VarToStr(), которая преобразует строковое значение NULL в пустую строку. Для других типов аналогичных функций не существует, однако можно создать собственные, по типу конструкции, предложенной ниже для целого типа данных:

```
function VarToInt(const V: Variant): Integer
begin
  if TVarData(V).VType <> varNull then
    Result := V
  else
    Result := 0;
end;
```

Будьте осторожны при повторном сохранении данных. В базах данных SQL значения NULL вполне корректны. Если вы замените эти данные пустыми строками, которые не равны NULL, то можете нарушить целостность данных. Возможно, потребуется найти какое-либо общее решение — например, во время выполнения проверять сохраняемые в базе значения на наличие NULL и заменять их некоторой заранее определенной строкой.

Кроме того, можно получать значения полей из компонента TQuery с помощью свойства TQuery.Fields. Свойство Fields используется так же, как и свойство TQuery.Params, за исключением того, что оно ссылается на столбцы в результирующем наборе данных. Кроме того, компонент Tquery включает метод FieldByName(), функционально аналогичный методу ParamByName().

Свойство UniDirectional

Для оптимизации доступа к базе данных в компоненте TQuery предусмотрено свойство UniDirectional. Это свойство применяется к базам данных, поддерживающим *двунаправленные курсоры*, которые позволяют перемещаться по результирующему набору запроса как вперед, так и назад. По умолчанию это свойство установлено равным False. Поэтому при использовании компонентов типа TDBGrid, связанных с базой данных, не поддерживающей двунаправленное перемещение, Delphi эмулирует подобное перемещение путем буферизации записей на стороне клиента, что резко повышает потребление ресурсов. Следовательно, если вы планируете перемещаться по результирующему набору только вперед, установите свойство UniDirectional в True.

Обновляемые результирующие наборы

По умолчанию компонент TQuery возвращает результирующие наборы "только для чтения". Однако вы можете потребовать, чтобы он возвращал модифицируемый результирующий набор, установив свойство TQuery.RequestLive равным True. Однако при этом сохраняются некоторые ограничения, описанные ниже.

Для запросов, возвращающих результирующие наборы из таблиц dBASE или Paradox, эти ограничения таковы.

- Используется локальный синтаксис SQL (информацию по этому вопросу можно найти в интерактивной справочной системе).
- Используется только одна таблица.
- В SQL-операторе не допускается конструкция ORDER BY.
- В SQL-операторе не допускаются групповые функции типа SUM или AVG.
- В SQL-операторе не допускаются вычисляемые поля.
- В предложении WHERE сравнения могут состоять только из имен столбцов скалярного типа.

Для запросов, использующих режим пересылаемых SQL-операторов для таблицы на сервере, имеются следующие ограничения.

- Используется только одна таблица.
- В SQL-операторе не допускается предложение ORDER BY.
- В SQL-операторе не допускаются групповые функции типа SUM или AVG.

Чтобы определить, могут ли модифицироваться результаты выполнения того или иного запроса, проанализируйте значение свойства TQuery.CanModify.

Кэшируемые обновления

Компонент TDataSet содержит свойство CachedUpdate, которое позволяет превращать любой запрос или хранимую процедуру в обновляемое представление. Это означает, что изменения в наборе данных записываются во временный буфер клиента, а не сервера. Эти изменения затем могут быть переданы на сервер с помощью вызова метода ApplyUpdates() компонента TQuery или TStoredProc. Кэшируемые обновления позволяют оптимизировать модификации. Более подробно кэшируемые обновления описаны в главе 13 руководства "Delphi 5 Database Application Developer's Guide".

Выполнение хранимых процедур

Оба компонента TStoredProc и TQuery Delphi 5 способны выполнять хранимые процедуры на сервере. В следующих разделах речь идет о том, как это делается.

Использование компонента TStoredProc

Компонент TStoredProc позволяет выполнять хранимые процедуры на сервере. В зависимости от типа сервера он может возвращать один или несколько результирующих наборов. Этот компонент может выполнять также хранимые процедуры, которые вообще не возвращают никаких данных. Для выполнения на сервере хранимых процедур необходимо установить следующие свойства компонента TStoredProc.

Свойство	Описание
DataBaseName	Имя базы данных, которая содержит хранимую процедуру. Обычно на требуемую базу данных на сервере указывает значение свойства DataBaseName компонента TDatabase
StoredProcName	Имя той хранимой процедуры, которая должна быть выполнена
Params	Входные и выходные параметры, определенные в хранимой процедуре. Порядок следования параметров должен соответствовать определению хранимой процедуры на сервере

Входные и выходные параметры компонента TStoredProc

Входные и выходные параметры определяются с помощью свойства TStoredProc.Params. Как и в случае компонента TQuery, параметры должны быть *подготовлены* с заданными по умолчанию типами данных. Это можно выполнить либо во время разработки с помощью редактора параметров, либо во время выполнения, как будет показано ниже.

Для подготовки параметров с помощью редактора параметров щелкните правой кнопкой мыши на компоненте TStoredProc. Раскроется окно Parameters Editor редактора параметров.

В списке Parameters Name этого окна отображаются все входные и выходные параметры хранимой процедуры. Не забывайте, что, прежде чем этот список сможет быть заполнен данными, полученными от сервера, в параметре StoredProcName следует указать имя хранимой процедуры. Для каждого параметра необходимо выбрать тип данных в раскрывающемся списке Data Type. Дополнительно каждому параметру можно задать исходное значение (в том числе значение NULL), как и компоненту TQuery. После щелчка на кнопке OK все параметры будут подготовлены.

Кроме того, можно подготовить параметры компонента TStoredProc и во время выполнения — с помощью метода TStoredProc.Prepare(). Этот метод функционирует точно так же, как и метод Prepare() компонента TQuery, рассмотренного выше в этой главе.

Выполнение хранимой процедуры, не возвращающей результирующий набор

Для иллюстрации выполнения хранимой процедуры, которая не возвращает результирующий набор, рассмотрим листинг 29.11. В нем приведена хранимая процедура для сервера InterBase, добавляющая запись в таблицу COUNTRY.

Листинг 29.11. Добавление записи в таблицу COUNTRY с помощью хранимой процедуры InterBase

```
CREATE PROCEDURE ADD_COUNTRY(
iCOUNTRY        VARCHAR(15),
iCURRENCY       VARCHAR(10)
)
AS
BEGIN
    INSERT INTO COUNTRY(COUNTRY, CURRENCY)
    VALUES (:iCOUNTRY, :iCURRENCY);
    SUSPEND;
END
^
```

Для выполнения этой хранимой процедуры из среды приложения Delphi необходимо сначала настроить компонент TStoredProc, установив соответствующие значения параметров, как было указано выше. Эта операция включает определение типов параметров в окне редактора параметров. Код для выполнения этой хранимой процедуры приведен в листинге 29.12.

Листинг 29.12. Выполнение хранимой процедуры с помощью компонента TStoredProc

```
with spAddCountry do
  begin
    ParamByName('iCOUNTRY').AsString := edtCountry.Text;
    ParamByName('iCURRENCY').AsString := edtCurrency.Text;
    ExecProc;
    edtCountry.Text := '';
    edtCurrency.Text := '';
    tblCountries.Refresh;
  end;
```

В приведенном фрагменте программы сначала с помощью метода ParamByName() устанавливаются значения двух параметров TEdit компонента TStoredProc, а затем вызывается функция TStoredProc.ExecProc(), которая и выполняет хранимую процедуру. Пример, в котором содержится приведенный фрагмент кода, можно найти в проекте AddCntry.dpr на прилагаемом компакт-диске.

На заметку

Для запуска проекта AddCntry.dpr сначала необходимо с помощью утилиты BDEADMIN.EXE установить новый псевдоним "DB". Этот псевдоним должен быть связан с файлом \CODE\DATA\DDGIB.GDB, содержащимся на прилагаемом компакт-диске. За более подробной информацией об утилите BDE Administrator обратитесь к ее документации.

Получение результирующего набора хранимой процедуры из компонента TQuery

Хранимую процедуру можно также выполнить, используя режим пересылки SQL-оператора компонента TQuery. Это может быть необходимо, например, при использовании сервера InterBase, который не поддерживает вызов хранимых процедур в операторе SELECT. Хранимая процедура, которая возвращает результирующие наборы, может вызываться так же, как если бы это была обычная таблица. В листинге 29.13 приведен текст хранимой процедуры сервера InterBase, которая возвращает список сотрудников отдела из таблицы EMPLOYEE. Отдел определяется входным параметром iDEPT_NO.

Листинг 29.13. Хранимая процедура GET_EMPLOYEES_BY_DEPT

```
CREATE PROCEDURE GET_EMPLOYEES_IN_DEPT (
iDEPT_NO            CHAR(3))
RETURNS(
EMP_NO            SMALLINT,
FIRST_NAME        VARCHAR(15),
LAST_NAME         VARCHAR(20),
DEPT_NO           CHAR(3),
HIRE_DATE         DATE)
AS
BEGIN
  FOR SELECT
    EMP_NO,
    FIRST_NAME,
    LAST_NAME,
    DEPT_NO,
    HIRE_DATE
  FROM EMPLOYEE
  WHERE DEPT_NO = :iDEPT_NO
  INTO
    :EMP_NO,
    :FIRST_NAME,
    :LAST_NAME,
    :DEPT_NO,
    :HIRE_DATE
  DO
    SUSPEND;
END ^
```

Чтобы выполнить эту хранимую процедуру в Delphi 5, необходимо использовать компонент TQuery, у которого свойство SQL будет иметь следующее значение:

```
SELECT * FROM GET_EMPLOYEES_IN_DEPT(
:iDEPT_NO)
```

Обратите внимание, что в данном случае оператор SELECT используется так, как если бы процедура была таблицей. Как видите, разница между хранимой процедурой и таблицей состоит в наличии входного параметра iDEPT_NO.

Теперь разработаем небольшой проект, иллюстрирующий выполнение описанной хранимой процедуры.

Проект qryGetEmployees содержит компонент TQuery, выполняющий хранимую процедуру, приведенную в листинге 29.13. Он получает параметр из объекта qryDepartment, который выполняет обычный оператор SELECT в таблице DEPARTMENT базы данных. Объект qryGetEmployees связан с объектом dbgEmployees, который отображает прокручиваемый список отделов. Когда пользователь прокручивает список dbgDepartment, вызывается обработчик события OnDataChange объекта dsDepartment. Обратите внимание на то, что объект dsDepartment связан с объектом qryDepartment. Этот обработчик события выполняет код, приведенный в листинге 29.14, который устанавливает значение параметра объекта qryGetEmployees и получает его результирующий набор.

Листинг 29.14. Обработчик события OnChange объекта DataSource1

```
procedure TMainForm.dsDepartmentDataChange(Sender: TObject; Field: TField);
begin
  with qryGetEmployees do
  begin
    Close;
    ParamByName('iDEPT_NO').AsString := qryDepartment['DEPT_NO'];
    Open;
  end;
end;
```

Для чего может понадобиться получать информацию с помощью хранимой процедуры, а не обычным путем — с помощью выполнения простой операции SELECT в таблице? Предположим, что есть сотрудники с различным уровнем допуска внутри отдела и им необходим доступ к информации. Если бы они имели прямой доступ к таблице, то могли бы увидеть конфиденциальную информацию, например, о заработной плате. Ограничивая доступ к таблице, но в то же время обеспечивая пользователей необходимой информацией с помощью хранимых процедур и представлений, можно не только организовать достаточную защиту, но и создать более гибкий набор бизнес-правил для базы данных.

Резюме

В этой главе рассмотрен процесс разработки приложений клиент/сервер. Подробно обсуждались элементы, составляющие систему клиент/сервер, было проведено сравнение процесса разработки приложений клиент/сервер с традиционными методологиями разработки локальных баз данных. Здесь вы также познакомились с различными методами использования Delphi 5 и сервера InterBase при разработке проектов клиент/сервер.

Глава

30

Расширения баз данных VCL

Использование BDE 651

Таблицы dBASE 653

Таблицы Paradox 657

Расширение возможностей компонента `TDataSet` 674

Резюме 699

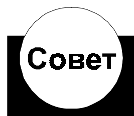
В своей основе архитектура базы данных библиотеки VCL разработана для взаимодействия с внешним миром с помощью механизма Borland Database Engine (BDE), обеспечивающего разнообразные и надежные средства работы с базами данных. Библиотека VCL представляет собой некий “интерфейс” между пользователем и базами данных, благодаря которому обеспечивается унифицированный способ доступа к различным типам баз данных. Хотя такая технология и повышает надежность, масштабируемость и простоту использования, у нее есть и свой недостаток: возможности, специфические для конкретной базы данных, библиотекой VCL, как правило, не поддерживаются. В этой главе речь пойдет о том, как расширить библиотеку VCL за счет непосредственного взаимодействия с механизмом BDE и другими источниками данных. Благодаря этому можно использовать новые функциональные возможности баз данных, ранее не доступные в Delphi.

Использование BDE

При создании приложений, которые напрямую обращаются к механизму BDE, следует помнить о некоторых эмпирических правилах. В этом разделе представлена общая информация, знание которой необходимо для использования интерфейса BDE API в приложениях Delphi.

Модуль BDE

Все функции, типы и константы BDE определены в модуле `BDE`. Он должен быть указан в операторе `uses` любого модуля, в котором выполняется обращение к механизму BDE. Кроме того, часть интерфейса модуля `BDE` доступна в файле `BDE.INT`, который находится в каталоге `..\Delphi 5\Doc`. Этот файл можно использовать как справочник доступных функций и типов данных BDE.



Дополнительная информация о программировании с использованием интерфейса BDE API содержится в файле справки `BDE32.hlp`, расположенном в каталоге BDE (по умолчанию это каталог `\Program Files\Borland\Common Files\BDE`). В этом файле можно найти подробные сведения обо всех функциях BDE API, а также прекрасные примеры на языках Object Pascal и C.

Функция Check()

Все функции BDE возвращают значение типа `DBIRESULT`, которое показывает, успешным ли был вызов функции. Вместо выполнения громоздкого процесса проверки результата каждого вызова функции BDE, Delphi предлагает процедуру `Check()` с параметром типа `DBIRESULT`. Если значение переданного ей параметра не соответствует успешному завершению вызова функции, эта процедура генерирует исключительную ситуацию. В приведенном ниже фрагменте кода показано, как нужно и как не нужно вызывать функцию BDE.

```
// !!! Никогда не делайте так:
var
  Rez: DBIRESULT;
  A: array[0..dbiMaxUserNameLen] of char;
begin
  Rez := dbiGetNetUserName(A);    // Вызов функции BDE
```

```
  if Rez <> DBIERR_NONE then      // Обработка ошибки
    // Ошибка обрабатывается здесь
  else begin
    // Продолжение функции
  end;
end;

// !!! Всегда делайте так:
var
  A: array[0..dbiMaxUserNameLen] of char;
begin
  { Вызывать функцию BDE и обрабатывать ошибку нужно одновременно.
    В случае возникновения ошибки генерируется исключение. }
  Check(dbiGetNetUserName(A));
  // Продолжение работы
end;
```

Курсоры и дескрипторы

Многим функциям BDE в качестве параметров передаются дескрипторы курсоров или баз данных. Строго говоря, *дескриптор курсора* (cursor handle) — это объект BDE, представляющий некоторый набор данных, позиционированный на определенной строке этих данных. Тип данных дескриптора курсора — `hDBICur`. В Delphi это понятие трактуется как "текущая запись в конкретной таблице или в результатах выполнения некоторого запроса или хранимой процедуры". Дескриптор курсора сохраняется в свойстве `Handle` компонентов `TTable`, `TQuery` и `TStoredProc`. Не забывайте передавать содержимое свойства `Handle` подобных объектов в любую из функций BDE, требующих указания дескриптор курсора.

Некоторые функции BDE используют также дескриптор базы данных. Дескриптор базы данных BDE имеет тип `hDBIDb` и представляет собой некоторую открытую базу данных — локальный или сетевой каталог, если используется dBASE или Paradox, либо файл базы данных сервера, как в случае баз данных SQL. Этот дескриптор можно получить из свойства `Handle` компонента `TDatabase`. Если работа ведется без подключения к базе данных с помощью объекта `TDatabase`, то в свойстве `DBHandle` компонентов `TTable`, `TQuery` и `TStoredProc` также будет содержаться этот дескриптор.

Синхронизация курсоров

Как известно, в Delphi для открытого набора данных используется понятие текущей записи, в то время как в BDE — понятие курсора, который указывает на некоторую определенную запись в наборе данных. Поскольку для оптимизации производительности в Delphi организуется кэширование записей, синхронизация текущей записи с основным BDE-курсором иногда нарушается. Как правило, это не вызывает никаких проблем, поскольку подпрограммы библиотеки VCL автоматически предпринимают нужные действия. Однако, если BDE-функция, которой в качестве параметра передается курсор, вызывается напрямую, то необходимо гарантировать, что текущая позиция курсора Delphi синхронизирована с основным BDE-курсором. Это звучит устрашающе, но выполняется довольно просто. Достаточно вызвать метод `UpdateCursorPos()` компонента, производного от класса `TDataSet`.

Аналогично, после вызова BDE-функции, изменяющей позицию основного курсора, подпрограммам библиотеки VCL необходимо сообщить о том, что текущее положение записи

требуется заново синхронизировать с BDE. Для этого сразу же после вызова функции BDE необходимо вызвать метод CursorPosChanged() любого компонента, производного от класса TDataSet. В приведенном ниже фрагменте кода показано, как использовать упомянутые функции синхронизации курсора:

```
procedure DoSomethingWithTable (T: TTable);
begin
  T.UpdateCursorPos;
  // Вызов функции BDE, изменяющей положение курсора
  T.CursorPosChanged;
end;
```

Таблицы dBASE

Таблицы dBASE обладают несколькими полезными возможностями, которые напрямую Delphi не поддерживаются. К ним относится поддержка уникального физического номера каждой записи, возможность "мягкого" удаления записей (без их непосредственного удаления из таблицы), восстановление удаленных таким образом записей и упаковка таблицы для физического удаления помеченных записей. В этом разделе вы узнаете о функциях BDE, используемых для выполнения описанных выше действий. Затем мы создадим потомок компонента TTable — TdBaseTable, который включает поддержку всех перечисленных возможностей.

Физический номер записи

Таблицы dBASE поддерживают уникальный физический номер каждой записи в таблице. Этот номер представляет собой физическое местоположение записи относительно начала таблицы (независимо от того, какой индекс применен к таблице в данный момент). Для того чтобы получить физический номер записи, необходимо вызвать BDE-функцию DbiGetRecord, которая определяется следующим образом:

```
function DbiGetRecord (hCursor: hDBICur; eLock: DBILockType;
  PRecBuff: Pointer; precProps: pRECProps): DBIResult stdcall;
```

Параметр hCursor — это дескриптор курсора. Чаще всего в качестве этого параметра указывается свойство Handle одного из компонентов, производных от класса TDataSet .

Параметр eLock — это необязательный параметр, задающий тип блокировки записи. Он имеет тип DBILockType, который является перечислением и определяется следующим образом:

```
type
  DBILockType = (
    dbiNOLOCK,       // Нет блокировки (по умолчанию)
    dbiWRITELOCK,    // Блокировка записи
    dbiREADLOCK);    // Блокировка чтения
```

В нашем случае блокировку записи устанавливать не нужно, поскольку содержимое записи изменяться не будет. Поэтому необходимо выбрать значение dbiNOLOCK.

Параметр pRecBuff — это указатель на буфер записи. Поскольку нужно получить только свойства записи, а не сами данные, то для этого параметра необходимо задать значение Nil.

Параметр pRecProps представляет собой указатель на запись типа RECProps, которая определяется следующим образом:

```
type
  PRECProps = ^RECProps;
  RECProps = packed record     // Свойства записи
    iSeqNum        : Longint;  // Если поддерживается только Seq#
    iPhyRecNum     : Longint;  // Если поддерживается только Phy Rec#
    iRecStatus     : Word;     // Состояние отложенного обновления записи
    bSeqNumChanged : WordBool; // Не используется
    bDeleteFlag    : WordBool; // Если поддерживается только "мягкое" удаление
  end;
```

Как видите, из этой записи можно получить различную информацию. В данном конкретном случае нас интересует только поле iPhyRecNum, применяемое лишь при использовании таблиц dBASE и FoxPro.

Обобщив сказанное, можно создать метод TdBaseTable, возвращающий физический номер текущей записи:

```
function TdBaseTable.GetRecNum: Longint;
{ Возвращает физический номер текущей записи. }
var
  RP: RECProps;
begin
  UpdateCursorPos;         // Обновление BDE из Delphi
  { Получение текущих свойств записи }
  Check(dbiGetRecord(Handle, dbiNOLOCK, Nil, @RP));
  Result := RP.iPhyRecNum; // Возврат значения из полученных свойств
end;
```

Просмотр удаленных записей

Просмотреть записи, которые были мягко удалены из таблицы dBASE, очень просто — необходимо вызвать функцию dbiSetProp() интерфейса BDE API. Эта функция является очень мощным средством, позволяющим изменять различные свойства нескольких типов объектов BDE. Полное описание этой функции и ее работы можно найти в интерактивной справочной системе BDE, в разделе "Properties — Getting and Setting" ("Свойства — получение и установка значений"). Эта функция определяется следующим образом:

```
function DbiSetProp(hObj: hDBIObj; iProp: Longint;
  iPropValue: Longint): DBIResult stdcall;
```

Параметр hObj представляет собой дескриптор некоторого типа объекта BDE. В данном случае это дескриптор курсора.

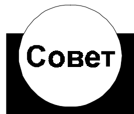
В качестве параметра iProp должен быть указан идентификатор устанавливаемого свойства. Полный список свойств можно найти в справочной системе BDE. Для разрешения/запрещения просмотра удаленных записей предназначен идентификатор curSOFTDELETEON.

Новое значение выбранного свойства определяется параметром iPropValue. В данном случае это значение имеет тип Boolean (0 запрещает просмотр, а 1 — разрешает).

В следующем фрагменте кода показан метод SetViewDeleted() компонента TdBaseTable:

```
procedure TdBaseTable.SetViewDeleted(Value: Boolean);
{ Позволяет пользователю переключаться между режимами
  отображения/сокрытия удаленных записей. }
begin
  { Таблица должна быть активна. }
  if Active and (FViewDeleted <> Value) then begin
    DisableControls; // Для устранения мерцания
    try
      { Вызов BDE для переключения между режимами просмотра. }
      Check(dbiSetProp(hDBIObj(Handle), curSOFTDELETEON, Longint(Value)));
    finally
      Refresh;       // Обновление Delphi
      EnableControls;
    end;
    FViewDeleted := Value
  end;
end;
```

В этом методе сначала проверяется, открыта ли таблица и отличается ли значение, которое будет установлено, от текущего значения поля FViewDeleted объекта. Затем вызывается метод DisableControls(), для того чтобы убрать нежелательное мерцание любых связанных с этой таблицей элементов управления, предназначенных для работы с данными. После этого вызывается функция dbiSetProp() (обратите внимание на преобразование параметра Handle к типу hDBIObj). Тип hDBIObj можно рассматривать как нетипизированный дескриптор некоторого объекта BDE. И, наконец, набор данных обновляется, и любые используемые элементы управления заново активизируются.



Всякий раз при использовании функции DisableControls() для приостановки взаимодействия наборов данных и подключенных к ним управляющих элементов необходимо использовать блок try..finally. Это гарантирует, что функция EnableControls() будет вызвана в любом случае, даже если произойдет ошибка.

Проверка записи на удаленность

При просмотре набора данных, в котором имеются удаленные записи, в процессе перемещения по этому набору, вероятно, потребуется уметь определять, какая из записей является удаленной, а какая — нет. Выше уже рассматривалось, как можно выполнить подобную проверку. Эту информацию можно получить с помощью функции DbiGetRecord(), которая ранее использовалась для получения физического номера записи. Описываемая процедура приведена в следующем фрагменте кода. Единственное различие между этой процедурой и процедурой GetRecNum() — проверка в записи RECProps поля bDeletedFlag, а не поля iPhyRecNo:

```
function TdBaseTable.GetIsDeleted: Boolean;
{ Определяет, удалена ли текущая запись. }
var
  RP: RECProps;
```

```
begin
  if not FViewDeleted then // Если удаленные записи не видны, то
    Result := False        // не стоит беспокоиться
  else begin
    UpdateCursorPos;  // Обновление BDE из Delphi
    { Получение свойств текущей записи }
    Check(DbiGetRecord(Handle, dbiNOLOCK, Nil, @RP));
    Result := RP.bDeleteFlag;  // Возврат флага
  end;
end;
```

Восстановление ранее удаленной записи

Теперь мы уже знаем, как просмотреть удаленные записи и как определить, была ли данная запись удалена. Безусловно, вам известно и то, как запись можно удалить. Осталось выяснить, как восстановить ранее удаленную запись. К счастью, благодаря функции BDE `dbiUndeleteRecord()`, определение которой приводится ниже, эта операция выполняется очень просто.

```
function dbiUndeleteRecord (hCursor: hDBICur): DBIResult stdcall;
```

Ее единственным параметром является дескриптор курсора текущего набора данных. Используя эту функцию, метод `UndeleteRecord()` компонента `TdBaseTable` можно создать следующим образом:

```
procedure TdBaseTable.UndeleteRecord;
begin
  if not IsDeleted then
    raise EDatabaseError.Create('Record is not deleted');
  Check(dbiUndeleteRecord(Handle));
  Refresh;
end;
```

Упаковка таблицы

Для того чтобы физически удалить из таблицы dBASE мягко удаленные записи, таблицу следует *упаковать* (pack). Для выполнения этой операции можно воспользоваться BDE-функцией `dbiPackTable()`, которая определяется следующим образом:

```
function DbiPackTable(hDb: hDBIDb; hCursor: hDBICur;
  pszTableName: PChar; pszDriverType: PChar;
  bRegenIdxs: Bool): DBIResult stdcall;
```

Параметр `hDb` представляет собой дескриптор базы данных. В качестве этого параметра необходимо передать свойство `DBHandle` компонента, производного от класса `TDataSet`, или свойство `Handle` компонента `TDatabase`.

Параметр `hCursor` является дескриптором курсора. В качестве этого параметра нужно передать свойство `Handle` компонента, производного от класса `TDataSet`. Если вместо параметра `hCursor` для идентификации таблицы используются параметры `pszTableName` и `pszDriverType`, то в качестве параметра `Handle` можно также предавать значение `Nil`.

Параметр pszTableName — это указатель на строку, содержащую имя таблицы.

Параметр pszDriverType является указателем на строку, определяющую тип драйвера таблицы. Если значение параметра hCursor равно Nil, то для параметра pszDriverType должно быть задано значение szDBASE. Обратите внимание на одну особенность: необходимо использовать именно этот параметр, поскольку функция поддерживается только таблицами dBASE (не мы устанавливаем правила игры, мы только их придерживаемся).

Параметр bRegenIdxs указывает, нужно ли перестроить все устаревшие индексы, связанные с данной таблицей.

Ниже приведен код метода Pack() класса TdBaseTable.

```
procedure TdBaseTable.Pack(RegenIndexes: Boolean);
{ Упаковка таблицы для физического удаления помеченных записей из файла. }
const
  SPackError = 'Table must be active and opened exclusively';
begin
  { Таблица должна быть активна и открыта в эксклюзивном режиме. }
  if not (Active and Exclusive) then
    raise EDatabaseError.Create(SPackError);
  try
    { Упаковка таблицы. }
    Check(dbiPackTable(DBHandle, Handle, Nil, Nil, RegenIndexes));
  finally
    { Обновление Delphi из BDE }
    CursorPosChanged;
    Refresh;
  end;
end;
```

Полный текст объявления и реализации объекта TdBaseTable приведен ниже в этой главе, в листинге 30.1.

Таблицы Paradox

Таблицы Paradox не обладают многими замечательными возможностями, такими, например, как мягкое удаление, однако поддерживают порядковые номера записей и упаковку таблиц. В этом разделе вы узнаете, как расширить компонент TTable с целью выполнения специфических для таблиц Paradox задач и как создать соответствующий производный класс TParadoxTable.

Порядковый номер

В таблицах Paradox не используется физический номер записи, как в dBASE. Однако эти таблицы поддерживают использование порядкового номера каждой записи в таблице. Порядковый номер отличается от физического номера записи тем, что он зависит от индекса, который в настоящее время применен к таблице. Порядковый номер записи определяет последовательность записей при их отображении на основе текущего индекса.

BDE упрощает получение порядкового номера. Для этого используется функция dbiGetSeqNo(), которая определяется следующим образом:

```
function dbiGetSeqNo(hCursor: hDBICur; var iSeqNo: Longint): DBIResult
  stdcall;
```

Параметр hCursor является дескриптором курсора для таблицы Paradox. После вызова функции в параметре iSeqNo будет содержаться порядковый номер текущей записи. В следующем фрагменте кода показан текст функции GetRecNum() объекта TParadoxTable:

```
function TParadoxTable.GetRecNum: Longint;
{ Возвращает порядковый номер текущей записи. }
begin
  UpdateCursorPos;        // Обновление BDE из Delphi
  { Помещение порядкового номера записи в Result. }
  Check(dbiGetSeqNo(Handle, Result));
end;
```

Упаковка таблицы

Упаковка таблицы Paradox отличается по смыслу от аналогичной операции в dBASE, так как в Paradox не поддерживается мягкое удаление записей. При удалении записи из таблицы Paradox она сразу же удаляется физически, но при этом на месте данной записи в файле таблице остается незаполненная данными "дыра". Для того чтобы избавиться от нее и, тем самым, уменьшить общий размер таблицы, последнюю необходимо упаковать.

К сожалению, в отличие от таблиц dBASE, в BDE нет функции, которую можно было бы использовать для упаковки таблиц Paradox. Вместо этого следует реструктуризировать таблицу с помощью метода dbiDoRestructure(), а затем указать, что таблица должна быть упакована. Метод dbiDoRestructure() определяется следующим образом:

```
function dbiDoRestructure(hDb: hDBIDb; iTblDescCount: Word;
  pTblDesc: pCRTblDesc; pszSaveAs, pszKeyviolName,
  pszProblemsName: PChar; bAnalyzeOnly: Bool): DBIResult stdcall;
```

Параметр hDb — это дескриптор базы данных. Однако, поскольку эта функция не будет работать при открытой в Delphi таблице, то воспользоваться свойством DBHandle компонента TDataSet не удастся. Для того чтобы обойти это ограничение, необходимо создать временную базу данных. Пример того, как это сделать, приведен несколько ниже.

С помощью параметра iTblDescCount задается число дескрипторов таблицы. Для этого параметра должно быть задано значение 1, поскольку текущая версия BDE на каждый вызов поддерживает лишь один дескриптор таблицы.

Параметр pTblDesc представляет собой указатель на запись типа CRTblDesc, которая идентифицирует таблицу и определяет, как таблица должна быть реструктуризована. Эта запись определяется следующим образом:

```
type
  pCRTblDesc = ^CRTblDesc;
  CRTblDesc = packed record     // Создание/реструктуризация таблицы
    szTblName    : DBITBLNAME;  // Имя таблицы. (Включает необязательный
                                // путь и расширение)
    szTblType    : DBINAME;     // Тип драйвера (необязательный)
    szErrTblName : DBIPATH;     // Имя таблицы ошибок (необязательный)
    szUserName   : DBINAME;     // Имя пользователя (если есть)
    szPassword   : DBINAME;     // Пароль (необязательный)
    bProtected   : WordBool;    // Основной пароль, содержащийся в szPassword
    bPack    : WordBool;        // Упаковка таблицы (только реструктуризация)
```

```
    iFldCount    : Word;        // Число используемых идентификаторов полей
    pecrFldOp    : pCROpType;   // Массив параметров полей
    pfldDesc     : pFLDDesc;    // Массив дескрипторов полей
    iIdxCount    : Word;        // Количество идентификаторов индексов
    pecrIdxOp    : pCROpType;   // Массив параметров индексов
    pidxDesc     : PIDXDesc;    // Массив дескрипторов индексов
    iSecRecCount : Word;        // Число идентификаторов безопасности
    pecrSecOp    : pCROpType;   // Массив параметров безопасности
    psecDesc     : pSECDesc;    // Массив дескрипторов безопасности
    iValChkCount : Word;        // Число проверок значений
    pecrValChkOp : pCROpType;   // Массив параметров проверок значений
    pvchkDesc    : pVCHKDesc;   // Массив дескрипторов проверок значений
    iRintCount   : Word;        // Число ссылок
    pecrRintOp   : pCROpType;   // Массив параметров ссылок
    printDesc    : pRINTDesc;   // Массив дескрипторов ссылок
    iOptParams   : Word;        // Количество необязательных параметров
    pfldOptParams: pFLDDesc;    // Массив дескрипторов полей
    pOptData     : Pointer;     // Необязательные параметры
end;
```

Для упаковки таблицы Paradox необходимо определить только значения полей szTblName, szTblType и bPack.

Параметр pszSaveAs — это необязательный строковый указатель, задающий выходную таблицу, если она отличается от исходной таблицы.

Параметр pszKeyviolName является необязательным строковым указателем, который идентифицирует таблицу для записей, вызывающих нарушения ключа при реструктуризации.

Параметр pszProblemsName — это необязательный строковый указатель, который идентифицирует таблицу для записей, вызывающих проблемы при реструктуризации.

Параметр bAnalyzeOnly не используется.

В следующем фрагменте приведен текст метода Pack() объекта TParadoxTable. Из этого фрагмента видно, как инициализировать запись CRTblDesc, а также как с помощью функции dbiOpenDatabase() создать временную базу данных. Обратите внимание на блок finally, который обеспечивает удаление временной базы данных после ее использования.

```
procedure TParadoxTable.Pack;
var
  TblDesc: CRTblDesc;
  TempDBHandle: HDBIDb;
  WasActive: Boolean;
begin
  { Инициализация записи TblDesc }
  FillChar(TblDesc, SizeOf(TblDesc), 0); // Заполнение нулями
  with TblDesc do begin
    StrPCopy(szTblName, TableName);      // Установка имени таблицы
    StrCopy(szTblType, szPARADOX);       // Установка типа таблицы
    bPack := True;                       // Установка флага упаковки
  end;
  { Сохранение активного состояния таблицы.
    Перед упаковкой таблица должна быть закрыта. }
  WasActive := Active;
```

```
  if WasActive then Close;
  try
    { Создание временной базы данных. }
    Check(dbiOpenDatabase(PChar(DatabaseName), Nil, dbiREADWRITE,
      dbiOpenExcl, Nil, 0, Nil, Nil, TempDBHandle));
    try
      { Упаковка таблицы }
      Check(dbiDoRestructure(TempDBHandle, 1, @TblDesc, Nil, Nil, Nil, False));
    finally
      { Закрытие временной базы данных }
      dbiCloseDatabase(TempDBHandle);
    end;
  finally
    { Восстановление активного состояния таблицы. }
    Active := WasActive;
  end;
end;
```

В листинге 30.1 приведен код модуля `DDGTbls`, в котором определены объекты `TdBaseTable` и `TParadoxTable`.

Листинг 30.1. Модуль `DDGTbls.pas`

```
unit DDGTbls;

interface

uses DB, DBTables, BDE;

type
  TdBaseTable = class(TTable)
  private
    FViewDeleted: Boolean;
    function GetIsDeleted: Boolean;
    function GetRecNum: Longint;
    procedure SetViewDeleted(Value: Boolean);
  protected
    function CreateHandle: HDBICur; override;
  public
    procedure Pack(RegenIndexes: Boolean);
    procedure UndeleteRecord;
    property IsDeleted: Boolean read GetIsDeleted;
    property RecNum: Longint read GetRecNum;
    property ViewDeleted: Boolean read FViewDeleted write SetViewDeleted;
  end;

  TParadoxTable = class(TTable)
  private
  protected
    function CreateHandle: HDBICur; override;
```

```
    function GetRecNum: Longint;
  public
    procedure Pack;
    property RecNum: Longint read GetRecNum;
  end;

implementation

uses SysUtils;

{ TdBaseTable }

function TdBaseTable.GetIsDeleted: Boolean;
{ Возвращает значение, показывающее, удалена ли текущая запись. }
var
  RP: RECProps;
begin
  if not FViewDeleted then  // Не стоит беспокоиться, если удаленные
    Result := False         // записи не отображаются
  else begin
    UpdateCursorPos;        // Обновление BDE из Delphi
    { Получение свойств текущей записи }
    Check(dbiGetRecord(Handle, dbiNOLOCK, Nil, @RP));
    Result := RP.bDeleteFlag;  // Возврат флага из свойств
  end;
end;

function TdBaseTable.GetRecNum: Longint;
{ Возврат физического номера текущей записи. }
var
  RP: RECProps;
begin
  UpdateCursorPos;          // Обновление состояния BDE из Delphi
  { Получение свойств текущей записи }
  Check(dbiGetRecord(Handle, dbiNOLOCK, Nil, @RP));
  Result := RP.iPhyRecNum;  // Возврат значения из свойств
end;

function TdBaseTable.CreateHandle: HDBICur;
{ Переопределение предка для выполнения проверки того,
  что это таблица dBASE. }
var
  CP: CURProps;
begin
  Result := inherited CreateHandle;  // Наследование
  if Result <> Nil then begin
    { Получение свойств курсора и генерация исключительной
      ситуации, если таблица не использует драйвер dBASE. }
    Check(dbiGetCursorProps(Result, CP));
    if not (CP.szTableType = szdBASE) then
```

```
      raise EDatabaseError.Create('Not a dBASE table');
  end;
end;

procedure TdBaseTable.Pack(RegenIndexes: Boolean);
{ Упаковка таблицы для удаления записей из файла. }
const
  SPackError = 'Table must be active and opened exclusively';
begin
  { Таблица должна быть активна и открыта эксклюзивно }
  if not (Active and Exclusive) then
    raise EDatabaseError.Create(SPackError);
  try
    { Упаковка таблицы }
    Check(dbiPackTable(DBHandle, Handle, Nil, Nil, RegenIndexes));
  finally
    { Обновление Delphi из BDE. }
    CursorPosChanged;
    Refresh;
  end;
end;

procedure TdBaseTable.SetViewDeleted(Value: Boolean);
{ Переключение между режимами отображения/сокрытия удаленных записей. }
begin
  { Таблица должна быть активна }
  if Active and (FViewDeleted <> Value) then begin
    DisableControls;       // Устранение мерцания
    try
      { Вызов функции BDE для переключения режима
        просмотра мягко удаленных записей. }
      Check(dbiSetProp(hdbiObj(Handle), curSOFTDELETEON, Longint(Value)));
    finally
      Refresh;           // Обновление Delphi
      EnableControls;    // Устранять мерцание больше не требуется
    end;
    FViewDeleted := Value
  end;
end;

procedure TdBaseTable.UndeleteRecord;
begin
  if not IsDeleted then
    raise EDatabaseError.Create('Record is not deleted');
  Check(dbiUndeleteRecord(Handle));
  Refresh;
end;

function TParadoxTable.CreateHandle: HDBICur;
{ Переопределение предка для выполнения проверки того,
```

```
  что это - таблица Paradox. }
var
  CP: CURProps;
begin
  Result := inherited CreateHandle;  // Наследование
  if Result <> Nil then begin
    { Получение свойств курсора и генерация исключительной
      ситуации, если таблица не использует драйвер Paradox. }
    Check(dbiGetCursorProps(Result, CP));
    if not (CP.szTableType = szPARADOX) then
      raise EDatabaseError.Create('Not a Paradox table');
  end;
end;

function TParadoxTable.GetRecNum: Longint;
{ Возврат порядкового номера текущей записи. }
begin
  UpdateCursorPos;      // Обновление BDE из Delphi
  { Помещение порядкового номера текущей записи в Result. }
  Check(dbiGetSeqNo(Handle, Result));
end;

procedure TParadoxTable.Pack;
var
  TblDesc: CRTblDesc;
  TempDBHandle: HDBIDb;
  WasActive: Boolean;
begin
  { Инициализация записи TblDesc }
  FillChar(TblDesc, SizeOf(TblDesc), 0); // Заполнение нулями
  with TblDesc do begin
    StrPCopy(szTblName, TableName);   // Установка имени таблицы
    szTblType := szPARADOX;           // Установка типа таблицы
    bPack := True;                    // Установка флага упаковки
  end;
  { Сохранение активного состояния таблицы.
    Перед упаковкой таблица должна быть закрыта. }
  WasActive := Active;
  if WasActive then Close;
  try
    { Создание временной базы данных }
    Check(dbiOpenDatabase(PChar(DatabaseName), Nil, dbiREADWRITE,
      dbiOpenExcl, Nil, 0, Nil, Nil, TempDBHandle));
    try
      { Упаковка таблицы }
      Check(dbiDoRestructure(TempDBHandle, 1, @TblDesc, Nil, Nil, Nil, False));
    finally
      { Закрытие временной базы данных }
      dbiCloseDatabase(TempDBHandle);
    end;
```

```
  finally
    { Восстановление активного состояния таблицы. }
    Active := WasActive;
  end;
end;

end.
```

Ограничение результирующих наборов данных компонента TQuery

Рассмотрим классическую ошибку при программировании на языке SQL. Приложение отправляет оператор SQL на сервер, который возвращает результирующий набор, состоящий из огромного количества строк. При этом пользователь приложения вынужден тратить свое время на ожидание завершения запроса и поступление всего возвращаемого набора данных, занимая сервер и перегружая сеть. Здравый смысл подсказывает, что не нужно создавать такие запросы, которые приводят к выборке огромного количества записей. Однако иногда подобная ситуация все же возникает, и компонент TQuery тут не поможет, поскольку в нем отсутствуют средства ограничения числа записей в результирующем наборе, получаемом с сервера. К счастью, BDE обеспечивает такую возможность и ее очень просто применить к потомку компонента TQuery.

В интерфейсе BDE API для этой цели предназначена функция dbiSetProp(), которая уже рассматривалась выше в этой главе. При этом в качестве первого параметра функции dbiSetProp() должен быть задан дескриптор курсора для запроса, в качестве второго — значение curMAXROWS, а в качестве последнего параметра — максимальное количество строк в результирующем наборе.

Идеальное место для вызова этой функции — метод PrepareCursor() компонента TQuery, который вызывается сразу же после открытия запроса. В листинге 30.2 приведен код модуля ResQuery, в котором определен компонент TRestrictedQuery.

Листинг 30.2. Модуль ResQuery.pas

```
unit ResQuery;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, DB, DBTables, BDE;

type
  TRestrictedQuery = class(TQuery)
  private
    FMaxRowCount: Longint;
  protected
    procedure PrepareCursor; override;
  published
    property MaxRowCount: Longint read FMaxRowCount write FMaxRowCount;
```

```
  end;

procedure Register;

implementation

procedure TRestrictedQuery.PrepareCursor;
begin
  inherited PrepareCursor;
  if FMaxRowCount > 0 then
    Check(DbiSetProp(hDBIObj(Handle), curMAXROWS, FMaxRowCount));
end;

procedure Register;
begin
  RegisterComponents('DDG', [TRestrictedQuery]);
end;

end.
```

Результирующий набор запроса можно ограничить, задав свойству MaxRowCount требуемое положительное значение. На рис. 30.1 показан результат запроса, ограниченный тремя строками (как видно в диалоговом окне SQL Monitor).

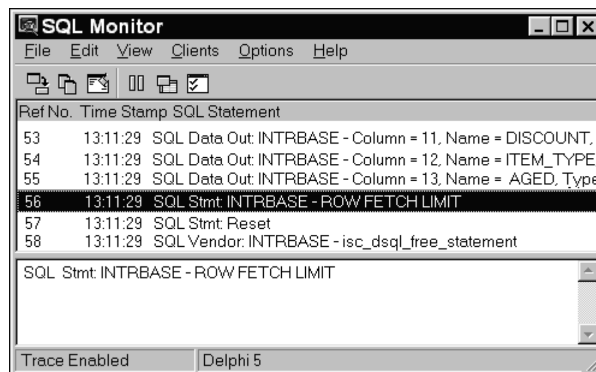


Рис. 30.1. Ограниченный запрос в окне SQL Monitor

Прочие полезные функции BDE

При разработке приложений баз данных определились несколько общих задач разработки, выполнение которых можно частично автоматизировать. К их числу относятся: выполнение в таблице обобщающих функций SQL, копирование таблиц и получение списка пользователей Paradox для определенного сеанса.

Обобщающие функции SQL

Строго говоря, обобщающие функции SQL представляют собой встроенные функции языка SQL, выполняющие некоторую арифметическую операцию для одного или нескольких полей одной или нескольких строк. Наиболее общими примерами таких функций являются:

sum(), которая складывает значения полей нескольких строк; avg(), вычисляющая среднее значение полей нескольких строк; min(), которая находит минимальное значение полей нескольких строк; max(), определяющая максимальное значение полей нескольких строк.

Иногда описанные выше обобщающие функции в Delphi применять не очень неудобно. Например, при доступе к данным с помощью компонента TTable использование этих функций требует создания компонента TQuery, генерации для таблицы и рассматриваемого столбца корректного оператора SQL, формирования запроса и получения его результата. Естественно, этот процесс можно автоматизировать (листинг 30.3).

Листинг 30.3. Автоматизация использования групповых функций SQL

```
type
  TSQLAggFunc = (safSum, safAvg, safMin, safMax);

const
  // Групповые функции SQL
  SQLAggStrs: array[TSQLAggFunc] of string = (
    'select sum(%s) from %s',
    'select avg(%s) from %s',
    'select min(%s) from %s',
    'select max(%s) from %s');

function CreateQueryFromTable(T: TTable): TQuery;
// Возврат запроса к той же базе данных и сеансу, что и таблица T
begin
  Result := TQuery.Create(nil);
  try
    Result.DatabaseName := T.DatabaseName;
    Result.SessionName := T.SessionName;
  except
    Result.Free;
    Raise;
  end;
end;

function DoSQLAggFunc(T: TTable; FieldNames: string;
  Func: TSQLAggFunc): Extended;
begin
  with CreateQueryFromTable(T) do
  begin
    try
      SQL.Add(Format(SQLAggStrs[Func], [FieldNames, T.TableName]));
      Open;
      Result := Fields[0].AsFloat;
    finally
      Free;
    end;
  end;
end;
```

```
function SumField(T: TTable; Field: String): Extended;
begin
  Result := DoSQLAggFunc(T, Field, safSum);
end;

function AvgField(T: TTable; Field: String): Extended;
begin
  Result := DoSQLAggFunc(T, Field, safAvg);
end;

function MinField(T: TTable; Field: String): Extended;
begin
  Result := DoSQLAggFunc(T, Field, safMin);
end;

function MaxField(T: TTable; Field: string): Extended;
begin
  Result := DoSQLAggFunc(T, Field, safMax);
end;
```

Как видно из приведенного выше листинга, каждая отдельная групповая функция содержит вызов функции DoSQLAggFun(), в которой функция CreateQueryFromTable() создает и возвращает компонент TQuery, использующий ту же базу данных и сеанс, что и компонент TTable, переданный в качестве параметра T. Затем из массива строк формируется соответствующая строка SQL, выполняется запрос и, наконец, функция возвращает его результат.

Быстрое копирование таблицы

Копирование таблицы можно выполнить несколькими способами. Для того чтобы файл (или файлы) таблицы физически скопировать из одного местоположения в другое, можно использовать функцию Win32 API CopyFile(). Для копирования одного компонента TTable в другой можно также воспользоваться компонентом TBatchMove. Кроме того, для копирования можно использовать метод BatchMove() компонента TTable.

Однако при применении каждого из этих традиционных методов возникают проблемы. Непосредственное копирование файла с помощью API-функции CopyFile() невозможно, если файлы таблицы открыты другим процессом или пользователем, а также если таблица содержится в файле базы данных на SQL-сервере. Копирование файла таблицы становится очень сложной задачей, если принять во внимание тот факт, что может потребоваться скопировать также связанные с таблицей индексы, поля BLOB или файлы значений. Эти проблемы можно решить с помощью компонента TBatchMove, однако его использование является достаточно сложным делом. Кроме того, есть и еще одна проблема: процесс копирования в пакетном режиме выполняется намного медленнее, чем непосредственное копирование файла. При использовании метода TTable.BatchMove() копирование таблицы упрощается, однако при этом не удается преодолеть ограничения производительности, присущие процессу копирования в пакетном режиме.

К счастью, описанная проблема была выявлена разработчиками BDE. В результате в программный интерфейс BDE API была добавлена функция DbiCopyTable(), обеспечивающая более высокое быстродействие и достаточно простая в использовании. Эта функция объявляется следующим образом:

```
function DbiCopyTable (       { Копирование одной таблицы в другую }
    hDb               : hDBIDb; { Дескриптор базы данных }
    bOverWrite        : Bool;   { True при записи поверх существующего файла }
    pszSrcTableName   : PChar;  { Имя исходной таблицы }
    pszSrcDriverType  : PChar;  { Тип исходного драйвера }
    pszDestTableName  : PChar   { Имя результирующей таблицы }
  ): DBIResult stdcall;
```

Функция BDE API не может взаимодействовать непосредственно с компонентами `TTable` библиотеки VCL. Ниже приведен код процедуры, использующий функцию `DbiCopyTable()`, в которую передается компонент `TTable` и имя исходной таблицы.

```
procedure QuickCopyTable(T: TTable; DestTblName: string; Overwrite: Boolean);
{ Копирование таблицы T в идентичную таблицу с именем DestTblName.
  Запись поверх существующей таблицы с именем DestTblName производится
  в том случае, если параметр Overwrite принимает значение True. }
var
  DBType: DBINAME;
  WasOpen: Boolean;
  NumCopied: Word;
begin
  WasOpen := T.Active;  // Сохранение активного состояния таблицы
  if not WasOpen then T.Open;  // Проверка открытия таблицы
  // Получение строки типа драйвера
  Check(DbiGetProp(hDBIObj(T.Handle), drvDRIVERTYPE, @DBType,
    SizeOf(DBINAME), NumCopied));
  // Копирование таблицы
  Check(DbiCopyTable(T.DBHandle, Overwrite, PChar(T.TableName), DBType,
    PChar(DestTblName)));
  T.Active := WasOpen;  // Восстановление активного состояния
end;
```

На заметку

В случае локальных баз данных (Paradox, dBASE, Access и FoxPro) в результирующую таблицу копируются также все связанные с таблицей файлы — например, индексные и BLOB-файлы. Однако в случае таблиц, содержащихся в базе данных SQL, копируется только таблица, и разработчик должен сам обеспечить копирование в результирующую таблицу необходимых индексов и других элементов.

Пользователи сеанса Paradox

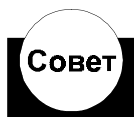
Если в приложении используются таблицы Paradox, то бывает необходимо определить, какой пользователь в настоящее время работает с определенной таблицей Paradox. Это можно выполнить с помощью функции BDE API `DbiOpenUserList()`, которая предоставляет BDE-курсор для списка пользователей текущего сеанса. В следующем фрагменте кода показано, как следует использовать эту функцию:

```
procedure GetPDoxUsersForSession(Sess: TSession; UserList: TStrings);
// Очистка UserList и добавление в список каждого пользователя с
// применением того же сетевого файла, что и для сеанса Sess.
// Если Sess = nil, то процедура работает со стандартным сетевым файлом.
var
```

```
  WasActive: Boolean;
  SessHand: hDBISes;
  ListCur: hDBICur;
  User: UserDesc;
begin
  if UserList = nil then Exit;
  UserList.Clear;
  if Assigned(Sess) then
  begin
    WasActive := Sess.Active;
    if not WasActive then Sess.Open;
    Check(DbiStartSession(nil, SessHand, PChar(Sess.NetFileDir)));
  end
  else
    Check(DbiStartSession(nil, SessHand, nil));
  try
    Check(DbiOpenUserList(ListCur));
    try
      while DbiGetNextRecord(ListCur, dbiNOLOCK, @User, nil) = DBIERR_NONE
        do UserList.Add(User.szUserName);
      finally
        DbiCloseCursor(ListCur);
          // Закрытие курсора "список пользователей таблицы"
    end;
  finally
    DbiCloseSession(SessHand);
    if Assigned(Sess) then Sess.Active := WasActive;
  end;
end;
```

Обратите внимание на интересную особенность функции DbiOpenUserList(): она создает курсор, который управляется так же, как и любой другой BDE-курсор. В данном случае функция DbiGetNextRecord() вызывается несколько раз, пока не будет достигнут конец таблицы. В буфере записей этой таблицы используется формат записи UserDesc, который в модуле BDE определен следующим образом:

```
type
  pUSERDesc = ^USERDesc;
  USERDesc = packed record        { Описание пользователя }
    szUserName     : DBIUSERNAME; { Имя пользователя }
    iNetSession    : Word;        { Сетевой уровень сеанса }
    iProductClass  : Word;        { Класс продукта пользователя }
    szSerialNum    : packed array [0..21] of Char;  { Серийный номер }
  end;
```



Обратите внимание на то, что в процедуре GetPDoxUsersForSession() используется блок защиты ресурсов try..finally. Тем самым гарантируется, что оба связанных с сеансом BDE-ресурса и курсор в любом случае будут выгружены из памяти.

Создание элементов управления VCL для работы с данными

В главах 21, “Создание пользовательских компонентов в Delphi”, и 22, “Сложные методики работы с компонентами”, содержатся подробные сведения о технике и методологии создания компонентов. Однако при этом остались открытыми вопросы создания управляющих элементов для работы с данными. В сущности, создание обычных управляющих элементов VCL мало чем отличается от разработки компонентов для работы с данными, и, тем не менее, есть несколько важных отличий.

- Компоненты работы с данными поддерживают внутренний объект связи с источником данных. Он является производным от класса `TDataLink` и обеспечивает средства, с помощью которых данный элемент управления будет связываться с компонентом `TDataSource`. Для элементов управления работой с данными, которые соединяются с одним полем набора данных, обычно используется объект класса `TFieldDataLink`. Элемент управления должен обрабатывать событие `OnDataChange` объекта связи с источником данных, что позволит получать уведомления об изменении данных поля или записи.
- Компоненты работы с данными должны обрабатывать сообщение `CM_GETDATALINK`. Типичная реакция на это сообщение — возврат указателя на объект связи с источником данных в поле `Result` сообщения.
- Компоненты работы с данными должны поддерживать свойство типа `TDataSource`, что позволит им подключаться к источнику данных. Согласно общему соглашению, это свойство должно называться `DataSource`. Элементы управления, которые соединяются с одним полем данных, должны также поддерживать строковое свойство, содержащее имя подключенного поля. Согласно общему соглашению, это свойство должно называться `DataField`.
- Компоненты работы с данными должны переопределять метод `Notification()`, определенный в классе `TComponent`. В результате этого переопределения элемент управления сможет получить сообщение в том случае, если связанный с ним компонент источника данных будет удален из формы.

В качестве примера создания элемента управления для работы с данными рассмотрим код модуля `DBSound`, приведенный в листинге 30.4. В этом модуле содержится компонент `TDBWavPlayer`, который воспроизводит `.wav`-файлы из BLOB-поля, содержащегося в наборе данных.

Листинг 30.4. Модуль `DBSound.pas`

```
unit DBSound;

interface

uses Windows, Messages, Classes, SysUtils, Controls, Buttons, DB,
  DBTables, DbCtrls;

type
  EDBWavError = class(Exception);
```

```
  TDBWavPlayer = class(TSpeedButton)
  private
    FAutoPlay: Boolean;
    FDataLink: TFieldDataLink;
    FDataStream: TMemoryStream;
    FExceptOnError: Boolean;
    procedure DataChange(Sender: TObject);
    function GetDataField: string;
    function GetDataSource: TDataSource;
    function GetField: TField;
    procedure SetDataField(const Value: string);
    procedure SetDataSource(Value: TDataSource);
    procedure CMGetDataLink(var Message: TMessage); message CM_GETDATALINK;
    procedure CreateDataStream;
    procedure PlaySound;
  protected
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Click; override;
    property Field: TField read GetField;
  published
    property AutoPlay: Boolean read FAutoPlay write FAutoPlay
      default False;
    property ExceptOnError: Boolean read FExceptOnError
      write FExceptOnError;
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;

implementation

uses MMSystem;

const
  // Строки ошибок
  SNotBlobField = 'Field "%s" is not a blob field';
  SPlaySoundErr = 'Error attempting to play sound';

constructor TDBWavPlayer.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);            // Вызов потомка
  FDataLink := TFieldDataLink.Create;   // Создание поля связи данных
  FDataLink.OnDataChange := DataChange; // Получение уведомления о данных
  FDataStream := TMemoryStream.Create;  // Создание рабочего потока памяти
end;

destructor TDBWavPlayer.Destroy;
```

```
begin
  FDataStream.Free;
  FDataLink.Free;
  FDataLink := Nil;
  inherited Destroy;
end;

procedure TDBWavPlayer.Click;
begin
  inherited Click;  // Установить состояние по умолчанию
  PlaySound;        // Воспроизвести звук
end;

procedure TDBWavPlayer.CreateDataStream;
// Создание потока памяти из звукового файла в blob-поле
var
  BS: TBlobStream;
begin
  // Проверить, что это blob-поле
  if not (Field is TBlobField) then
    raise EDBWavError.CreateFmt(SNotBlobField, [DataField]);
  // Создание blob-потока
  BS := TBlobStream.Create(TBlobField(Field), bmRead);
  try
    // Копирование из blob-потока в поток памяти
    FDataStream.SetSize(BS.Size);
    FDataStream.CopyFrom(BS, BS.Size);
  finally
    BS.Free;    // Освобождение blob-потока
  end;
end;

procedure TDBWavPlayer.PlaySound;
// Воспроизведение звука, загруженного в поток памяти
begin
  // Проверить связь набора данных и поля
  if (DataSource <> nil) and (DataField <> '') then
  begin
    // Проверить, создан ли поток данных
    if FDataStream.Size = 0 then CreateDataStream;
    // Воспроизведение звука в потоке памяти,
    // генерирование исключительной ситуации в случае ошибки
    if (not MMSystem.PlaySound(FDataStream.Memory, 0, SND_ASYNC
      or SND_MEMORY)) and FExceptOnError then
      raise EDBWavError.Create(SPlaySoundErr);
  end;
end;

procedure TDBWavPlayer.DataChange(Sender: TObject);
// Дескриптор OnChange метода FFieldDataLink.DataChange
```

```
begin
  // Очистка памяти от предыдущего звукового файла
  with FDataStream do if Size <> 0 then SetSize(0);
  // Если используется AutoPlay, воспроизвести звук
  if FAutoPlay then PlaySound;
end;

procedure TDBWavPlayer.Notification(AComponent: TComponent; Operation:
TOperation);
begin
  inherited Notification(AComponent, Operation);
  // Выполнить необходимые приготовления
  if (Operation = opRemove) and (FDataLink <> nil) and
    (AComponent = FDataSource) then DataSource := nil;
end;

function TDBWavPlayer.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TDBWavPlayer.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
  if Value <> nil then Value.FreeNotification(Self);
end;

function TDBWavPlayer.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

procedure TDBWavPlayer.SetDataField(const Value: string);
begin
  FDataLink.FieldName := Value;
end;

function TDBWavPlayer.GetField: TField;
begin
  Result := FDataLink.Field;
end;

procedure TDBWavPlayer.CMGetDataLink(var Message: TMessage);
begin
  Message.Result := Integer(FDataLink);
end;

end.
```

Описываемый компонент является потомком компонента `TSpeedButton`. По щелчку на нем воспроизводится `.wav`-файл, находящийся в BLOB-поле базы данных. В свойстве `AutoPlay` может содержаться значение `True`, что приведет к автоматическому воспроизведению звука при перемещении пользователя к новой записи в таблице. В этом случае значение свойства `Visible` имеет смысл установить равным `False`, чтобы кнопка не отображалась на форме.

В обработчике `DataChange()` с помощью компонента `TBlobStream` извлекается BLOB-поле, а затем BLOB-поток копируется в поток памяти `FDataStream`. Когда звуковой файл находится в потоке, его можно воспроизводить с помощью функции Win32 API `PlaySound()`.

Расширение возможностей компонента `TDataSet`

Одной из замечательных свойств библиотеки VCL баз данных является абстрактный компонент `TDataSet`, с помощью которого в среде разработки компонентов VCL можно управлять источниками данных, не связанными с механизмом BDE.

Давным-давно...

В предыдущих версиях Delphi поддерживаемая библиотекой VCL архитектура баз данных была закрытой, что весьма усложняло управление с помощью VCL-компонентов любыми источниками данных, отличными от BDE. На рис. 30.2 показана BDE-центрированнная архитектура набора данных в Delphi 1 и Delphi 2.

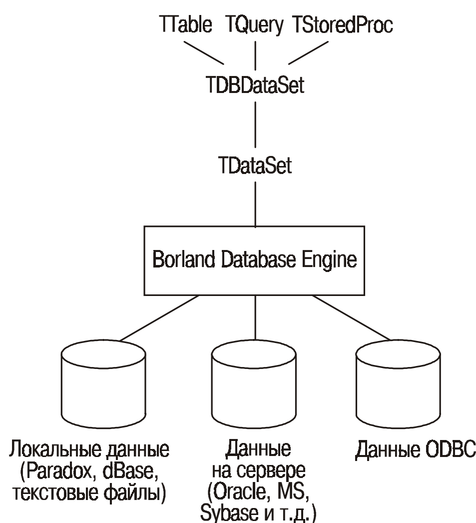


Рис. 30.2. Схема доступа к источникам данных, принятая в Delphi 1 и Delphi 2

Из рис. 30.2 видно, что компонент `TDataSet` жестко привязан к BDE и в этой архитектуре нет места для источников данных, отличных от BDE. Разработчики, желающие использовать такие источники данных в среде VCL, могли в этом случае воспользоваться одним из следующих способов.

- Создать библиотеку DLL, которая с точки зрения VCL выполняет те же функции, что и механизм BDE, однако работает с другим типом данных.
- Отказаться от использования компонента `TDataSet` и для работы с набором данных создать собственный класс и соответствующие элементы управления.

Очевидно, что каждый из этих способов требует слишком больших затрат и не может претендовать на роль изящного решения проблемы.

...и теперь

Зная о существующих проблемах, группа разработчиков Delphi поставила цель — расширить в Delphi 3 архитектуру использования источников данных. В основе новой архитектуры лежит следующая идея: требуется создать абстрактный класс `TDataSet` для представления любого набора данных в библиотеке VCL, а специфический для BDE код переместить в новый класс `TBDEDataSet`. Эта новая архитектура проиллюстрирована на рис. 30.3.

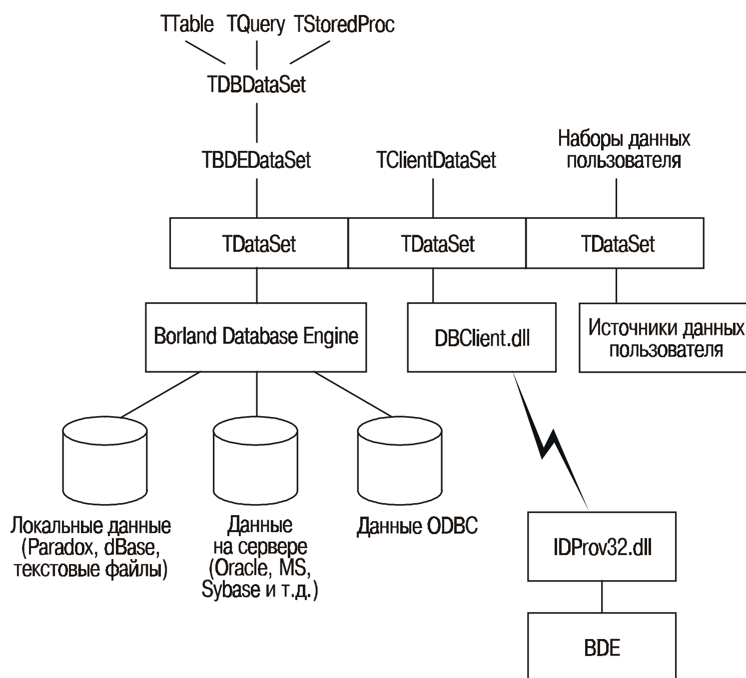
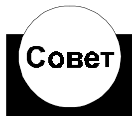


Рис. 30.3. Схема доступа к источникам данных, принятая в Delphi 3 и последующих версиях

Теперь, когда компонент `TDataSet` не связан с механизмом BDE, возникает проблема создания потомка `TDataSet`, работающего с конкретным типом данных, отличных от BDE. Создание полнофункционального потомка компонента `TDataSet` является непростой задачей. Ее решение требует знания используемой в компонентах библиотеки VCL архитектуры доступа к базам данных, а также методов создания компонентов.



В комплект поставки Delphi входят два примера создания потомка TDataSet — очень простой и очень сложный. В простом примере представлен класс TTextDataSet, содержащийся в модуле TextData в каталоге \Delphi 5\Demos\Db\TextData. Он инкапсулирует компонент TStringList как набор данных, состоящий из одного поля. В сложном примере реализован класс TBDEDataSet, расположенный в модуле DbTables в исходном коде VCL. Как уже упоминалось, этот класс соответствует архитектуре базы данных VCL для работы с наборами данных BDE.

Создание потомка компонента TDataSet

По степени сложности большинство реализаций набора данных располагается между компонентами TTextDataSet и TBDEDataSet. Давайте рассмотрим, как создать потомок компонента TDataSet, который обрабатывает данные типизированных файлов Object Pascal (описание типизированных файлов file of record можно найти в главе 12, "Работа с файлами" (том I)). В рассматриваемом примере используется следующая запись и тип файла:

```
type
  // Массив символов произвольной длины, используемый для имени поля
  TNameStr = array[0..31] of char;

  // Запись, представляющая собой "табличную" структуру
  PDDGData = ^TDDGData;
  TDDGData = record
    Name: TNameStr;
    Height: Double;
    ShoeSize: Integer;
  end;

  // Типизированный Файл
  TDDGDataFile = file of TDDGData;
```

Тип Object Pascal file of record обеспечивает удобный и эффективный способ хранения информации, однако в таком формате нельзя вставить или удалить запись в середине файла. Для того чтобы обойти это ограничение, используется двухфайловая схема. Первым является *файл типизированных данных*, а вторым — *индексный файл*, в котором содержится список целых чисел, задающих смещение искомых записей в первом файле. Это означает, что положение записи в файле данных не обязательно совпадает с ее положением в наборе данных. Положение конкретной записи в наборе данных определяется ее смещением в индексном файле: первое целое число в индексном файле задает смещение первой записи в файле данных, второе — смещение второй записи в файле данных и т.д.

В этом разделе будут рассмотрены действия, которые необходимо выполнить для создания потомка класса TDataSet под именем TDDGDataSet, связанного с типизированным файлом.

Абстрактные методы компонента TDataSet

Компонент TDataSet является абстрактным классом, и для его использования необходимо переопределить методы управления набором данных некоторого определенного типа. Потребуется переопределить как минимум каждый из 23 абстрактных методов компонента

TDataSet, а, возможно, и некоторые дополнительные методы. Для удобства эти методы разделены на шесть логических групп: методы буферизации записи, навигационные методы, методы работы с закладками, методы редактирования, прочие методы и дополнительные методы.

В следующем фрагменте кода приведена отредактированная версия компонента TDataSet, определенного в модуле Db.pas. Для простоты изложения показаны лишь логические группы методов.

```
type
  TDataSet = class(TComponent)
  { ... }
  protected
  { Методы буферизации записи }
    function AllocRecordBuffer: PChar; virtual; abstract;
    procedure FreeRecordBuffer(var Buffer: PChar); virtual; abstract;
    procedure InternalInitRecord(Buffer: PChar); virtual; abstract;
    function GetRecord(Buffer: PChar; GetMode: TGetMode;
      DoCheck: Boolean): TGetResult; virtual; abstract;
    function GetRecordSize: Word; virtual; abstract;
    function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;
    procedure SetFieldData(Field: TField; Buffer: Pointer); virtual; abstract;
  { Методы работы с закладками }
    function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; override;
    procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag);
      override;
    procedure GetBookmarkData(Buffer: PChar; Data: Pointer); override;
    procedure SetBookmarkData(Buffer: PChar; Data: Pointer); override;
    procedure InternalGotoBookmark(Bookmark: Pointer); override;
    procedure InternalSetToRecord(Buffer: PChar); override;
  { Навигационные методы }
    procedure InternalFirst; virtual; abstract;
    procedure InternalLast; virtual; abstract;
  { Методы редактирования }
    procedure InternalAddRecord(Buffer: Pointer; Append: Boolean);
      virtual; abstract;
    procedure InternalDelete; virtual; abstract;
    procedure InternalPost; virtual; abstract;
  { Прочие методы }
    procedure InternalClose; virtual; abstract;
    procedure InternalHandleException; virtual; abstract;
    procedure InternalInitFieldDefs; virtual; abstract;
    procedure InternalOpen; virtual; abstract;
    function IsCursorOpen: Boolean; virtual; abstract;
  { Дополнительные методы }
    function GetRecordCount: Integer; virtual;
    function GetRecNo: Integer; virtual;
    procedure SetRecNo(Value: Integer); virtual;
  { ... }
  end;
```

Методы буферизации записи

Необходимо переопределить несколько методов, которые имеют отношение к буферизации записей. Компоненты библиотеки VCL инкапсулируют детали реализации буферизации записи, поэтому в классе TDataSet обеспечено создание группы буферов и управление ими, а разработчику остается лишь принять решение о содержании буферов и способе перемещения данных между различными буферами. Поскольку во всех потомках класса TDataSet должны быть реализованы закладки, то информация о них будет храниться непосредственно в буфере записей, сразу после самой записи данных. Тип, применяемый для описания закладки, имеет следующий вид:

```
type
  // Запись с информацией о закладке, применяемая для
   // реализации закладок в компоненте TDataSet
  PDDGBookmarkInfo = ^TDDGBookmarkInfo;
  TDDGBookmarkInfo = record
    BookmarkData: Integer;
    BookmarkFlag: TBookmarkFlag;
  end;
```

Поле BookmarkData представляет собой искомое значение в файле данных. Поле BookmarkFlag используется для определения корректности содержащейся в буфере закладки. Если набор данных позиционирован в начало или в конец файла, то в этом поле могут содержаться также специальные значения.

На заметку

Не забывайте о том, что приведенная здесь реализация закладок и буферов записей — всего лишь одно из возможных решений. При создании потомка компонента TDataSet для обработки данных какого-либо другого типа, возможно, лучше подойдет другая реализация буферов записей и закладок. Например, некоторый способ управления закладками может поддерживать сам инкапсулируемый источник данных.

Прежде чем приступить к рассмотрению методов буферизации записей, познакомимся с конструктором класса TDDGDataSet:

```
constructor TDDGDataSet.Create(AOwner: TComponent);
begin
  FIndexList := TIndexList.Create;
  FRecordSize := SizeOf(TDDGData);
  FBufferSize := FRecordSize + SizeOf(TDDGBookmarkInfo);
  inherited Create(AOwner);
end;
```

В этом конструкторе выполняются три основные задачи. Во-первых, создается объект TIndexList, который используется в качестве индексного файла, определяющего порядок записей в наборе данных. Во-вторых, инициализируются поля FRecordSize и FBufferSize. В поле FRecordSize содержится размер записи данных, а поле FBufferSize — полный размер буфера записи (общий размер записи данных и записи с информацией о закладке). И, наконец, для заполнения компонента TDataSet значениями, используемыми по умолчанию, вызывается унаследованный конструктор предка.

В потомке компонента TDataSet должны быть переопределены следующие методы работы с буферами записей. Все они, за исключением метода GetFieldData(), объявлены в базовом классе как абстрактные:

```
function AllocRecordBuffer: PChar; override;
procedure FreeRecordBuffer(var Buffer: PChar); override;
procedure InternalInitRecord(Buffer: PChar); override;
function GetRecord(Buffer: PChar; GetMode: TGetMode; DoCheck: Boolean):
  TGetResult; override;
function GetRecordSize: Word; override;
function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;
procedure SetFieldData(Field: TField; Buffer: Pointer); override;
```

Метод AllocRecordBuffer()

Метод AllocRecordBuffer() вызывается для выделения памяти под буфер для одной записи. В рассматриваемой реализации этого метода для предоставления объема памяти, необходимого для хранения данных записи и данных закладки, используется функция AllocMem():

```
function TDDGDataSet.AllocRecordBuffer: PChar;
begin
  Result := AllocMem(FBufferSize);
end;
```

Метод FreeRecordBuffer()

Несложно догадаться, что этот метод должен освобождать память, выделенную методом AllocRecordBuffer(). В данном случае это реализовано с помощью процедуры FreeMem() следующим образом:

```
procedure TDDGDataSet.FreeRecordBuffer(var Buffer: PChar);
begin
  FreeMem(Buffer);
end;
```

Метод InternalInitRecord()

Метод InternalInitRecord() вызывается для инициализации буфера записей. Этот метод позволяет выполнять такие операции, как установка значений поля, используемых по умолчанию, или некоторая инициализация пользовательских данных буфера записи. В рассматриваемой реализации буфер записей просто инициализируется нулевым значением:

```
procedure TDDGDataSet.InternalInitRecord(Buffer: PChar);
begin
  FillChar(Buffer^, FBufferSize, 0);
end;
```

Метод GetRecord()

Основное назначение метода GetRecord() заключается в предоставлении данных из предыдущей, текущей или последующей записи набора данных. Значение, возвращаемое этой функцией, имеет тип TGetResult, который в модуле Db определен следующим образом:

```
TGetResult = (grOK, grBOF, grEOF, grError);
```

Назначение каждого из перечисленных параметров очевидно: grOk означает успешное выполнение, grBOF — набор данных позиционирован в начало, grEOF — позиционирован в конец, grError — произошла ошибка.

Реализация этого метода имеет следующий вид:

```
function TDDGDataSet.GetRecord(Buffer: PChar; GetMode: TGetMode;
  DoCheck: Boolean): TGetResult;
var
  IndexPos: Integer;
begin
  if FIndexList.Count < 1 then
    Result := grEOF
  else begin
    Result := grOk;
    case GetMode of
      gmPrior:
        if FRecordPos <= 0 then
        begin
          Result := grBOF;
          FRecordPos := -1;
        end
        else
          Dec(FRecordPos);
      gmCurrent:
        if (FRecordPos < 0) or (FRecordPos >= RecordCount) then
          Result := grError;
      gmNext:
        if FRecordPos >= RecordCount-1 then
          Result := grEOF
        else
          Inc(FRecordPos);
    end;
    if Result = grOk then
    begin
      IndexPos := Integer(FIndexList[FRecordPos]);
      Seek(FDataFile, IndexPos);
      BlockRead(FDataFile, PDDGData(Buffer)^, 1);
      with PDDGBookmarkInfo(Buffer + FRecordSize)^ do
      begin
        BookmarkData := FRecordPos;
        BookmarkFlag := bfCurrent;
      end;
    end
    else if (Result = grError) and DoCheck then
      DatabaseError('No records');
  end;
end;
```

В поле FRecordPos хранится позиция текущей записи в наборе данных. Обратите внимание на то, что значение этого поля увеличивается или уменьшается каждый раз, когда для получения следующей или предыдущей записи вызывается метод GetRecord(). Если в поле

FRecordPos содержится допустимый номер записи, то он используется в качестве индекса в FIndexList. Этот индекс применяется для поиска требуемой записи в файле данных. Найденная запись считывается из файла данных в буфер записи, определяемый параметром Buffer.

Метод GetRecord() выполняет еще одну дополнительную задачу: если при его вызове параметр DoCheck принимает значение True и grError задает возможное возвращаемое значение, то генерируется исключительная ситуация.

Метод GetRecordSize()

Метод GetRecordSize() возвращает размер в байтах той части записи из буфера, которая соответствует самой записи данных. Убедитесь, что возвращается не размер всей записи буфера, а только размер записи данных. В предлагаемой реализации возвращается значение поля FRecordSize:

```
function TDDGDataSet.GetRecordSize: Word;
begin
  Result := FRecordSize;
end;
```

Метод GetFieldData()

Метод GetFieldData() применяется для копирования данных из буфера активной записи (предоставляемой свойством ActiveBuffer) в буфер для отдельного поля. Чаще всего эту операцию целесообразно выполнять с помощью процедуры Move(). С помощью свойства Index или Name компонента Field можно указать, какое поле будет скопировано. Кроме того, убедитесь, что в свойстве ActiveBuffer задано корректное смещение, поскольку в нем содержатся все поля записи, а в свойстве Buffer — лишь одно поле. В рассматриваемой реализации поля копируются из внутреннего буфера в соответствующий объект TField:

```
function TDDGDataSet.GetFieldData(Field: TField; Buffer: Pointer): Boolean;
begin
  Result := True;
  case Field.Index of
    0:
      begin
        Move(ActiveBuffer^, Buffer^, Field.Size);
        Result := PChar(Buffer)^ <> #0;
      end;
    1: Move(PDDGData(ActiveBuffer)^.Height, Buffer^, Field.DataSize);
    2: Move(PDDGData(ActiveBuffer)^.ShoeSize, Buffer^, Field.DataSize);
  end;
end;
```

Этот метод, как и метод SetFieldData(), может стать гораздо более сложным, если необходимо обеспечить реализацию дополнительных возможностей, например, таких как использование вычисляемых полей и фильтров.

Метод SetFieldData()

Назначение метода SetFieldData() противоположно назначению метода GetFieldData(). Этот метод копирует данные из буфера поля в буфер активной записи. Как видно из приведенного ниже фрагмента кода, реализации этих двух методов очень похожи:

```
procedure TDDGDataSet.SetFieldData(Field: TField; Buffer: Pointer);
begin
  case Field.Index of
    0: Move(Buffer^, ActiveBuffer^, Field.Size);
    1: Move(Buffer^, PDDGData(ActiveBuffer)^.Height, Field.DataSize);
    2: Move(Buffer^, PDDGData(ActiveBuffer)^.ShoeSize, Field.DataSize);
  end;
  DataEvent(deFieldChange, Longint(Field));
end;
```

По окончании копирования данных вызывается метод DataEvent(), сообщающий о том, что поле было изменено и для него следует сгенерировать событие OnChange.

Методы работы с закладками

Как уже отмечалось, потомки компонента TDataSet должны поддерживать работу с закладками. Для этого необходимо переопределить следующие абстрактные методы компонента TDataSet:

```
function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; override;
procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag); override;
procedure GetBookmarkData(Buffer: PChar; Data: Pointer); override;
procedure SetBookmarkData(Buffer: PChar; Data: Pointer); override;
procedure InternalGotoBookmark(Bookmark: Pointer); override;
procedure InternalSetToRecord(Buffer: PChar); override;
```

Что касается компонента TDDGDataSet, то реализация этих методов в основном сводится к операциям с закладками, помещаемыми в конец буфера записи.

Методы GetBookmarkFlag() и SetBookmarkFlag()

Флаги закладки внутренне используются самим компонентом TDataSet для определения того, не является ли некоторая запись первой или последней в наборе данных. Для этого необходимо переопределить методы GetBookmarkFlag() и SetBookmarkFlag(). Ниже приведена реализация этих методов в компоненте TDDGDataSet.

```
function TDDGDataSet.GetBookmarkFlag(Buffer: PChar): TBookmarkFlag;
begin
  Result := PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkFlag;
end;

procedure TDDGDataSet.SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag);
begin
  PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkFlag := Value;
end;
```

Методы GetBookmarkData() и SetBookmarkData()

Методы GetBookmarkData() и SetBookmarkData() обеспечивают средства управления информацией о закладке записи компонента TDataSet без повторного позиционирования текущей записи. Они реализованы аналогично методам, описанным в предыдущем фрагменте кода:

```
procedure TDDGDataSet.GetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  PInteger(Data)^ := PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkData;
end;

procedure TDDGDataSet.SetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkData := PInteger(Data)^;
end;
```

Метод InternalGotoBookmark()

Метод InternalGotoBookmark() вызывается для перемещения указателя текущей записи в соответствии со значением параметра Bookmark. Поскольку значение закладки — это то же самое, что и номер записи для компонента TDDGDataSet, реализация этого метода имеет совсем простой вид:

```
procedure TDDGDataSet.InternalGotoBookmark(Bookmark: Pointer);
begin
  FRecordPos := Integer(Bookmark);
end;
```

Метод InternalSetToRecord()

Метод InternalSetToRecord() подобен методу InternalGotoBookmark() во всем, за исключением того, что в качестве параметра ему передается не значение закладки, а буфер записи. Задачей этого метода является позиционирование набора данных на запись, определенную параметром Buffer. В данной реализации буфера записи содержится информация о закладке, поскольку значение закладки совпадает с позицией записи. Его реализация имеет следующий вид:

```
procedure TDDGDataSet.InternalSetToRecord(Buffer: PChar);
begin
  // Значение закладки соответствует смещению в файле
  FRecordPos := PDDGBookmarkInfo(Buffer + FRecordSize)^.Bookmarkdata;
end;
```

Навигационные методы

Для поддержки перемещения указателя набора данных на первую или последнюю запись, в компоненте TDataSet необходимо переопределить два абстрактных навигационных метода:

```
procedure InternalFirst; override;
procedure InternalLast; override;
```

Реализация этих методов очень проста: метод InternalFirst() присваивает свойству FRecordPos значение -1 (значение BOF), а метод InternalLast() — значение, равное количеству записей. Поскольку индексы начинаются с нуля, количество записей на единицу больше последнего индекса (значение EOF). Например:

```
procedure TDDGDataSet.InternalFirst;
begin
  FRecordPos := -1;
```

```
end;

procedure TDDGDataSet.InternalLast;
begin
  FRecordPos := FIndexList.Count;
end;
```

Методы редактирования

Для того чтобы обеспечить редактирование, добавление, вставку и удаление записей, нужно переопределить следующих три абстрактных метода компонента TDataSet:

```
procedure InternalAddRecord(Buffer: Pointer; Append: Boolean); override;
procedure InternalDelete; override;
procedure InternalPost; override;
```

Метод InternalAddRecord()

Метод InternalAddRecord() вызывается при вставке или добавлении записи в набор данных. Параметр Buffer указывает на буфер записи, которая будет добавлена в набор данных. Параметр Append должен иметь значение True при добавлении записи и False — при вставке записи. В реализации этого метода для компонента TDDGDataSet сначала осуществляется поиск конца файла данных, запись сохраняется в файле, а затем в соответствующую позицию списка индексов добавляется или вставляется значение, применяемое для поиска в файле данных:

```
procedure TDDGDataSet.InternalAddRecord(Buffer: Pointer; Append: Boolean);
var
RecPos: Integer;
begin
  Seek(FDataFile, FileSize(FDataFile));
  BlockWrite(FDataFile, PDDGData(Buffer)^, 1);
  if Append then
  begin
    FIndexList.Add(Pointer(FileSize(FDataFile) - 1));
    InternalLast;
  end
  else begin
    if FRecordPos = -1 then RecPos := 0
    else RecPos := FRecordPos;
    FIndexList.Insert(RecPos, Pointer(FileSize(FDataFile) - 1));
  end;
  FIndexList.SaveToFile(FIdxName);
end;
```

Метод InternalDelete()

Метод InternalDelete() удаляет из набора данных текущую запись. Поскольку этот метод не удаляет запись из середины файла данных, текущая запись удаляется только из индексного списка. В результате этого в наборе данных удаленная запись оказывается "потерянной". Например:

```
procedure TDDGDataSet.InternalDelete;
begin
  FIndexList.Delete(FRecordPos);
  if FRecordPos >= FIndexList.Count then Dec(FRecordPos);
end;
```

На заметку

Такой метод удаления означает, что файл данных не уменьшается в размере даже при удалении всех записей (как в файлах dBASE). Если при реализации коммерческих проектов вы собираетесь использовать именно этот тип набора данных, то следует реализовать также метод упаковки файла, с помощью которого записи будут удаляться из файла данных физически.

Метод InternalPost()

Метод InternalPost() вызывается в методе TDataSet.Post(). В этом методе данные из активного буфера записи следует записать в файл данных. Обратите внимание на то, что реализация этого метода очень похожа на реализацию метода InternalAddRecord():

```
procedure TDDGDataSet.InternalPost;
var
  RecPos, InsPos: Integer;
begin
  if FRecordPos = -1 then
    RecPos := 0
  else begin
    if State = dsEdit then RecPos := Integer(FIndexList[FRecordPos])
    else RecPos := FileSize(FDataFile);
  end;
  Seek(FDataFile, RecPos);
  BlockWrite(FDataFile, PDDGData(ActiveBuffer)^, 1);
  if State <> dsEdit then
  begin
    if FRecordPos = -1 then InsPos := 0
    else InsPos := FRecordPos;
    FIndexList.Insert(InsPos, Pointer(RecPos));
  end;
  FIndexList.SaveToFile(FIdxName);
end;
```

Прочие методы

Для создания работоспособного потомка компонента TDataSet должно быть переопределено несколько других абстрактных методов. Они являются общими вспомогательными методами, и поскольку не могут быть выделены в определенную категорию, названы прочими методами:

```
procedure InternalClose; override;
procedure InternalHandleException; override;
procedure InternalInitFieldDefs; override;
procedure InternalOpen; override;
function IsCursorOpen: Boolean; override;
```

Метод InternalClose()

Метод InternalClose() вызывается в методе TDataSet.Close(). В этом методе должны быть освобождены все связанные с набором данных ресурсы, которые были выделены методом InternalOpen() или распределены при использовании этого набора данных. В рассматриваемой реализации метода InternalClose() закрывается файл данных и гарантируется, что индексный список будет сохранен на диске. Кроме того, параметр FRecordPos устанавливается в начало файла (BOF), а запись файла данных обнуляется:

```
procedure TDDGDataSet.InternalClose;
begin
  if TFileRec(FDataFile).Mode <> 0 then
    CloseFile(FDataFile);
  FIndexList.SaveToFile(FIdxName);
  FIndexList.Clear;
  if DefaultFields then
     DestroyFields;
  FRecordPos := -1;
  FillChar(FDataFile, SizeOf(FDataFile), 0);
end;
```

Метод InternalHandleException()

Метод InternalHandleException() вызывается в том случае, если при чтении или записи в поток компонентом сгенерирована исключительная ситуация. Если такие исключительные ситуации не требуется обрабатывать особым способом, то этот метод можно реализовать следующим образом:

```
procedure TDDGDataSet.InternalHandleException;
begin
  // Стандартная реализация этого метода
  Application.HandleException(Self);
end;
```

Метод InternalInitFieldDefs()

В методе InternalInitFieldDefs() должны быть определены поля, содержащиеся в наборе данных. Эта операция выполняется с помощью создания экземпляров объектов TFieldDef с передачей свойства TDataSet.FieldDefs в качестве параметра Owner. В данном случае создаются три объекта TFieldDef, представляющие три поля в используемом наборе данных:

```
procedure TDDGDataSet.InternalInitFieldDefs;
begin
  // Создание объектов FieldDefs, соответствующих
  // каждому полю записи набора данных
  FieldDefs.Clear;
  TFieldDef.Create(FieldDefs, 'Name', ftString, SizeOf(TNameStr), False, 1);
  TFieldDef.Create(FieldDefs, 'Height', ftFloat, 0, False, 2);
  TFieldDef.Create(FieldDefs, 'ShoeSize', ftInteger, 0, False, 3);
end;
```

Метод InternalOpen()

Метод InternalOpen() вызывается в методе TDataSet.Open(). В этом методе необходимо открыть основной источник данных, инициализировать все необходимые внутренние поля или свойства, создать определения полей и связать поля с данными. В приведенной ниже реализации этого метода открывается файл данных, загружается индексный список из файла, инициализируется поле FRecordPos и свойство BookmarkSize, а затем создаются и связываются поля. Кроме того, метод InternalOpen() позволяет пользователю создать файлы базы данных, если их еще нет на диске:

```
procedure TDDGDataSet.InternalOpen;
var
  HFile: THandle;
begin
  // Проверка существования файлов таблицы и индекса
  FIdxName := ChangeFileExt(FTableName, feDDGIndex);
  if not (FileExists(FTableName) and FileExists(FIdxName)) then
  begin
    if MessageDlg('Table or index file not found. Create new table?',
      mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    begin
      HFile := FileCreate(FTableName);
      if HFile = INVALID_HANDLE_VALUE then
        DatabaseError('Error creating table file');
      FileClose(HFile);
      HFile := FileCreate(FIdxName);
      if HFile = INVALID_HANDLE_VALUE then
        DatabaseError('Error creating index file');
      FileClose(HFile);
    end
    else
      DatabaseError('Could not open table');
  end;
  // Открытие файла данных
  FileMode := fmShareDenyNone or fmOpenReadWrite;
  AssignFile(FDataFile, FTableName);
  Reset(FDataFile);
  try
    FIndexList.LoadFromFile(FIdxName); // Загрузка из файла
                                       // списка индексов TList
    FRecordPos := -1;                  // Установка указателя
                                       // текущей записи в BOF
    BookmarkSize := SizeOf(Integer);   // Определение размера
                                       // закладки для VCL
    InternalInitFieldDefs;             // Инициализация объектов FieldDef
    // Создание компонентов TField, если не созданы постоянные поля
    if DefaultFields then CreateFields;
    BindFields(True);                  // Связывание объектов FieldDef
                                       // с реальными данными
  except
    CloseFile(FDataFile);
```

```
    FillChar(FDataFile, SizeOf(FDataFile), 0);
    raise;
  end;
end;
```

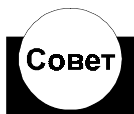
На заметку

Любые ресурсы, распределенные в методе `InternalOpen()`, должны быть освобождены в методе `InternalClose()`.

Метод `IsCursorOpen()`

Метод `IsCursorOpen()` вызывается компонентом `TDataSet` при открытии набора данных, если требуется определить, являются ли данные доступными даже в том случае, когда набор данных не активен. Реализация этого метода для компонента `TDDGDataSet` возвращает значение `True`, если файл данных был открыт:

```
function TDDGDataSet.IsCursorOpen: Boolean;
begin
  // "Курсор" открывается, если файл данных открыт
  Result := TFileRec(FDataFile).Mode <> 0;
end;
```



Предыдущий метод иллюстрирует интересную возможность Object Pascal: типизированный или нетипизированный файл может быть преобразован к типу `TFileRec`, благодаря чему об этом файле можно будет получить информацию низкого уровня. Тип `TFileRec` рассматривался в главе 12, “Работа с файлами” (том I).

Необязательные методы работы с номером записи

Если вы хотите воспользоваться преимуществами компонента `TDBGrid` чтобы иметь средство прокручивания набора данных относительно позиции курсора, необходимо переопределить три метода:

```
function GetRecordCount: Integer; override;
function GetRecNo: Integer; override;
procedure SetRecNo(Value: Integer); override;
```

Хотя в рассматриваемом примере переопределение этих методов не лишено смысла, во многих других случаях оно оказывается не таким полезным, а зачастую даже невозможным. Например, при работе с огромным количеством данных может оказаться невозможным определить количество записей. При взаимодействии с SQL-сервером информация о количестве записей может быть вообще недоступной.

Реализация компонента `TDataSet` чрезвычайно проста, поэтому достаточно проста и реализация этих методов:

```
function TDDGDataSet.GetRecordCount: Integer;
begin
  Result := FIndexList.Count;
end;

function TDDGDataSet.GetRecNo: Integer;
```

```
begin
  UpdateCursorPos;
  if (FRecordPos = -1) and (RecordCount > 0) then
    Result := 1
  else
    Result := FRecordPos + 1;
end;

procedure TDDGDataSet.SetRecNo(Value: Integer);
begin
  if (Value >= 0) and (Value <= FIndexList.Count-1) then
  begin
    FRecordPos := Value - 1;
    Resync([]);
  end;
end;
```

Модуль TDDGDataSet

В листинге 30.5 приведен код модуля DDG_DS, который содержит полную реализацию класса TDDGDataSet.

Листинг 30.5. Модуль DDG_DS.pas

```
unit DDG_DS;

interface

uses Windows, Db, Classes, DDG_Rec;

type
  // Запись с информацией о закладке, применяемая
  // для поддержки работы с закладками в компоненте TDataSet:
  PDDGBookmarkInfo = ^TDDGBookmarkInfo;
  TDDGBookmarkInfo = record
    BookmarkData: Integer;
    BookmarkFlag: TBookmarkFlag;
  end;

  // Список, применяемый для обеспечения доступа к файлу записей:
  TIndexList = class(TList)
  public
    procedure LoadFromFile(const FileName: string); virtual;
    procedure LoadFromStream(Stream: TStream); virtual;
    procedure SaveToFile(const FileName: string); virtual;
    procedure SaveToStream(Stream: TStream); virtual;
  end;

  // Определение потомка класса TDataSet
  TDDGDataSet = class(TDataSet)
  private
```

```
    function GetDataFileSize: Integer;
  public
    FDataFile: TDDGDataFile;
    FIdxName: string;
    FIndexList: TIndexList;
    FTableName: string;
    FRecordPos: Integer;
    FRecordSize: Integer;
    FBufferSize: Integer;
    procedure SetTableName(const Value: string);
  protected
    { Обязательные переопределения }
    // Методы буферизации записи:
    function AllocRecordBuffer: PChar; override;
    procedure FreeRecordBuffer(var Buffer: PChar); override;
    procedure InternalInitRecord(Buffer: PChar); override;
    function GetRecord(Buffer: PChar; GetMode: TGetMode;
      DoCheck: Boolean): TGetResult; override;
    function GetRecordSize: Word; override;
    procedure SetFieldData(Field: TField; Buffer: Pointer); override;
    // Методы работы с закладками:
    procedure GetBookmarkData(Buffer: PChar; Data: Pointer); override;
    function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; override;
    procedure InternalGotoBookmark(Bookmark: Pointer); override;
    procedure InternalSetToRecord(Buffer: PChar); override;
    procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag);
      override;
    procedure SetBookmarkData(Buffer: PChar; Data: Pointer); override;
    // Навигационные методы:
    procedure InternalFirst; override;
    procedure InternalLast; override;
    // Методы редактирования:
    procedure InternalAddRecord(Buffer: Pointer; Append: Boolean);
      override;
    procedure InternalDelete; override;
    procedure InternalPost; override;
    // Прочие методы:
    procedure InternalClose; override;
    procedure InternalHandleException; override;
    procedure InternalInitFieldDefs; override;
    procedure InternalOpen; override;
    function IsCursorOpen: Boolean; override;
    { Необязательные переопределения }
    function GetRecordCount: Integer; override;
    function GetRecNo: Integer; override;
    procedure SetRecNo(Value: Integer); override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function GetFieldData(Field: TField; Buffer: Pointer): Boolean;
```

```
      override;

    // Дополнительные процедуры
    procedure EmptyTable;
  published
    property Active;
    property TableName: string read FTableName write SetTableName;
    property BeforeOpen;
    property AfterOpen;
    property BeforeClose;
    property AfterClose;
    property BeforeInsert;
    property AfterInsert;
    property BeforeEdit;
    property AfterEdit;
    property BeforePost;
    property AfterPost;
    property BeforeCancel;
    property AfterCancel;
    property BeforeDelete;
    property AfterDelete;
    property BeforeScroll;
    property AfterScroll;
    property OnDeleteError;
    property OnEditError;

    // Дополнительные свойства
    property DataFileSize: Integer read GetDataFileSize;
  end;

procedure Register;

implementation

uses BDE, DBTables, SysUtils, DBConsts, Forms, Controls, Dialogs;

const
  feDDGTable = '.ddg';
  feDDGIndex = '.ddx';
  // Обратите внимание: файл не должен быть заблокирован!

{ TIndexList }

procedure TIndexList.LoadFromFile(const FileName: string);
var
  F: TFileStream;
begin
  F := TFileStream.Create(FileName, fmOpenRead or fmShareDenyWrite);
  try
    LoadFromStream(F);
```

```
  finally
    F.Free;
  end;
end;

procedure TIndexList.LoadFromStream(Stream: TStream);
var
  Value: Integer;
begin
  while Stream.Position < Stream.Size do
  begin
    Stream.Read(Value, SizeOf(Value));
    Add(Pointer(Value));
  end;
  ShowMessage(IntToStr(Count));
end;

procedure TIndexList.SaveToFile(const FileName: string);
var
  F: TFileStream;
begin
  F := TFileStream.Create(FileName, fmCreate or fmShareExclusive);
  try
    SaveToStream(F);
  finally
    F.Free;
  end;
end;

procedure TIndexList.SaveToStream(Stream: TStream);
var
  i: Integer;
  Value: Integer;
begin
  for i := 0 to Count - 1 do
  begin
    Value := Integer(Items[i]);
    Stream.Write(Value, SizeOf(Value));
  end;
end;

{ TDDGDataSet }

constructor TDDGDataSet.Create(AOwner: TComponent);
begin
  FIndexList := TIndexList.Create;
  FRecordSize := SizeOf(TDDGData);
  FBufferSize := FRecordSize + SizeOf(TDDGBookmarkInfo);
  inherited Create(AOwner);
end;
```

```
destructor TDDGDataSet.Destroy;
begin
  inherited Destroy;
  FIndexList.Free;
end;

function TDDGDataSet.AllocRecordBuffer: PChar;
begin
  Result := AllocMem(FBufferSize);
end;

procedure TDDGDataSet.FreeRecordBuffer(var Buffer: PChar);
begin
  FreeMem(Buffer);
end;

procedure TDDGDataSet.InternalInitRecord(Buffer: PChar);
begin
  FillChar(Buffer^, FBufferSize, 0);
end;

function TDDGDataSet.GetRecord(Buffer: PChar; GetMode: TGetMode;
  DoCheck: Boolean): TGetResult;
var
  IndexPos: Integer;
begin
  if FIndexList.Count < 1 then
    Result := grEOF
  else begin
    Result := grOk;
    case GetMode of
      gmPrior:
        if FRecordPos <= 0 then
        begin
          Result := grBOF;
          FRecordPos := -1;
        end
        else
          Dec(FRecordPos);
      gmCurrent:
        if (FRecordPos < 0) or (FRecordPos >= RecordCount) then
          Result := grError;
      gmNext:
          if FRecordPos >= RecordCount-1 then
            Result := grEOF
          else
            Inc(FRecordPos);
    end;
    if Result = grOk then
    begin
```

```
      IndexPos := Integer(FIndexList[FRecordPos]);
      Seek(FDataFile, IndexPos);
      BlockRead(FDataFile, PDDGData(Buffer)^, 1);
      with PDDGBookmarkInfo(Buffer + FRecordSize)^ do
      begin
        BookmarkData := FRecordPos;
        BookmarkFlag := bfCurrent;
      end;
    end
    else if (Result = grError) and DoCheck then
      DatabaseError('No records');
  end;
end;

function TDDGDataSet.GetRecordSize: Word;
begin
  Result := FRecordSize;
end;

function TDDGDataSet.GetFieldData(Field: TField; Buffer: Pointer): Boolean;
begin
  Result := True;
  case Field.Index of
    0:
      begin
        Move(ActiveBuffer^, Buffer^, Field.Size);
        Result := PChar(Buffer)^ <> #0;
      end;
    1: Move(PDDGData(ActiveBuffer)^.Height, Buffer^, Field.DataSize);
    2: Move(PDDGData(ActiveBuffer)^.ShoeSize, Buffer^, Field.DataSize);
  end;
end;

procedure TDDGDataSet.SetFieldData(Field: TField; Buffer: Pointer);
begin
  case Field.Index of
    0: Move(Buffer^, ActiveBuffer^, Field.Size);
    1: Move(Buffer^, PDDGData(ActiveBuffer)^.Height, Field.DataSize);
    2: Move(Buffer^, PDDGData(ActiveBuffer)^.ShoeSize, Field.DataSize);
  end;
  DataEvent(deFieldChange, Longint(Field));
end;

procedure TDDGDataSet.GetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  PInteger(Data)^ := PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkData;
end;

function TDDGDataSet.GetBookmarkFlag(Buffer: PChar): TBookmarkFlag;
begin
```

```
  Result := PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkFlag;
end;

procedure TDDGDataSet.InternalGotoBookmark(Bookmark: Pointer);
begin
  FRecordPos := Integer(Bookmark);7
end;

procedure TDDGDataSet.InternalSetToRecord(Buffer: PChar);
begin
  // Значение закладки совпадает со смещением в файле
  FRecordPos := PDDGBookmarkInfo(Buffer + FRecordSize)^.Bookmarkdata;
end;

procedure TDDGDataSet.SetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkData := PInteger(Data)^;
end;

procedure TDDGDataSet.SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag);
begin
  PDDGBookmarkInfo(Buffer + FRecordSize)^.BookmarkFlag := Value;
end;

procedure TDDGDataSet.InternalFirst;
begin
  FRecordPos := -1;
end;

procedure TDDGDataSet.InternalInitFieldDefs;
begin
  // Создание объектов FieldDef, соответствующих каждому
  // полю в записи набора данных
  FieldDefs.Clear;
  TFieldDef.Create(FieldDefs, 'Name', ftString, SizeOf(TNameStr), False, 1);
  TFieldDef.Create(FieldDefs, 'Height', ftFloat, 0, False, 2);
  TFieldDef.Create(FieldDefs, 'ShoeSize', ftInteger, 0, False, 3);
end;

procedure TDDGDataSet.InternalLast;
begin
  FRecordPos := FIndexList.Count;
end;

procedure TDDGDataSet.InternalClose;
begin
  if TFileRec(FDataFile).Mode <> 0 then
    CloseFile(FDataFile);
  FIndexList.SaveToFile(FIdxName);
  FIndexList.Clear;
```

```
  if DefaultFields then
    DestroyFields;
  FRecordPos := -1;
  FillChar(FDataFile, SizeOf(FDataFile), 0);
end;

procedure TDDGDataSet.InternalHandleException;
begin
  // Стандартная реализация этого метода
  Application.HandleException(Self);
end;

procedure TDDGDataSet.InternalDelete;
begin
  FIndexList.Delete(FRecordPos);
  if FRecordPos >= FIndexList.Count then Dec(FRecordPos);
end;

procedure TDDGDataSet.InternalAddRecord(Buffer: Pointer; Append: Boolean);
var
  RecPos: Integer;
begin
  Seek(FDataFile, FileSize(FDataFile));
  BlockWrite(FDataFile, PDDGData(Buffer)^, 1);
  if Append then
  begin
    FIndexList.Add(Pointer(FileSize(FDataFile) - 1));
    InternalLast;
  end
  else begin
    if FRecordPos = -1 then RecPos := 0
    else RecPos := FRecordPos;
    FIndexList.Insert(RecPos, Pointer(FileSize(FDataFile) - 1));
  end;
  FIndexList.SaveToFile(FIdxName);
end;

procedure TDDGDataSet.InternalOpen;
var
  HFile: THandle;
begin
  // Проверка существования файлов таблицы и индекса
  FIdxName := ChangeFileExt(FTableName, feDDGIndex);
  if not (FileExists(FTableName) and FileExists(FIdxName)) then
  begin
    if MessageDlg('Table or index file not found. Create new table?',
      mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    begin
      HFile := FileCreate(FTableName);
      if HFile = INVALID_HANDLE_VALUE then
```

```
        DatabaseError('Error creating table file');
      FileClose(HFile);
      HFile := FileCreate(FIdxName);
      if HFile = INVALID_HANDLE_VALUE then
        DatabaseError('Error creating index file');
      FileClose(HFile);
    end
    else
      DatabaseError('Could not open table');
  end;
  // Открытие файла данных
  FileMode := fmShareDenyNone or fmOpenReadWrite;
  AssignFile(FDataFile, FTableName);
  Reset(FDataFile);
  try
     FIndexList.LoadFromFile(FIdxName); // Загрузка из файла
                                      // списка индексов TList
    FRecordPos := -1;                 // Установка указателя
                                      // текущей записи в BOF
    BookmarkSize := SizeOf(Integer);  // Определение размера
                                      // закладки для VCL
    InternalInitFieldDefs;            // Инициализация объектов FieldDef
    // Создание компонентов TField, если не создано постоянных полей
    if DefaultFields then CreateFields;
    BindFields(True);                 // Связывание объектов FieldDef
                                      // с реальными данными
  except
    CloseFile(FDataFile);
    FillChar(FDataFile, SizeOf(FDataFile), 0);
    raise;
  end;
end;

procedure TDDGDataSet.InternalPost;
var
  RecPos, InsPos: Integer;
begin
  if FRecordPos = -1 then
    RecPos := 0
  else begin
    if State = dsEdit then RecPos := Integer(FIndexList[FRecordPos])
    else RecPos := FileSize(FDataFile);
  end;
  Seek(FDataFile, RecPos);
  BlockWrite(FDataFile, PDDGData(ActiveBuffer)^, 1);
  if State <> dsEdit then
  begin
    if FRecordPos = -1 then InsPos := 0
    else InsPos := FRecordPos;
    FIndexList.Insert(InsPos, Pointer(RecPos));
```

```
  end;
  FIndexList.SaveToFile(FIdxName);
end;

function TDDGDataSet.IsCursorOpen: Boolean;
begin
  // "Курсор" открыт, если открыт файл
  Result := TFileRec(FDataFile).Mode <> 0;
end;

function TDDGDataSet.GetRecordCount: Integer;
begin
  Result := FIndexList.Count;
end;

function TDDGDataSet.GetRecNo: Integer;
begin
  UpdateCursorPos;
  if (FRecordPos = -1) and (RecordCount > 0) then
    Result := 1
  else
    Result := FRecordPos + 1;
end;

procedure TDDGDataSet.SetRecNo(Value: Integer);
begin
  if (Value >= 0) and (Value <= FIndexList.Count-1) then
  begin
    FRecordPos := Value - 1;
    Resync([]);
  end;
end;

procedure TDDGDataSet.SetTableName(const Value: string);
begin
  CheckInactive;
  FTableName := Value;
  if ExtractFileExt(FTableName) = '' then
    FTableName := FTableName + feDDGTable;
  FIdxName := ChangeFileExt(FTableName, feDDGIndex);
end;

procedure Register;
begin
  RegisterComponents('DDG', [TDDGDataSet]);
end;

function TDDGDataSet.GetDataFileSize: Integer;
begin
  Result := FileSize(FDataFile);
```

```
end;

procedure TDDGDataSet.EmptyTable;
var
  HFile: THandle;
begin
  Close;

  DeleteFile(FTableName);
  HFile := FileCreate(FTableName);
  FileClose(HFile);

  DeleteFile(FIdxName);
  HFile := FileCreate(FIdxName);
  FileClose(HFile);

  Open;
end;

end.
```

Резюме

Прочитав эту главу, вы узнали, как расширить функциональность приложений работы с базами данных за счет добавления возможностей, отсутствующих в библиотеке VCL. Были рассмотрены некоторые правила, лежащие в основе реализации прямых обращений к механизму BDE из приложений Delphi. Мы познакомились также с расширениями компонента `TTable`, позволяющими работать с таблицами Paradox и dBASE. И, наконец, шаг за шагом нами был пройден весь путь создания потомка компонента `TDataSet`. В следующей главе, "Компоненты WebBroker открывают двери в Internet", вы узнаете, как создать серверные приложения для Web и как передавать данные клиентам в Web в реальном времени.

Глава

31 Компоненты WebBroker открывают двери в Internet

Расширения Web-серверов: ISAPI, NSAPI и CGI 702

Создание Web-приложений с помощью Delphi 704

Создание динамических HTML-страниц 711

Поддержка информации о пользователях с помощью cookies 720

Перенаправление на другой Web-узел 724

Считывание информации из HTML-форм 725

Формирование потоков данных 727

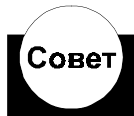
Резюме 731

Для многих владельцев компьютеров использование Internet — нечто само собой разумеющееся. Кажущаяся простота работы в Internet привела к созданию в рамках организаций мини-Internet, которые получили название "intranet", т.е. небольших внутренних Web-сетей, обеспечивающих доступ только в пределах одной организации. Они проявили себя как относительно недорогое, но весьма эффективное средство создания информационных систем предприятий и организаций. Благодаря разработке новых технологий intranet в некоторых случаях может даже перерастать в "extranet", т.е. в сеть с ограниченным доступом, но уже выходящую за рамки одной организации.

Все это, конечно, делает программирование для Internet и intranet очень важным аспектом работы любого программиста. Ну а Delphi, как вы, наверное, догадываетесь, превращает программирование для Internet/intranet в очень простую задачу. Эта простота достигается благодаря следующему:

- инкапсуляции протокола HTTP (Hypertext Transfer Protocol — Протокол передачи гипертекста) в легкодоступных объектах;
- предоставлению среды разработки на основе интерфейса прикладного программирования (API) для самых популярных и мощных Web-серверов;
- использованию подхода RAD (Rapid Application Development — Быстрая разработка приложений) для создания расширений Web-серверов.

С помощью Delphi и входящих в ее состав компонентов WebBroker можно легко создавать расширения Web-серверов, обеспечивающих подготовку настраиваемых динамических HTML-страниц (HTML — Hypertext Markup Language — Язык разметки гипертекста), включающих доступ к данным практически из любого источника.



Компоненты WebBroker поставляются только в варианте Delphi Enterprise Edition. Если вы — профессиональный пользователь Delphi, то можете приобрести компоненты WebBroker в виде отдельной надстройки. За дополнительной информацией обращайтесь на Web-узел компании Borland (`http://www.borland.com`).

Базовая технология доступа к Web довольно проста. Два агента — Web-клиент и Web-сервер — должны установить между собой связь для обмена информацией. Клиент запрашивает некоторую информацию, а сервер ему ее предоставляет. Конечно же, клиент и сервер должны договориться о характере связи и о том, в какой форме будет передаваться совместно используемая ими информация. В Web-среде все действия реализуются в виде потоков ASCII-байтов. Клиент посылает текстовый запрос и получает назад текстовый ответ. Web-клиент очень мало знает о том, что именно происходит на сервере. Подобная простота взаимодействия позволяет организовывать межплатформенную связь, которая обычно реализуется посредством использования протокола TCP/IP.

В качестве стандартного средства осуществления взаимодействий в Web-среде используется протокол передачи гипертекста (HTTP). Под понятием *протокол* обычно понимается соглашение о порядке выполнения некоторых действий, а HTTP — это протокол, разработанный для передачи информации от клиента серверу в форме запроса и от сервера клиенту в форме ответа на запрос. Этот протокол предусматривает представление информации в виде потока байтов, каждый из которых является ASCII-символом, и пересылку этой информации между двумя агентами. Сам по себе протокол HTTP является гибким и мощным средством передачи информации, а в сочетании с языком HTML он позволяет быстро и легко пересылать в броузер требуемые Web-страницы.

HTML-запрос может выглядеть примерно так:

```
GET /mysite/webapp.dll/dataquery?name=CharlieTuna&company=Borland HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.mysite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

HTTP не использует информации о состоянии, т.е. сервер не имеет сведений о состоянии клиента. Взаимодействие между сервером и клиентом завершается после удовлетворения запроса. Это делает создание приложений баз данных с помощью HTTP несколько проблематичным, так как многие приложения баз данных опираются на информацию о клиенте, имеющем доступ к реальному набору данных. Информацию о состоянии можно сохранить путем использования *cookies* (читается “кукиз”; термин не имеет “официального” перевода на русский язык, — *прим. ред.*), т.е. фрагментов информации, полученных клиентом как ответ на HTTP-запрос и сохраняемых им. О cookies читайте ниже в этой главе.

Расширения Web-серверов: ISAPI, NSAPI и CGI

Web-серверы — это двигатели, которые обеспечивают функционирование Web. Именно они предоставляют Web-броузерам любую доступную информацию: HTML-страницы, Java-аплеты или элементы управления ActiveX. Другими словами, Web-серверы являются теми инструментами, которые предоставляют ответы на запросы клиента. Большое количество разнообразных Web-серверов создано для разных вычислительных платформ.

Интерфейс CGI

Первые Web-серверы были способны лишь просто считывать и возвращать клиенту уже существующие статические HTML-страницы. Менеджеры Web-узлов не могли предоставить посетителям их узлов ничего, кроме Web-страниц, уже имевшихся на сервере в момент запроса. Однако вскоре стало ясно, что требуется более высокий уровень взаимодействия между клиентом и сервером, вследствие чего был разработан общий интерфейс маршрутизации (Common Gateway Interface — CGI). Интерфейс CGI позволяет Web-серверу запускать независимый процесс, основываясь на входных данных, полученных от пользователя, обрабатывать эту информацию и возвращать клиенту динамически создаваемую Web-страницу. При этом CGI-программа могла выполнять любой тип обработки данных, который требовался программисту, а также возвращать страницу любого вида, допустимого в рамках HTML.

Стандартные CGI-приложения считывают информацию из стандартного потока ввода (STDIN) и записывают результаты в стандартный поток вывода (STDOUT), а также считывают переменные окружения. Функционирование интерфейса WinCGI включает следующие этапы: сохранение параметров запроса в файле, запуск приложения WinCGI, считывание и обработка данных файла с последующей записью HTML-файла. Именно этот файл затем возвращается клиенту Web-сервером. Тем самым Web сделала большой шаг вперед, поскольку серверы “научились” предоставлять обработанные, уникальные ответы на запросы пользователей.

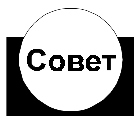
Однако приложениям CGI и WinCGI присущи и некоторые недостатки. Каждый запрос должен запускать на сервере свой собственный процесс, и поэтому несколько запросов могут существенно замедлить работу даже умеренно загруженного сервера — ведь ему приходится выполнять такие относительно медленные задачи, как создание файла, запуск отдельного процесса, его выполнение, запись и возвращение другого файла.

Интерфейсы ISAPI и NSAPI

Основные производители Web-серверов — компании Microsoft и Netscape — быстро поняли слабые стороны CGI-программирования, но при этом они не могли не видеть и преимуществ создания динамических Web-страниц. В результате, чтобы избежать использования отдельного процесса для каждого запроса, обе компании разработали собственные интерфейсы API для своих Web-серверов, которые позволяли расширять возможности серверов с помощью функций библиотек DLL. Библиотеки DLL с интерфейсными функциями можно загрузить один раз, после чего они будут готовы отвечать на любое количество запросов. Они работают как часть процесса Web-сервера, выполняясь в том же пространстве адресов памяти, в котором работает и сам Web-сервер. Вместо передачи информации в обе стороны в виде файлов, теперь расширения Web-серверов передают информацию в пределах одного и того же адресного пространства, без необходимости записи в файл. Благодаря этому Web-приложения стали работать быстрее, с большей эффективностью и с меньшим потреблением ресурсов.

Компания Microsoft разработала довольно простой интерфейс — *Internet Server Application Programming Interface* (ISAPI) — для своего информационного сервера Internet (Internet Information Server — IIS), а компания Netscape для своего семейства Web-серверов предложила более сложный интерфейс — *Netscape Application Programming Interface* (NSAPI).

Delphi предоставляет доступ к обоим типам API с помощью модулей `NSAPI.PAS` и `ISAPI.PAS`. Для запуска приложений, рассматриваемых в этой главе, необходимо запустить сервер IIS, сервер Netscape или один из условно-бесплатных или свободно распространяемых серверов, соответствующих спецификации ISAPI.



Если в данный момент Web-сервер у вас не установлен, можно загрузить приложение Microsoft Personal Web Server с Web-узла `http://www.microsoft.com`. Это свободно распространяемый продукт, который отвечает требованиям ISAPI. С его помощью можно запустить любые примеры, рассматриваемые в этой главе.

Использование Web-серверов

Какой бы сервер вы ни применяли, при выполнении приложений Web-сервера необходимо учитывать некоторые моменты. Прежде всего, поскольку расширения представляют собой библиотеки DLL, они будут загружены в память и останутся там до тех пор, пока работает Web-сервер. Следовательно, если вы создаете и тестируете приложения Delphi, вам, скорее всего, придется закрыть Web-сервер, чтобы перекомпилировать приложение, поскольку Windows не позволит перезаписать выполняемый в данный момент файл. Это требование не является догмой для всех Web-серверов, но обязательно для Microsoft Personal Web Server. Кроме того, обычно Web-серверы требуют выбрать некоторый каталог вашей системы как базовый, который будет служить корневым каталогом для всех HTML-файлов. Можно дать указание Delphi создавать Web-приложения прямо в этом каталоге, введя его полный путь в текстовое поле Output Directory, расположенное во вкладке Directories/Conditionals диалогового окна Project Options. (Это окно открывается посредством выбора команды Project⇨Options.) Наконец, можно даже отлаживать Web-приложения прямо в процессе их работы. О том, как это делается, вы узнаете из документации Delphi (соответствующие инструкции можно найти также в интерактивной справочной системе, в разделе ISAPI⇨Debugging). При этом Web-сервер используется в качестве главного приложения (host application). Однако необходимо учитывать, что все Web-серверы конфигурируются несколько по-разному, поэтому перед работой стоит заглянуть в документацию к вашему конкретному серверу и в документацию Delphi.

Создание Web-приложений с помощью Delphi

Разработка приложений для Internet/intranet в среде Delphi существенно упрощается благодаря наличию компонентов WebBroker. В следующих разделах мы поближе познакомимся с этими компонентами, и вы поймете, что благодаря их наличию можно сосредоточиться на содержимом Web-серверов, не беспокоясь о деталях реализации протоколов связи HTTP.

Классы TWebModule и TWebDispatcher

При выборе в меню Delphi команды File⇨New открывается диалоговое окно New Items. Выберите в нем пиктограмму Web Server Application. В результате будет запущен мастер, с помощью которого можно выбрать тип расширения Web-сервера. Предлагается три варианта: приложения ISAPI/NSAPI, CGI и Win-CGI. В этой главе мы обсудим приложения ISAPI/NSAPI. Создание CGI-расширений серверов выполняется приблизительно так же, но с приложениями ISAPI несколько проще работать.

На заметку

В Delphi имеется также проект ISAPITER.DPR, с помощью которого модули ISAPI можно запускать на Web-сервере, поддерживающем интерфейс NSAPI. В справочной системе содержится информация о том, как установить Web-сервер компании Netscape, чтобы запустить библиотеки ISAPI DLL.

После выбора типа приложения Delphi создает проект на базе класса TWebModule. Главный проект представляет собой библиотеку DLL, а главный модуль содержит класс TWebModule, который является производным от класса TDataModule и содержит всю логику, необходимую для получения HTML-запроса и ответа на него. Класс TWebModule, подобно своему предку, может принимать только невизуальные элементы управления. Для наполнения класса TWebModule можно использовать все элементы управления базами данных, а также элементы вкладки Internet палитры компонентов, связанные с созданием HTML-данных. Это позволит добавлять бизнес-правила для Web-приложений подобно тому, как это можно было бы сделать с помощью класса TDataModule в обычных приложениях.

Класс TWebModule обладает свойством Actions, которое содержит коллекцию элементов TWebActionItem. Элемент TWebActionItem позволяет выполнять код, основанный на определенном запросе. Каждый элемент TWebActionItem имеет собственное имя, а когда клиент посылает основанный на этом имени запрос, выполняется подготовленный код и формируется соответствующий ответ.

На заметку

Приложение Web-сервера можно создать с помощью одного из уже существующих модулей данных. Одним из полей класса TWebModule служит класс TWebDispatcher, включенный в виде самостоятельного компонента в палитру компонентов. Если в создаваемом приложении Web-сервера с помощью менеджера проектов заменить предлагаемый по умолчанию модуль TWebModule существующим модулем данных, то достаточно будет поместить в него компонент TWebDispatcher — и этот модуль превратится в приложение Web-сервера. Компонент TWebDispatcher вкладки Internet палитры компонентов включает все функциональные возможности, инкапсулированные в классе TWebModule. Поэтому, если все необходимые бизнес-правила уже оформлены в виде существующего модуля данных (класса TDataModule), сделать эти правила доступными Web-приложениям очень

На заметку

просто — достаточно указать мышью и щелкнуть. Класс TDataModule с компонентом TWebDispatcher функционально эквивалентен классу TWebModule. Единственное отличие заключается в том, что доступ к действиям протокола HTTP осуществляется через компонент TWebDispatcher, а не непосредственно силами самого класса TDataModule.

Выберите компонент TWebModule, чтобы его свойства были отображены в окне инспектора объектов. Выберите свойство Actions и либо дважды щелкните на нем, либо вызовите редактор этого свойства щелчком на маленькой кнопке с многоточием. Откроется диалоговое окно WebModule Actions. Щелкните на кнопке New и выберите в раскрывшемся окне элемент WebActionItem. В результате в окне инспектора объектов будут отображены свойства элемента Action. Выберите свойство PathInfo и введите "/test" в качестве его значения. Затем перейдите во вкладку Events инспектора объектов и дважды щелкните на событии OnAction, чтобы создать новый обработчик событий. Он будет иметь следующий вид:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin

end;
```

Этот обработчик событий содержит всю информацию о запросе, который сгенерировал это действие, и средства для ответа на него. Содержимое запроса клиента содержится в параметре Request, который имеет тип TWebRequest. Параметр Response имеет тип TWebResponse и используется для отправки требуемой информации обратно клиенту. Внутри этого обработчика события можно написать любой код, необходимый для ответа на запрос, включая обработку файла, действия над базой данных или еще что-нибудь, нужное для построения отправляемой клиенту HTML-страницы.

Прежде чем погрузиться в глубины класса TWebModule, рассмотрим простой пример, демонстрирующий основные принципы работы приложения Web-сервера. Простейший способ создать HTML-страницу, которая отвечает на запрос клиента, — построить ее HTML-текст непосредственно при выполнении программы. Эта задача легко решается с помощью класса TStringList. После размещения HTML-страницы в компоненте TStringList ее без труда можно будет присвоить свойству Content параметра Response. Свойство Content представляет собой строку, которая используется для хранения HTML-страницы, предназначенной для возвращения клиенту. Это единственное свойство параметра Response, которое должно быть заполнено, поскольку оно содержит данные, подлежащие отображению. Если это свойство остается пустым, броузер клиента сообщит, что запрашиваемый документ пуст. В листинге 31.1 представлен код, который следует поместить в элемент действия /test обработчика событий.

Листинг 31.1. Обработчик событий WebModule1WebActionItem1Action

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  Page: TStringList;
begin
  Page := TStringList.Create;
  try
```

```
    with Page do
    begin
      Add('<HTML>');
      Add('<HEAD>');
      Add('<TITLE>Web Server Application -- Basic Sample</TITLE>');
      Add('</HEAD>');
      Add('<BODY>');
      Add('<B>This page was created on the fly by Delphi</B><P>');
      Add('<HR>');
      Add('See how easy it was to create a page on the fly with Delphi''s
      ↳Web Extensions?');
      Add('</BODY>');
      Add('</HTML>');
    end;
      Response.Content := Page.Text;
  finally
    Page.Free;
  end;
  Handled := True;
end;
```

Сохраните этот проект под именем `SAMPLE1.DLL`, скомпилируйте его и поместите полученный в результате файл в используемый по умолчанию каталог Web-сервера (поддерживающего интерфейс ISAPI или NSAPI). Затем направьте свой броузер по адресу:

<*каталог по умолчанию*>/sample1.dll/test

В окне броузера вы должны увидеть ожидаемую Web-страницу, как показано на рис. 31.1.

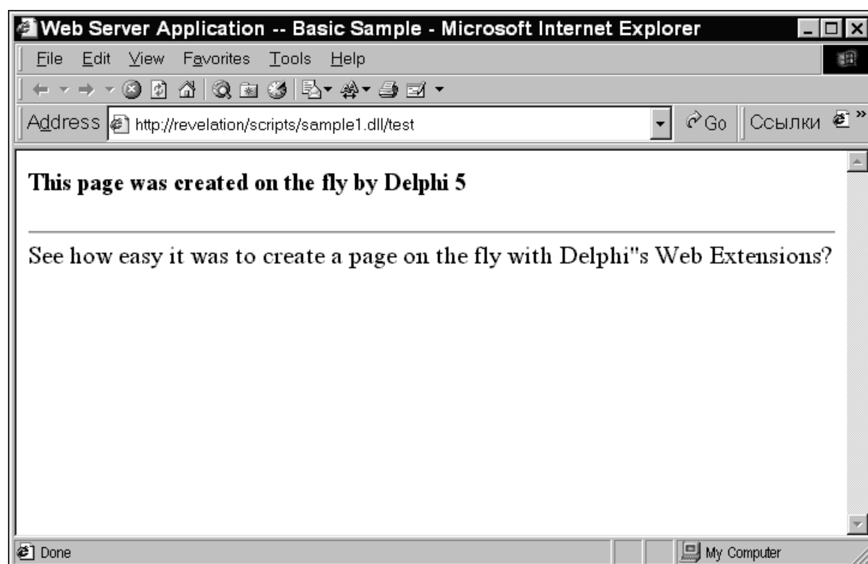


Рис. 31.1. Пример Web-страницы

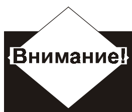
На заметку

При копировании листинга 31.1 с прилагаемого компакт-диска создайте на своем компьютере ту же иерархию каталогов, что и на компакт-диске. В этом случае вы без труда сможете настроить Web-сервер для получения доступа ко всем HTML-страницам и библиотекам DLL с целью запуска всех примеров, приведенных в этой главе. Просто определите виртуальный каталог Web-сервера в качестве корневого и каталог ISAPI, указывающий на каталог `\bin`. Открытие в корневом каталоге файла `INDEX.HTM` позволит получить доступ ко всему исходному коду. Обратите внимание на то, что при копировании файлов с компакт-диска у них будет установлен атрибут "только для чтения". Если впоследствии потребуется эти файлы отредактировать, то данный флажок необходимо будет сбросить в окне программы **Проводник**.

Обратите внимание на то, что результатом компиляции этого проекта является библиотека DLL, которая удовлетворяет спецификации ISAPI. Исходный код этого проекта выглядит следующим образом:

```
library Sample1;
uses
  WebBroker,
  HTTPApp,
  ISAPIApp,
  Unit1 in 'Unit1.pas' { WebModule1: TWebModule} ;
{$R *.RES}
exports
  GetExtensionVersion,
  HttpExtensionProc,
  TerminateExtension;
begin
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end.
```

Обратите внимание на три экспортируемые процедуры: `GetExtensionVersion()`, `HttpExtensionProc()` и `TerminateExtension()`. Это единственные процедуры, которые требуются спецификацией ISAPI.



Подобно любому типичному приложению, каждое ISAPI-приложение использует глобальный объект `Application`. Однако, в отличие от обычного приложения, этот тип проекта не использует модуль `Forms`. Переменная `Application` типа `TWebApplication` объявляется в модуле `WebBroker`. Ее подпрограммы обрабатывают все специальные вызовы, необходимые Web-серверу, поддерживающему интерфейс ISAPI или NSAPI. Поэтому никогда не следует пытаться добавить модуль `Forms` в приложение-расширение для поддерживающего ISAPI Web-сервера — это может заставить компилятор использовать неправильную переменную `Application`.

На примере этого небольшого проекта демонстрируется простота построения в среде Delphi приложения для Web-сервера, способного предоставлять ответ на запрос клиента. Это относительно простой пример, в котором HTML-страница динамически создается непосредственно в самом коде. Но, как вы увидите позже, в Delphi предусмотрены средства генерирования ответов гораздо более сложным и интересным способом, однако для этого необходимо глубже вникнуть в работу приложения с компонентами WebBroker. Поэтому вашему вниманию предлагается следующий раздел.

Классы TWebRequest и TWebResponse

Классы TWebRequest и TWebResponse являются абстрактными классами, инкапсулирующими протокол HTTP. Класс TWebRequest предоставляет доступ ко всем данным, передаваемым серверу клиентом, а в классе TWebResponse содержатся свойства и методы, которые позволяют ответить на запрос, причем любым из допускаемых протоколом HTTP способов. Оба класса объявлены в модуле HTTPAPP.pas, который используется модулем WebBroker.pas. Web-приложения, ориентированные на интерфейс ISAPI, обычно используют классы TISAPIResponse и TISAPIRequest, которые являются потомками этих абстрактных классов и объявлены в модуле ISAPIAPP.PAS. Благодаря полиморфизму, Delphi может передавать классы TISAPIxxx в качестве параметров типа TWebxxx обработчика события OnAction класса TWebModule.

Класс TISAPIRequest содержит всю информацию, передаваемую клиентом при выполнении запроса к Web-странице. Из этого запроса можно выбрать информацию о клиенте. Для любого конкретного запроса многие свойства могут оставаться пустыми, поскольку зачастую для HTTP-запросов заполняются не все поля. В свойствах RemoteHost и RemoteAddr содержится IP-адрес запрашивающего компьютера, а в свойстве UserAgent — информация об используемом клиентом броузере. Свойство Accept включает список типов графических изображений, которые этот броузер способен выводить; а свойство Referer — URL страницы, на которой пользователь создал данный запрос. Если была предоставлена информация об обработке cookies (они рассматриваются ниже в этой главе), то она будет доступна в свойстве Cookie. Доступ к нескольким элементам cookies упрощается благодаря использованию массива CookieFields. Все переданные вместе с запросом параметры содержатся в одной строке внутри свойства Query. К ним также можно получить доступ как элементам массива QueryFields.

На заметку

При передаче параметров по адресу URL они обычно следуют за знаком вопроса (?) после значения URL. Несколько параметров разделяются символом амперсанта (&), а если параметры содержат пробелы, то эти пробелы заменяются знаками “плюс” (+). Следовательно, допустимый набор параметров внутри HTML-страницы может выглядеть примерно так:

```
<A HREF="http//www.someplace.com/ISAPIApp?Param1=This+Parameter
↳&Param2=That+Parameter">Some Link</A>
```

(Ответ на резонный вопрос: “Как отличить сам символ + от такого же символа, но указанного вместо пробела?” — весьма несложен: “Многие символы запроса (в том числе +) передаются в виде %XX, где XX — шестнадцатеричное представление символа”. — *Прим. ред.*)

Большая часть информации класса TISAPIRequest содержится в свойствах, но этот класс также оставляет открытыми многие функции, используемые для заполнения его свойств, тем самым позволяя при необходимости получать непосредственный доступ к данным. Класс TISAPIRequest, помимо рассмотренных, включает и другие свойства, но упомянутые выше являются наиболее важными из них. Все эти свойства в создаваемых обработчиках события OnAction можно использовать с целью определения типа ответа, который будет предоставляться приложением Web-сервера. Если, например, требуется включить в предоставляемый ответ информацию об IP-адресе пользователя и построить ответ исходя из типа используемого пользователем броузера, — все это вполне можно будет выполнить в обработчике события OnAction.

Увидеть, как выглядит класс TISAPIRequest на практике, можно будет после запуска предлагаемого ниже проекта на вашем Web-сервере. Создайте новое приложение Web-сервера, откройте редактор свойства Actions, дважды щелкнув на этом свойстве в окне инспектора объектов, и создайте новый элемент TWebActionItem, установив свойство PathInfo равным http. Перейдите во вкладку Internet палитры компонентов и поместите в модуль WebModule компонент TPageProducer (рассматриваемый ниже в этой главе). В обработчик события OnAction добавьте код, представленный в листинге 31.2.

Листинг 31.2. Обработчик события OnAction

```
procedure TWebModule1.WebModule1Actions0Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  Page: TStringList;
begin
  Page := TStringList.Create;
  try
    with Page do
    begin
      Add('<HTML>');
      Add('<HEAD>');
      Add('<TITLE>Web Server Extensions THTTPRequest Demo</TITLE>');
      Add('</HEAD>');
      Add('<BODY>');
      Add('<H3><FONT="RED">This page displays the properties
        ➥of the HTTP request that asked for it.</FONT></H3>');
      Add('<P>');

      Add('Method = ' + Request.Method + '<BR>');
      Add('ProtocolVersion = ' + Request.ProtocolVersion + '<BR>');
      Add('URL = ' + Request.URL + '<BR>');
      Add('Query = ' + Request.Query + '<BR>');
      Add('PathInfo = ' + Request.PathInfo + '<BR>');
      Add('PathTranslated = ' + Request.PathTranslated + '<BR>');
      Add('Authorization = ' + Request.Authorization + '<BR>');
      Add('CacheControl = ' + Request.CacheControl + '<BR>');
      Add('Cookie = ' + Request.Cookie + '<BR>');
      Add('Date = ' + FormatDateTime ('mmm dd, yyyy hh:mm',
        ➥Request.Date) + '<BR>');
      Add('Accept = ' + Request.Accept + '<BR>');
      Add('From = ' + Request.From + '<BR>');
      Add('Host = ' + Request.Host + '<BR>');
      Add('IfModifiedSince = ' + FormatDateTime ('mmm dd, yyyy hh:mm',
        ➥Request.IfModifiedSince) + '<BR>');
      Add('Referer = ' + Request.Referer + '<BR>');
      Add('UserAgent = ' + Request.UserAgent + '<BR>');
      Add('ContentEncoding = ' + Request.ContentEncoding + '<BR>');
      Add('ContentType = ' + Request.ContentType + '<BR>');
      Add('ContentLength = ' + IntToStr(Request.ContentLength) + '<BR>');
      Add('ContentVersion = ' + Request.ContentVersion + '<BR>');
```

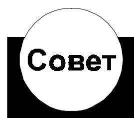
```
    Add('Content = ' + Request.Content + '<BR>');
    Add('Connection = ' + Request.Connection + '<BR>');
    Add('DerivedFrom = ' + Request.DerivedFrom + '<BR>');
    Add('Expires = ' + FormatDateTime ('mmm dd, yyyy hh:mm',
      ↳Request.Expires) + '<BR>');
    Add('Title = ' + Request.Title + '<BR>');
    Add('RemoteAddr = ' + Request.RemoteAddr + '<BR>');
    Add('RemoteHost = ' + Request.RemoteHost + '<BR>');
    Add('ScriptName = ' + Request.ScriptName + '<BR>');
    Add('ServerPort = ' + IntToStr(Request.ServerPort) + '<BR>');

    Add('</BODY>');
    Add('</HTML>');
  end;
    PageProducer1.HTMLDoc := Page;
    Response.Content := PageProducer1.Content;
 finally
  Page.Free;
 end;
 Handled := True;
end;
```

Постройте проект и скопируйте результирующий файл Project.dll в каталог, используемый по умолчанию Web-сервером, поддерживающим ISAPI или NSAPI. Укажите своему броузеру адрес http://<*ваш сервер*>/project1.dll/http — и это приложение отобразит все значения полей HTTP, переданные серверу в запросе от вашего броузера.

Конечно же, на каждый запрос должен поступить соответствующий ответ, поэтому в Delphi определен класс TISAPIResponse, позволяющий возвращать информацию клиенту, отправившему запрос. Самым важным свойством класса TISAPIResponse является свойство Content. В этом свойстве будет содержаться HTML-текст, предназначенный для отображения в броузере клиента.

Класс TISAPIResponse содержит несколько дополнительных свойств, которые можно устанавливать в создаваемом приложении. В свойство Version можно поместить информацию о версии. С помощью свойства LastModified можно сообщить клиенту, когда последний раз была модифицирована передаваемая ему информация. Используя свойства ContentEncoding, ContentType и ContentVersion, можно передать сведения о самом содержимом. И, наконец, свойство StatusCode позволяет возвращать клиенту коды ошибок и другие коды состояния.



Большинство броузеров специфически реагирует на определенные значения свойства StatusCode. Эту информацию можно уточнить, просмотрев спецификацию HTTP для определенных кодов состояния на Web-узле, расположенном по адресу: http://www.w3.org.

Реальную мощь класса TISAPIResponse составляют его методы. Метод SendResponse() заставляет приложение Web-сервера немедленно вернуть ответ, не ожидая завершения работы обработчика события OnAction. Используя метод SendStream(), можно отправить клиенту любой тип данных. Если необходимо, чтобы приложение отправило клиенту нечто, отличное от стандартного ответа, обеспечиваемого самим этим приложением, подобное требование можно реализовать с помощью метода SendRedirect(), который рассматривается ниже в этой главе.

Создание динамических HTML-страниц

Конечно же, динамическое построение HTML-кода нельзя считать самым эффективным способом создания Web-страниц, поэтому Delphi предоставляет несколько инструментов, упрощающих решение этой задачи. Так, в Delphi имеется абстрактный класс `TCustomContentProducer`, обеспечивающий базовые функции, необходимые для обработки и манипуляции HTML-страницами. Производными от этого класса являются классы `TPageProducer`, `TDataSetTableProducer` и `TQueryTableProducer`. Используемые совместно, в совокупности с уже существующими либо динамически создаваемыми HTML-страницами, эти компоненты позволяют создать узел из динамических Web-страниц, включающих табличные данные, гиперссылки и полный набор всех прочих возможностей, поддерживаемых языком HTML. Эти элементы управления не предназначены для автоматической генерации HTML-кода, однако их использование существенно упрощает работу с HTML-кодом и динамическое создание Web-страниц на основе параметров и других входных данных.

Компонент `TPageProducer`

Компонент `TPageProducer` используется для манипуляции обычным HTML-кодом. Во время разработки или в процессе выполнения создается HTML-шаблон, содержащий специальные дескрипторы, игнорируемые в стандартном языке HTML. Компонент `TPageProducer` способен обнаруживать эти дескрипторы и заменять их соответствующей информацией. В дескрипторах могут содержаться параметры для размещения передаваемых данных. Можно даже заменить один настраиваемый дескриптор текстом, содержащимся в других настраиваемых дескрипторах. Это позволяет связать несколько страниц и построить из них последовательную цепочку, образующую Web-узел с динамическим содержимым, определяемым входными данными.

Упомянутые динамические дескрипторы выглядят как обычные дескрипторы языка HTML, однако они игнорируются броузером клиента, поскольку являются нестандартными. Такой дескриптор может выглядеть следующим образом:

```
<#CustomTag Param1=SomeValue "Param2=Some Value with Spaces">
```

Настраиваемый дескриптор заключается в угловые скобки (`<` и `>`), а его имя начинается с символа `#`. При этом имя должно быть допустимым идентификатором языка Pascal. Параметры с пробелами заключаются в кавычки. Настраиваемые дескрипторы могут содержаться с любом месте документа HTML, в том числе и внутри других дескрипторов HTML.

В Delphi предусмотрен ряд встроенных имен дескрипторов. При этом ни одно из значений не имеет специального действия, связанного с ними, — они определяются лишь для удобства и ясности кода. Например, вовсе не обязательно использовать дескриптор `tgLink` для определения гиперсвязи, но имеет смысл применять его именно с этой целью — просто с точки зрения понятности создаваемых шаблонов. Заметим, что можно пользоваться и исключительно собственными настраиваемыми дескрипторами. При этом все они будут иметь значение `tgCustom`. В табл. 31.1 перечислены существующие предопределенные дескрипторы.

Таблица 31.1. Предопределенные дескрипторы

Имя	Значение	Значение преобразования дескриптора
Custom	TgCustom	Определенный пользователем или неопределенный дескриптор. Преобразуется в любое значение, определенное пользователем
Link	TgLink	Преобразуется в якорь. Обычно это гипертекстовая связь или значение закладки (`<A>..</A>`)
Image	TgImage	Преобразуется в дескриптор изображения (`<IMG SRC=...>`)
Table	TgTable	Этот дескриптор заменяется HTML-таблицей (`<TABLE>..</TABLE>`)
ImageMap	TgImageMap	Заменяется графической картой. Графическая карта определяет связи на базе зон внутри некоторого изображения (`<MAP>...</MAP>`)
Object	TgObject	Этот дескриптор заменяется кодом, вызывающим элемент управления ActiveX
Embed	TgEmbed	Преобразуется в дескриптор, который относится к Netscape-совместимой DLL расширения

Использование компонента TPageProducer вполне очевидно. Этому компоненту назначается некоторый HTML-код, помещаемый либо в свойство HTMLDoc, либо в свойство HTMLFile. При назначении свойства Content другой переменной (обычно это свойство TISAPIResponse.Content) выполняется сканирование помещенного в него HTML-кода. При обнаружении в этом коде любого пользовательского дескриптора генерируется событие OnHTMLTag. Обработчик события OnHTMLTag выглядит примерно так:

```
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
end;
```

В параметре Tag указывается тип обнаруженного дескриптора (см. табл. 31.1). Параметр TagString содержит значение самого дескриптора в полном объеме. Параметр TagParams представляет собой индексированный список всех параметров, включая имя параметра, знак равенства (=) и само значение. Параметр ReplaceText — это строковая переменная, которую следует заполнить новым значением, предназначенным для замещения дескриптора. В HTML-коде весь текст дескриптора, включая угловые скобки (< и >), заменяется тем значением, которое передается назад этому параметру.

Назначить HTML-шаблон компоненту TPageProducer можно одним из двух способов: либо во время выполнения создать HTML-код в виде строки и присвоить эту строку свойству HTMLDoc, либо назначить свойству HTMLFile уже существующий HTML-файл. Это позволяет как строить HTML-текст прямо по ходу выполнения программы, так и использовать уже существующие шаблоны, подготовленные заранее.

Предположим, например, что имеется HTML-файл MYPAGE.HTM, содержащий следующий HTML-текст:

```
<HTML>
<HEAD>
  <TITLE>My Cool Homepage</TITLE>
</HEAD>
```

```
<BODY>
Howdy <#Name>!  Thanks for stopping by my web site!
</BODY>
</HTML>
```

Обработчику события PageProducer.OnHTMLTag можно назначить следующий код:

```
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  case Tag of
   tgCustom: if TagString = 'Name' then ReplaceText := 'Partner';
  end;
end;
```

В результате работы этого обработчика события образуется HTML-код

```
<HTML>
<HEAD>
    <TITLE>My Cool Homepage</TITLE>
</HEAD>
<BODY>
Howdy Partner!  Thanks for stopping by my web site!
</BODY>
</HTML>
```

Этот код можно использовать при обработке события OnAction в модуле WebModule, например, следующим образом:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  PageProducer1.HTMLFile := 'MYPAGE.HTM';
  Response.Content := PageProducer1.Content;
end;
```

Вновь созданную страницу следует отправить клиенту, приславшему запрос. При вызове свойства PageProducer.Content выполняется замена текста для каждого найденного дескриптора путем вызова обработчика события OnHTMLTag. Более сложные страницы могут иметь несколько элементов в операторе case, заменяющих различные пользовательские дескрипторы большими порциями HTML-кода, ссылками на другие страницы, графическими изображениями, таблицами и пр.

Компоненты TCustomPageProducer можно также связывать в цепочку. При этом для построения одной страницы можно использовать два таких компонента. Например, можно создать базовый HTML-шаблон для хранения стандартного текста нижнего и верхнего колонтитулов, а также пользовательских дескрипторов, определяющих некоторые общие значения параметров для Web-страницы и местонахождение главного тела страницы. Затем все это можно пропустить через первый генератор страницы, заменив дескрипторы общих данных информацией, выбранной на основании характеристик пользователя. После этого дескрипторы в главном теле страницы можно заменить пользовательским кодом или другими пользовательскими дескрипторами, в зависимости от запрошенной пользователем

информации. Полученный результат можно будет передать второму компоненту TPageProducer, который заменит вновь добавленные специфические значения дескрипторов соответствующей информацией.

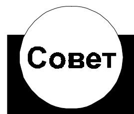
Классы TDataSetTableProducer и TQueryTableProducer

Помимо поддержки обычных HTML-документов, Delphi предоставляет класс TDataSetTableProducer, с помощью которого можно легко и эффективно создавать HTML-таблицы на основе заданного набора данных. Этот класс позволяет полностью настраивать все характеристики таблицы в пределах, установленных правилами языка HTML. По своим функциональным возможностям компонент TDataSetTableProducer во многом напоминает компонент TDBGrid, так как с его помощью можно форматировать отдельные ячейки, строки и столбцы. При этом обеспечивается доступ к любому доступному в системе набору данных, локальному или удаленному. В результате появляется возможность строить Web-узлы корпоративного уровня, имеющие доступ к данным практически из любого источника.

Своим поведением класс TDataSetTableProducer несколько отличается от других элементов управления базами данных, поскольку он получает доступ к данным через класс, производный непосредственно от класса TDataSet, а не от класса TDataSource. Класс TDataSetTableProducer имеет свойство DataSet, которое в период разработки можно установить равным любому потомку класса TDataSet, содержащемуся в том же модуле TWebModule, а во время выполнения — любому динамически создаваемому значению. После установки свойства DataSet класс TDataSetTableProducer можно настроить на отображение любых столбцов выбранного набора данных. С помощью свойства TableAttributes можно установить общие характеристики данной таблицы, естественно, в пределах, допускаемых языком HTML.

Свойства Header и Footer имеют тип TStrings и позволяют добавлять HTML-текст до и после самой таблицы. Эти свойства можно использовать в сочетании с собственным динамически создаваемым HTML-текстом или с текстом из компонента TPageProducer. Например, если главным элементом страницы является таблица, то свойства Header и Footer можно использовать для заполнения базовой структуры HTML-страницы. Если же таблица не является центром построения страницы, для ее размещения в соответствующем месте страницы имеет смысл воспользоваться пользовательским дескриптором TTag компонента TPageProducer. В любом случае, для создания Web-страниц на базе наборов данных можно использовать компонент TDataSetTableProducer.

Настройка самой таблицы выполняется с помощью свойств Columns, RowAttributes и TableAttributes. Свойство Columns имеет довольно мощный редактор компонентов, который можно использовать для установки большинства атрибутов компонентов.



Чтобы открыть редактор свойства Columns, достаточно дважды щелкнуть на самом компоненте или на свойстве Columns в окне инспектора объектов.

Свойства Caption и CaptionAlign определяют способ отображения заголовка таблицы. В свойстве Caption хранится текст, предназначенный для объяснения содержимого таблицы и выводимый либо над, либо под нею. Свойство DataSet (свойство Query в классе TQueryTableProducer) служит для определения данных, используемых в таблице.

За исключением способа получения доступа к данным, классы TDataSetTableProducer и TQueryTableProducer функционируют идентично. Они имеют одни и те же свойства, настраиваемые одинаковым образом. Исходя из этого мы создадим таблицу, которая будет представлять собой результат простого объединения двух других таблиц, а затем на примере класса TQueryTableProducer продемонстрируем методы работы с обоими классами.

Создайте новое Web-приложение и поместите в окно модуля TWebModule компонент TQueryTableProducer из вкладки Internet, а также компоненты TQuery и TSession из вкладки Data Access палитры компонентов. Установите свойство QueryTableProducer1.Query равным значению Query1, а свойство Query1.DatabaseName — DBDEMOS. Сохраните проект под именем TABLEEX.DPR. Затем поместите в свойство Query1.SQL следующее значение:

```
SELECT CUSTNO, ORDERNO, COMPANY, AMOUNTPAID, ITEMSTOTAL FROM CUSTOMER,
↳ORDERS WHERE
   CUSTOMER.CUSTNO = ORDERS.CUSTNO
   AND
   ORDERS.AMOUNTPAID <> ORDERS.ITEMSTOTAL
```

Это приведет к созданию небольшой объединенной таблицы, в которой будут зарегистрированы все заказчики из таблицы CUSTOMER.DB (принадлежащей базе данных с псевдонимом DBDemos), которые еще не оплатили полностью свой заказ. Теперь можно построить HTML-таблицу для отображения этих данных и выделить в ней суммы, подлежащие оплате. Для вывода данных на экран достаточно в редакторе свойства Columns установить свойство Query1.Active равным значению True.

На заметку

В модуль TWebModule любого приложения для Web-сервера, предназначенного для обработки данных с использованием компонентов Delphi для работы с данными, обязательно должен быть включен компонент TSession. К приложениям Web-сервера возможен многократный параллельный доступ, и для каждого поступившего запроса Delphi запустит ISAPI- или NSAPI-приложение сервера в отдельном потоке. В результате при каждом обращении к BDE приложению потребуется иметь собственный уникальный сеанс. Компонент Tsession, у которого свойство AutoSessionName имеет значение True, гарантирует, что каждый поток получит собственный сеанс и не вступит в конфликт с другими потоками, пытающимися получить доступ к одним и тем же данным. Все, что требуется — это поместить в проект компонент Tsession, а об остальном позаботится Delphi.

Совет

При построении приложений Web-расширений использование свойства TWebApplication.CacheConnections может ускорить работу приложения. При каждом поступлении запроса к ISAPI- или NSAPI-приложению, для его обработки генерируется новый поток, причем это происходит в процессе создания нового экземпляра класса TWebModule. Обычно каждый поток используется только для одного соединения, и при закрытии соединения экземпляр класса TWebModule уничтожается. Если свойство CacheConnections равно True, то каждый поток сохраняется и при необходимости используется повторно. Новые потоки создаются только в том случае, если свободный кэшированный поток недоступен. В результате быстродействие приложения увеличивается за счет экономии времени, затрачиваемого на создание очередного экземпляра класса TWebModule. Однако в этом случае следует проявлять повышенную осторожность, поскольку событие TWebModule.OnCreate вызывается для каждого кэшированного потока только один раз. Завершив работу с очередным пользователем, кэшированный поток сохраняет свое конечное состояние. Это может

вызвать проблемы при следующем использовании этого потока, в зависимости от того, что должно происходить при обработке события OnCreate. Если данное событие используется для инициализации переменных или выполнения других начальных действий, то, вероятнее всего, не имеет смысла использовать кэширование связей. Для инициализации данных Web-приложения с кэшируемыми потоками целесообразно использовать дополнительный метод, вызываемый в обработчике события BeforeDispatch. В этом случае используемые в Web-модуле данные будут инициализироваться всякий раз при поступлении очередного запроса.
Текущее количество неиспользуемых кэшированных соединений можно определить с помощью свойства TWebApplication.InactiveCount. Свойство TWebApplication.ActiveCount указывает количество соединений, активных в данный момент. Этих два свойства помогут определить подходящее значение для свойства TWebApplication.MaxConnections, ограничивающее общее число соединений, которые класс TWebModule сможет обрабатывать одновременно. Если же значение свойства ActiveCount превысит значение свойства MaxConnections, будет сгенерирована исключительная ситуация.

Дважды щелкните на компоненте QueryTableProducer1 — и на экране раскроется окно редактора компонента Columns. В верхней левой части редактора компонента можно установить общие свойства таблицы в целом. Нижняя часть редактора содержит элемент управления HTML, который будет отображать таблицу в ее текущем состоянии. Верхняя правая часть окна содержит коллекцию элементов управления THTMLTableColumn, настройка которых определяет, какие поля базы данных будут включены в таблицу и как эти поля будут в ней отображены. Delphi автоматически импортирует поля из компонента TQuery и помещает сведения он них в редактор полей. Создаваемое нами приложение не нуждается в последнем поле таблицы, так что выделите поле ItemsTotal и удалите его. Кроме того, выделите поле AMOUNTPAID и задайте его свойству BgColor значение Lime.

При необходимости измените размеры окна редактора свойства Columns. Это не помешает, особенно в том случае, если таблица содержит много столбцов.

В верхнем левом углу окна установите значение Border равным 1, чтобы иметь возможность видеть границы таблицы в редакторе компонентов в процессе построения. Значение свойства CellPadding установите равным 2, чтобы создать достаточный интервал между границей и текстом. Если вы захотите слегка раскрасить таблицу, установите свойство BgColor равным значению Aqua. Это изменит черный (по умолчанию) цвет фона таблицы на аквамариновый. Обратите внимание на то, что данное значение принимается лишь по умолчанию и его можно переопределить, динамически задавая цвет фона отдельной строки или столбца. Кроме того, учтите, что установки цвета столбца имеют преимущество перед установками цвета для строки.

В Delphi при создании столбцов HTML-таблицы в их заголовки будут помещены имена соответствующих полей исходной таблицы. Однако имена полей базы данных часто оказываются не слишком удачными заголовками столбцов, поэтому вы можете изменить их значения с помощью свойства Title. Это составное свойство, и одним из его подсвойств является Caption. Установите значения свойств Title.Caption четырех столбцов равными "Cust #", "Order #", "Company" и "Amount Owed" соответственно. Значение "Amount Owed", возможно, не вполне подходит для информации, которая сейчас содержится в четвертом

столбце, но впоследствии мы изменим содержимое этого столбца. Свойство Title позволяет также выполнять вертикальное и горизонтальное выравнивание и настраивать цвет ячейки заголовка столбца.

На заметку

Класс THTMLTableColumn, как и другие связанные с таблицами классы, имеет свойство Custom, которое позволяет ввести строковое значение данного элемента в таблице. Это значение будет введено прямо в дескриптор HTML, определяющий данный элемент таблицы. Элементы свойства Custom могут включать HTML-модификаторы ячейки, строки или столбца, не включенные в свойства класса или HTML-расширений. Броузер Microsoft Internet Explorer включает несколько расширений, предназначенных для форматирования таблиц и позволяющих настроить вид их рамки. Если желательно использовать эти возможности, в свойстве Custom создайте элемент формата *имя_параметра=значение*. Можно также добавить несколько параметров, разделив их пробелами.

Этим исчерпываются все основные свойства таблиц, которые можно устанавливать во время разработки. Теперь рассмотрим события, связанные с классом TQueryTableProducer. Они позволяют настраивать таблицы непосредственно во время выполнения приложения. Событие OnCreateContent возникает до генерации любого HTML-кода. Оно содержит параметр Continue типа Boolean, которому можно присвоить необходимое значение. Если ваше приложение определит, что по каким-то причинам таблица не должна быть сгенерирована, достаточно установить этот параметр равным значению False, что исключит какую бы то ни было ее обработку, а обращение к свойству Content вернет пустую строку. Такая возможность приостановки обработки может пригодиться при подготовке запроса, установке свойства TQueryTableProducer.MaxRows или для других действий, выполняемых перед отображением таблицы.

В частности, в рассматриваемом нами примере при отображении таблицы приложение должно обработать каждую запись в результатах выполнения запроса Query. Чтобы запрос гарантированно указывал на нужную запись, приложение должно самостоятельно передвигать курсор в запросе в начале построения каждой новой строки. Иными словами, запрос должен начинаться с самого начала, как в классе TQueryTableProducer. Следовательно, обращение к методу Query1.First в обработчике события OnCreateContent будет гарантировать, что запрос и HTML-таблица синхронизированы относительно друг друга. Поэтому добавьте следующий код в обработчик события QueryTableProducer1.OnCreateContent:

```
procedure TWebModule1.QueryTableProducer1CreateContent(Sender: TObject;
  var Continue: Boolean);
begin
  QueryTableProducer1.MaxRows := Query1.RecordCount;
  Query1.First;
  Continue := True;
end;
```

Событие OnGetTableCaption позволяет отформатировать заголовок таблицы в соответствии с текущими требованиями. Дважды щелкните на этом событии в окне инспектора объектов — и вы увидите следующий обработчик событий:

```
procedure TWebModule1.QueryTableProducer1GetTableCaption(Sender: TObject;
  var Caption: String; var Alignment: THTMLCaptionAlignment);
begin

end;
```

Параметр Caption представляет собой var-параметр, в котором будет храниться окончательный вариант заголовка. Этим параметром можно манипулировать произвольным образом — например, добавлять HTML-дескрипторы для изменения размера, цвета и формата шрифта заголовка таблицы. Параметр Alignment можно использовать для указания местоположения заголовка таблицы (вверху или внизу).

Находясь в нашем приложении, щелкните дважды на событии OnGetTableCaption в инспекторе объектов и создайте его обработчик. Введите следующий код для форматирования заголовка таблицы, чтобы он стал более заметным на странице (эти изменения не должны отражаться на HTML-таблице, показанной в редакторе свойства Columns):

```
procedure TWebModule1.QueryTableProducer1GetTableCaption(Sender: TObject;
  var Caption: String; var Alignment: THTMLCaptionAlignment);
begin
  Caption := '<B><FONT SIZE="+2" COLOR="RED">Delinquent Accounts</FONT></B>';
  Alignment := caTop;
end;
```

Событие OnFormatCell можно использовать для изменения внешнего вида отдельной ячейки. В нашем примере можно добавить код для выделения ячейки Amount Owed любой компании, которая еще не оплатила свой счет полностью. Конечно, это нельзя сделать так просто, как в случае обычных сеток, поскольку класс TQueryTableProducer предоставляет только строковые значения. Но, как упоминалось ранее, при создании таблицы для перемещения курсора компонента TQuery можно использовать параметры CellRow и CellColumn, собирая соответствующие данные и выполняя вычисления непосредственно в процессе обработки каждой строки.

Обработчику события OnFormatCell информация о текущей форматируемой ячейке передается в значениях параметров CellRow и CellColumn. Оба этих параметра могут принимать только целые значения (считая от нуля). Остальные параметры являются var-параметрами, которым можно присваивать значения, зависящие от логики создаваемого приложения. С помощью параметров Align и VAlign к данным ячейки можно применить операции горизонтального и вертикального выравнивания. При этом в параметре CustomAttrs можно передать дополнительные параметры Custom ячейки. Безусловно, в программе доступно и само содержимое ячейки таблицы, помещаемое в параметр CellData, — при необходимости его также можно изменять.

Параметр CellData имеет тип string, что ограничивает возможности обработки содержимого ячейки в его исходном формате. Если же данные хранятся в базе данных в виде целых значений, для обратного преобразования значения ячейки в число потребуется вызвать функцию StrToInt. Один из возможных способов сбора значений типа TField для некоторой ячейки показан в приведенном ниже фрагменте кода. Возможно, в следующей версии Delphi, помимо (или вместо) строкового значения, в обработчик события OnFormatCell будет передаваться и реальное значение компонента TField. Приведенный в листинге 31.3 текст поместите в обработчик события OnFormatCell класса TQueryTableProducer.

Листинг 31.3. Обработчик события OnFormatCell

```
procedure TWebModule1.QueryTableProducer1FormatCell(Sender: TObject;
  CellRow, CellColumn: Integer; var BgColor: THTMLBgColor;
  var Align: THTMLAlign; var VAlign: THTMLVAlign; var CustomAttrs,
  CellData: String);
```

```
var
  Owed, Paid, Total: Currency;
begin
  if CellRow = 0 then Exit; // Не обрабатываем строку заголовка
  if CellColumn = 3 then // Если столбец Amount Owed, выполняем:
  begin
    //Вычисляем объем долга компании
    Paid := Query1.FieldByName('AmountPaid').AsCurrency;
    Total := Query1.FieldByName('ItemsTotal').AsCurrency;
    Owed := Total - Paid;
    //Устанавливаем переменную CellData равной сумме долга
    CellData := FormatFloat('$0.00', Owed);
    //Если эта сумма больше нуля, выделяем ячейку
    if Owed > 0 then
    begin
      BgColor := 'RED';
    end;
    {Продвигаем курсор вперед, поскольку достигнут конец строки. }
    Query1.Next;
  end;
end;
```

В этом коде производится накопление данных по каждому неоплаченному счету, из общей суммы вычитается сумма оплаченных счетов и, если сумма долга остается положительной, ячейки выделяются красным цветом. Этот фрагмент иллюстрирует также способ использования текущего курсора компонента TQuery для получения доступа к данным, отображаемым в таблице HTML.

Затем в свойство TQueryTableProducer.Header добавьте следующие строки:

```
<HTML>
<HEAD>
  <TITLE>Delinquent Accounts</TITLE>
</HEAD>
<BODY>
<CENTER><H2>Big Shot Widgets</H2></CENTER>
<P>
The Accounts highlighted in red are late in paying:
<P>
```

А в свойство TQueryTableProducer.Footer добавьте такие строки:

```
<P>
<I>This information is to be kept in the strictest confidence</I><P>
<B><I>Copyright 1999 by BigShotWidgets</I></B><P>
</BODY>
</HTML>
```

Установка значений этих свойств приведет к размещению таблицы между указанными наборами HTML-кода, а при обращении к свойству Content класса TQueryTableProducer, как показано в приведенном ниже коде, будет создана законченная страница.

И, наконец, вернитесь в главный модуль TWebModule создаваемого приложения и добавьте одно свойство Action, установив его свойство PathInfo равным значению /TestTable. В его обработчик события OnAction добавьте следующий код:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := QueryTableProducer1.Content;
end;
```

Затем скомпилируйте этот проект и убедитесь в том, что созданная библиотека DLL доступна вашему Web-серверу. Теперь, если в броузере вызвать URL http://<*ваш сервер*>/tableex.dll/TestTable, в его окне будет отображена таблица с заголовком и нижним колонтитулом. В ней красным цветом будут выделены ячейки, содержащие сумму долга соответствующих компаний, — как показано на рис. 31.2).

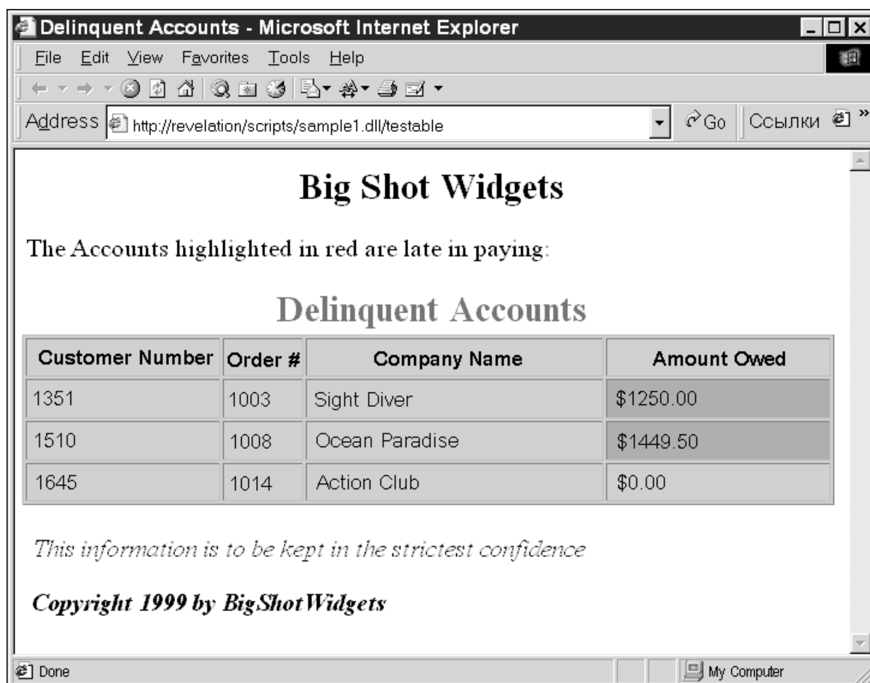


Рис. 31.2. Web-страница, содержащая таблицу с выделенными данными

Поддержка информации о пользователях с помощью cookies

Несмотря на то что протокол HTTP — весьма мощный инструмент, и он имеет слабые места, одним из которых является отсутствие фиксации *состояний*. Это значит, что после завершения HTTP-обмена, ни клиент, ни сервер совершенно не помнят даже о самом факте

взаимодействия, не говоря уже о том, по какому поводу оно происходило. Подобная “забывчивость” может вызвать ряд проблем в Web-приложениях, поскольку сервер не может запоминать такие посланные клиенту важные сведения, как пароли, данные, позиции записей и т.д. Из-за отсутствия памяти особенно страдают приложения баз данных, поскольку они часто опираются на знания клиента о том, какая запись является текущей.

Протокол HTTP предоставляет базовый метод записи информации на компьютер клиента, позволяющий серверу получать информацию о клиенте в результате предыдущих операций обмена HTTP. Эта технология получила странное название *cookies* (на американском жаргоне — ловкач, проныра) и позволяет серверу записать информацию о состоянии в файл на жестком диске клиента, а затем вставить эту информацию в последующие HTTP-запросы данного клиента. Такой подход значительно увеличивает возможности сервера в части работы с динамическими Web-страницами.

Cookies представляет собой не что иное, как некоторое текстовое значение, записанное в формате `CookieName=CookieValue`. Значения cookies не должны включать точек с запятой или запятых. Следует иметь в виду, что пользователь может отказаться от использования файлов cookies, поэтому ни одно приложение не должно рассчитывать на обязательное их присутствие. Cookies получают все бóльшее распространение, поскольку Web-узлы становятся все более высокоорганизованными. Если вы — пользователь Netscape, вас может попросту удивить содержимое файла `COOKIES.TXT` (пользователи Internet Explorer могут заглянуть в папку `\WINDOWS\COOKIES`). Если вам необходимо отслеживать появление файлов cookies на вашем компьютере, установите соответствующий режим безопасности в свойствах используемого броузера.

Что же касается Delphi, то управлять этими cookies ему, как говорится, сам Бог велел. Классы `THTTPRequest` и `THTTPResponse` инкапсулируют их обработку, позволяя легко управлять как установкой новых значений cookies на компьютере клиента, так и считыванием ранее записанных значений.

Вся работа по установке cookies выполняется в методе `TWebResponse.SetCookieField()`. Этому методу можно передать параметр типа `TStrings`, содержащий значения cookies, а также ограничения на их применение.

Метод `SetCookieField` объявляется в модуле `HTTPAPP` следующим образом:

```
procedure SetCookieField(Values: TStrings; const ADomain, APath: string;
➥AExpires: TDateTime; ASecure: Boolean);
```

Параметр `Values` представляет собой потомок класса `TStrings` (можно также использовать потомок класса `TStringList`), который содержит реальные строковые значения cookies. В одном параметре `Values` можно передать несколько cookies.

С помощью параметра `ADomain` можно определить, к какому домену относятся данные cookies. Если значение домена не передано, то данный элемент cookies будет посылаться каждому серверу, к которому клиент делает запрос. Обычно Web-приложение устанавливает в этом параметре собственный домен, чтобы ему передавались только требуемые элементы cookies. Клиент будет оценивать значения существующих cookies и возвращать только те из них, которые соответствуют заданному критерию.

Например, если в параметре `ADomain` передать домен `widgets.com`, то все будущие запросы к серверам в домене `widgets.com` будут отправляться с включением значений cookies, записанных в результате сообщений, поступивших от этого домена. Другим доменам эти значения cookies передаваться не будут. Если клиент запрашивает домен `big.widgets.com` или `small.widgets.com`, то передача упомянутых cookies состоится в обоих случаях. Устанавливать значения cookies для домена могут лишь узлы, принадлежащие этому домену, что позволяет избежать возможных недоразумений.

С помощью параметра APath можно установить подмножество адресов URL внутри домена, в котором данный экземпляр cookie действителен. Параметр APath является подмножеством параметра ADomain. Если домен сервера совпадает с параметром ADomain, то проверяется параметр APath на совпадение с текущим путем запрашиваемого домена. Если параметр APath совпадает с информацией о пути в запросе клиента, cookie считается действительным.

Так (в продолжение предыдущего примера), если бы параметр APath имел значение /nuts, элементы cookies были бы действительны для запроса, направляемого по пути widgets.com/nuts, и для любых вложенных путей, таких как widgets.com/nuts/andbolts.

Параметр AExpires определяет, как долго элемент cookie должен оставаться действительным. В этом параметре допускается передавать любое значение типа TDateTime. Поскольку клиент может находиться в любой точке земного шара, это значение должно опираться на время по Гринвичу (GMT). Если требуется, чтобы элемент cookie оставался действительным в течение 10 дней, передайте в качестве параметра AExpires значение Now + 10.

Если требуется удалить некоторый элемент cookie, передайте в него значение уже прошедшей даты, что сделает его недействительным. При этом заметьте, что, хотя cookie сам по себе может быть недействительным и не передаваться, это совершенно не означает, что данные на самом деле удаляются из компьютера клиента.

Последний параметр — Asecure — представляет собой значение типа Boolean, которое определяет, может ли элемент cookie передаваться по незащищенным каналам. Значение True говорит о том, что данный элемент cookie может передаваться только с помощью средств защищенного HTML-протокола (HTTP-Secure protocol) или с использованием SSL. Для обычного применения этот параметр нужно установить равным False.

Серверное Web-приложение получает посланные клиентом элементы cookies в свойстве TWebRequest.CookieFields. Оно является потомком класса TStrings и хранит значения в виде индексированного массива. В строках содержатся полные значения элементов cookies, записанные в формате param=value. Доступ к ним осуществляется так же, как и к другим значениям типа TStrings. Кроме того, в свойстве TWebRequest.Cookie все поступившие элементы cookies передаются в виде одной строки, однако на практике этот параметр используется нечасто. С помощью метода TWebRequest.ExtractCookieFields() значения cookies можно также непосредственно присвоить уже существующему объекту типа TStrings.

Простоту работы с cookies в Delphi можно продемонстрировать на простом примере. Итак, создайте новое Web-приложение и добавьте в его инструкцию uses имя модуля WebUtils. Этот модуль можно найти на прилагаемом компакт-диске. Затем создайте новое приложение Web-сервера и включите в него два свойства Action — SetCookie и GetCookie. В обработчик события OnAction для SetCookie включите следующий код:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  List: TStringList;
begin
  List := TStringList.Create;
  try
    List.Add('LastVisit=' + FormatDateTime('mm/dd/yyyy hh:mm:ss', Now));
    Response.SetCookieField(List, '', '', Now + 10, False);
    Response.Content := 'Cookie set -- ' + Response.Cookies[0].Name;
  finally
    List.Free;
```

```
  end;
  Handled := True;
end;
```

Код обработчика события OnAction для GetCookie должен выглядеть так:

```
procedure TWebModule1.WebModule1WebActionItem2Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  Params: TParamsList;
begin
    Params := TParamsList.Create;
    try
      Params.AddParameters(Request.CookieFields);
      Response.Content := 'You last set the cookie on ' + Params['LastVisit'];
    finally
      Params.Free;
    end;
end;
```

Настройте Web-страницу на вызов следующих двух URL:

```
http://<ваш сервер>/project1.dll/SetCookie
http://<ваш сервер>/project1.dll/GetCookie
```

На заметку

Класс TParamList входит в состав модуля WebUtils, содержащегося на прилагаемом компакт-диске. Этот класс автоматически анализирует параметры потомков класса TStrings и позволяет индексировать их по имени. Например, класс TWebResponse собирает все элементы cookies, содержащиеся в поступившем HTTP-запросе, и помещает их в свойство CookieFields, которое является потомком класса TStrings. Cookies имеют формат CookieName=CookieValue. Класс TParamList производит анализ этих значений и индексирует их по имени. Таким образом, доступ к предшествующему параметру может быть осуществлен с помощью массива MyParams['CookieName'], и в результате будет получено значение CookieValue. Вы можете использовать либо этот класс, либо свойство Values класса TStrings, содержащегося в библиотеке VCL.

Установите cookie посредством вызова первого URL с Web-страницы, расположенной в одном каталоге с библиотекой DLL. Тем самым вы установите на компьютере клиента элемент cookie с именем LastVisit, который содержит дату и время, соответствующие сделанному запросу, и будет действителен в течение 10 дней. Если у вас есть Web-броузер, настроенный на прием cookies, он должен предложить вам подтвердить прием cookie от Web-узла. Затем вызовите действие GetCookie, чтобы прочитать только что созданный элемент cookie, и вы увидите дату и время, соответствующие последнему вызову действия SetCookie.

Элементы cookies могут содержать любую информацию, которую можно сохранить в строке. Размер одного элемента cookies может достигать 4 Кбайт, а один клиент может хранить до 300 элементов cookies. Отдельному серверу или домену разрешается размещать не более 20 элементов cookies. Как видите, возможности cookies небезграничны, а потому их нельзя использовать для хранения на компьютере клиента больших объемов данных.

Довольно часто требуется запомнить информацию, превышающую по объему возможности cookies. Иногда желательно зафиксировать предпочтения пользователя, адрес, личную информацию или даже содержимое корзины, обычно закупаемой им на вашем коммерческом электронном узле. Объем такой информации имеет свойство быстро возрастать, поэтому, вместо того чтобы хранить информацию о пользователе в чистом виде, лучше прибегнуть к ее кодированию. Например, для запоминания в cookies данных о предпочтениях конкретного пользователя, которые представляют собой значения типа Boolean, можно использовать двоичный формат. Предположим, значение cookies '1001' может означать, что пользователь хочет в дальнейшем обновлять содержимое своего электронного почтового ящика, возражает против передачи его адреса другим пользователям, не хочет, чтобы вы добавили его в список почтовой рассылки подписчиков системы LISTSERV, и желает присоединиться к вашим интерактивным дискуссионным группам. Для сокращения объемов информации, хранимой в cookies, подобный способ кодирования можно применить и к другим видам сведений о пользователе.

С помощью cookies можно также хранить идентификационное значение, которое однозначно определяет пользователя. Затем можно считать это значение из cookies и использовать его для поиска в базе данных информации именно об этом пользователе. Благодаря этому можно минимизировать объем данных, хранимых на компьютере пользователя, и максимизировать возможности управления информацией о пользователях.

Таким образом, cookies представляют собой мощный и в то же время простой способ поддержки данных о пользователях между отдельными HTTP-сеансами.

Перенаправление на другой Web-узел

Часто бывает, что некоторому URL нужно поменять место назначения запроса пользователя. Адресованное Web-приложение может обработать некоторые данные на основе запроса, а затем вернуть страницу, которая будет зависеть от природы запроса или некоторого элемента базы данных. Особенно часто это делается при публикации объявлений в Web. Так, рекламная графика указывает на другой URL, и по щелчку на таких изображениях пользователь переходит на начальную страницу рекламодателя. Часто HTML-код для графики рекламного характера содержит параметры рекламодателя, предназначенные для сервера. Сервер может зафиксировать эту информацию, а затем отправить клиента на соответствующую страницу. Этот метод, называемый *перенаправлением*, может весьма эффективно использоваться для решения многих задач.

Уже известный нам класс TWebResponse включает метод SendRedirect(), которому в качестве параметра передается одна строка. Эта строка должна содержать полный адрес узла, на который должен быть перенаправлен клиент. Метод SendRedirect объявляется следующим образом:

```
procedure SendRedirect(const URI: string); virtual; abstract;
```

Этот метод объявляется как абстрактный в модуле HTTPAPP.PAS и определяется в модуле ISAPIAPP.PAS.

Для Web-сервера не составит труда обработать HTTP-запрос с параметрами, а затем передать его на узел, имя которого указано в одном из параметров. Например, если на одной из страниц находится GIF-файл, а все графическое изображение представляет собой гиперссылку, то назначаемый ей URL будет выглядеть примерно так:

```
<A
HREF="http://www.someplace.com/transfer?www.borland.com&coolgif.gif&borland">
↳<IMG SRC="coolgif.gif"></A>
```

При этом обработчик события OnAction в приложении Web-сервера под именем /transfer, которому будет передана эта строка, может выглядеть следующим образом:

```
procedure TWebModule1.WebModule1WebActionItem3Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  { Обрабатываем параметр запроса Request.QueryFields[1], возможно размещая
    его в базе данных. Он содержит имя GIF-файла, на котором пользователь
    щелкнул мышью. Возможно, имеет смысл контролировать, какие из GIF-файлов
    являются самыми эффективными. Для этого достаточно зафиксировать,
    сколько раз происходил переход на узел каждой из компаний, разместивших
    рекламу на вашем узле. С этой целью можно использовать имя получающей
    вызов компании, поместив его в параметр Request.QueryFields[2]. }
  Response.SendRedirect(Request.QueryFields[0]);
end;
```

Используя этот метод, можно создать общее приложение передачи, которое будет обрабатывать каждое рекламное объявление на узле. Конечно, помимо рекламных, для вызова метода SendRedirect() могут быть и другие причины. Этот метод можно использовать и для фиксации любых специфических запросов URL и любых данных, которые могут быть связаны с определенными гиперссылками. При этом достаточно собрать данные из свойства QueryFields, а затем при необходимости вызвать метод SendRedirect().

Считывание информации из HTML-форм

С ростом популярности Internet и intranet растет и потребность в HTML-ориентированных формах. Поэтому нет ничего удивительного в том, что разработчики Delphi позаботились об упрощении процесса сбора информации из таких форм. В этой главе уделяется внимание не подробностям создания HTML-форм и соответствующим элементам управления, а обработке этих форм и содержащихся в них данных.

На прилагаемом компакт-диске содержится простое приложение ведения книги регистрации посетителей, предназначенное для сбора входных данных из HTML-форм и последующего заполнения базы данных на основе собранной информации. Предложенная для этой книги регистрации посетителей HTML-форма GUEST.HTM использует следующую строку для определения формы и действия, предпринимаемого при нажатии пользователем кнопки Submit:

```
<form method="post" action="guestbk.dll/form">
```

Этот код заставляет форму переслать ("post") свои данные и вызвать указанную функцию библиотеки DLL в качестве обработчика события OnAction. Эта форма предоставляет пользователю возможность ввести свое имя, адрес электронной почты, название города и комментарии. Когда пользователь щелкает на кнопке Submit, эта информация считывается и передается Web-приложению.

На Web-узле компонент класса Action с указанным именем (/form) получает эти данные в своем свойстве Request.ContentFields в виде стандартной строки HTML-параметров. Свойство ContentFields представляет собой класс, производный от класса TStrings, в котором хранится содержимое отправленной пользователем формы. Приложение содержит объект таблицы класса

TTable с именем GBTable, доступ к которой осуществляется с помощью псевдонима GBDATA. Необходимо создать этот псевдоним и связать его с каталогом /GBDATA, в котором содержатся таблицы в формате Paradox, используемые в приложении книги регистрации посетителей. В листинге 31.4 представлен код, который принимает содержимое формы и вводит его в базу данных.

Листинг 31.4. Код, предназначенный для приема содержимого из формы

```
var
  MyPage: TStringList;
  ParamsList: TParamsList;
begin
  begin
    ParamsList := TParamsList.Create;
    try try
      ParamsList.AddParameters(Request.ContentFields);
      GBTable.Open;
      GBTable.Append;
      GBTable.FieldByName('Name').Value := ParamsList['fullnameText'];
      GBTable.FieldByName('EMail').Value := ParamsList['emailText'];
      GBTable.FieldByName('WhereFrom').Value := ParamsList['wherefromText'];
      GBTable.FieldByName('Comments').Value :=
         ↳ParamsList['commentsTextArea'];
      GBTable.FieldByName('FirstTime').Value :=
         ↳(CompareStr(ParamsList['firstVisitCheck'], 'on') = 0);
      GBTable.FieldByName('DateTime').Value := Now;
      GBTable.Post;
    except
      Response.Content := 'An Error occurred in processing your data.';
      Handled := True;
    end;
    finally
      ParamsList.Free;
      GBTable.Close;
    end;
end;
```

Сначала в этом коде выполняется вставка свойства ContentFields в класс TParamsList. Затем открывается таблица GBTable, и данные из формы вставляются в соответствующие поля. Программный текст в листинге 31.4 относительно несложен и не требует дополнительных пояснений.

Во фрагменте кода, представленном в листинге 31.5, создается HTML-ответ, в котором пользователю выражается благодарность за внесение информации в базу данных. При этом используется взятое из формы имя адресата и подтверждается достоверность адреса его электронной почты.

Листинг 31.5. Код, предназначенный для создания HTML-ответа

```
MyPage := TStringList.Create;
ParamsList := TParamsList.Create;
  try
    with MyPage do
```

```
    begin
      Add('<HTML>');
      Add('<HEAD><TITLE>Guest Book Demo Page</TITLE></HEAD>');
      Add('<BODY>');
      Add('<H2>Delphi Guest Book Demo</H2><HR>');
      ParamsList.AddParameters(Request.ContentFields);
      Add('<H3>Hello <FONT COLOR="RED">'+ ParamsList['fullnameText']
        ➥+'</FONT> from '+ParamsList['wherefromText']+'!</H3><P>');
      Add('Thanks for visiting my homepage and making
        ➥an entry into my Guestbook.<P>');
      Add('If we need to e-mail you, we will use this address -- <B>'
        ➥+ParamsList['emailText']+'</B>');
      Add('<HR></BODY>');
      Add('</HTML>');
    end;
    PageProducer1.HtmlDoc := MyPage;
  finally
    MyPage.Free;
    ParamsList.Free;
  end;
Response.Content := PageProducer1.Content;
Handled := True;
```

И, наконец, это приложение предоставляет сводку по всем записям книги регистрации посетителей с именем /entries.

Формирование потоков данных

Большинство данных, которые будут передаваться клиентам в ответ на их HTTP-запросы, вероятнее всего, окажутся HTML-страницами. Но не исключено, что в некоторых случаях понадобится в ответ на запрос пользователя переслать ему и информацию некоторого другого типа. Это могут быть различные графические изображения или звуковые файлы, указанные в запросе пользователя. Данные, посылаемые пользователю, могут иметь специальный формат и поэтому должны быть специальным образом обработаны броузером клиента. Например, Netscape обеспечивает надстраиваемую архитектуру, которая позволяет разработчикам добавлять к Navigator Browser расширения, предназначенные для обработки данных любого типа. Примерами надстроек Netscape, расширяющих возможности броузера клиента, служат модули, поддерживающие RealAudio, Shockwave и другие типы потоковой организации данных.

Delphi существенно упрощает передачу потоков в адрес клиента, независимо от того, какой тип данных потребуется передать. С помощью метода `TWebResponse.SendStream` и свойства `TWebResponse.ContentStream` можно передать клиенту данные любого типа, загрузив их в потоковый класс Delphi. Безусловно, потребуется обязательно сообщить броузеру клиента сведения о типе посылаемых данных, что легко достигается с помощью свойства `TWebResponse.ContentType`. Установка этого строкового значения равным соответствующему MIME-типу, позволит броузеру надлежащим образом обработать приходящие данные. Например, если требуется передать Windows-файл с расширением .WAV, придется значение свойства `ContentType` установить равным `'audio/wav'`.

На заметку

MIME — это сокращение от названия стандарта Multipurpose Internet Mail Extensions (многоцелевое расширение почты Internet). Расширения MIME были разработаны для того, чтобы позволить клиентам и серверам передавать по электронной почте данные, которые по сложности отличаются от стандартной текстовой информации. Броузеры и HTTP-протокол адаптировали расширения MIME, благодаря чему появилась возможность передавать с Web-сервера в Web-броузер данные практически любого вида. Любой Web-броузер включает достаточно большой список этих типов MIME и каждое устанавливаемое приложение или надстройку связывает с определенным типом MIME. При получении информации конкретного типа броузер отыскивает приложение, которое должно использоваться для обработки данных этого типа, после чего передает ему эти данные.

С помощью потоков можно передавать данные практически любого типа из любого источника, доступного на Web-сервере. Можно передавать данные из файлов, которые размещаются на данном сервере или в любой точке локальной сети, а также из ресурсов Windows, встроенных в библиотеку ISAPI DLL или другие библиотеки DLL, доступные данной библиотеке ISAPI DLL. Можно даже формировать данные непосредственно в текущий момент и тут же отправлять их клиенту. Если броузер клиента "знает", что делать с принимаемыми данными, то, по сути, не существует ограничений ни в типах передаваемых данных, ни в способах их передачи.

Для иллюстрации всего вышеизложенного давайте создадим простое Web-приложение. Нам потребуется подготовить Web-страницу, содержащую изображения, полученные из различных источников. Приложение по запросу клиента будет обрабатывать графические данные и возвращать их ему в виде потока. Решить поставленную задачу совсем не сложно, поскольку Delphi предоставляет разработчику множество различных потоковых классов, весьма упрощающих доступ к данным. В то же время, классы расширений ISAPI превращают отправку данных пользователю в совершенно тривиальную задачу.

Для построения простого Web-приложения, иллюстрирующего использование потоков данных, выберите в главном меню команду File⇨New, а затем в раскрывшемся диалоговом окне New Items щелкните на пиктограмме Web Server Application. В результате будет создана заготовка модуля `TWebModule`. Перейдите в окно Web Module, выберите его, а затем перейдите в окно инспектора объектов. Дважды щелкните на свойстве `Actions` и создайте три действия с именами `/file`, `/bitmap` и `/resource`.

Выберите действие `/file`, перейдите в окно инспектора объектов и выберите вкладку Events. Создайте событие `OnAction`, а затем добавьте в обработчик этого события следующий код:

```
procedure TWebModule1.WebModule1WebActionItem2Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  FS: TFileStream;
begin
  FS := TFileStream.Create(JPEGFilename, fmOpenRead);
  try
    Response.ContentType := 'image/jpeg';
    Response.ContentStream := FS;
    Response.SendResponse;
    Handled := True;
  finally
    FS.Free;
  end;
end;
```

Приведенный код довольно прост. На прилагаемом компакт-диске в каталоге \bin содержится также JPEG-файл под именем TESTIMG.JPG. Обработчик события OnAction создает класс TFileStream, который загружает этот файл. Затем он устанавливает соответствующий тип MIME, чтобы сообщить броузеру клиента о передаче JPEG-файла, и присваивает результат создания экземпляра класса TFileStream свойству Response.ContentStream. Данные отправляются клиенту посредством вызова метода Response.SendResponse. В результате на полученной клиентом HTML-странице должно появиться изображение из указанного файла.

На заметку

В HTML-коде, предназначенном для отображения JPEG-файла в броузере, ссылку на свойство Action Web-приложения можно разместить непосредственно в дескрипторе IMG, как показано ниже.

```
<IMG SRC="../bin/streamex.dll/file" BOEDER=0>
```

Перейти на страницу с примерами передачи потоков можно со страницы INDEX.HTM, расположенной в каталоге \STREAMS, содержащемся на прилагаемом компакт-диске.

Это приложение может выполнять поиск JPEG-файла, поскольку при его создании значение переменной JPEGFilename устанавливается следующим образом:

```
procedure TWebModule1.WebModule1Create(Sender: TObject);
var
  Path: array[0..MAX_PATH - 1] of Char;
  PathStr: string;
begin
  SetString(PathStr, Path, GetModuleFileName(HInstance, Path, SizeOf(Path)));
  JPEGFilename := ExtractFilePath(PathStr) + 'TESTIMG.JPG';
end;
```

Действие /bitmap загрузит другое графическое изображение, причем совершенно иным способом. Код реализации этого действия несколько сложнее и имеет следующий вид:

```
procedure TWebModule1.WebModule1WebActionItem3Action(Sender: TObject; Request:
TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  BM: TBitmap;
  JPEGImage: TJPEGImage;
begin
  BM := TBitmap.Create;
  JPEGImage := TJPEGImage.Create;
  try
    BM.Handle := LoadBitmap(hInstance, 'ATHENA');
    JPEGImage.Assign(BM);
    Response.ContentStream := TMemoryStream.Create;
    JPEGImage.SaveToStream(Response.ContentStream);
    Response.ContentStream.Position := 0;
    Response.SendResponse;
    Handled := True;
  finally
    BM.Free;
    JPEGImage.Free;
  end;
end;
```

Преобразование растрового изображения в JPEG-файл с последующей передачей клиенту требует бо́льших усилий. Для захвата растрового изображения из файла ресурсов используется класс TBitmap, а для выполнения операции преобразования в JPEG-файл создается экземпляр класса TJPEGImage из модуля JPEG.

После создания класса TBitmap вызывается функция Windows API LoadBitmap, используемая для захвата растра из файла ресурсов под именем 'ATHENA'. Функция LoadBitmap возвращает дескриптор растра, который присваивается свойству Handle. Сам растр после этого сразу же назначается экземпляру класса TJPEGImage, а метод Assign благодаря перегрузке обладает достаточным интеллектом, чтобы “сообразить”, что от него требуется преобразовать растр в формат JPEG.

Последующие инструкции служат прекрасным примером полиморфизма. Свойство Response.ContentStream объявляется как абстрактный класс TStream. Благодаря полиморфизму, его можно определить с типом любого потомка класса TStream. В данном случае он создается как экземпляр класса TMemoryStream и используется для сохранения JPEG-файла, полученного с помощью вызова метода TJPEGImage.SaveToStream(). Теперь наш JPEG-файл находится в потоке и готов к пересылке. Очень важный, но часто забываемый шаг состоит в возвращении позиции потока на нуль после помещения в него JPEG-файла. Если этого не сделать, текущая позиция потока будет указывать на его конец, а значит, никакие данные клиенту отправлены не будут. Теперь можно вызвать метод Response.SendResponse(), который отправит клиенту данные, сохраненных в потоке.

Еще один способ загрузки JPEG-файла заключается в использовании файла ресурсов. Можно загрузить JPEG-данные в файл *.RES, поместив приведенный ниже код в RC-файл и откомпилировав его с помощью утилиты BRCC32.EXE. Если загружать данные как ресурс типа RCDATA, то можно воспользоваться классом TResourceStream, позволяющим легко загрузить данные и отправить их броузеру клиента. Класс TResourceStream очень мощный и способен загрузить ресурс как из EXE-файла данного приложения, так и из в любого внешнего файла библиотеки DLL. Как это делается, показано в подпрограмме действия /resource, где загрузка JPEG-данных выполняется из файла ресурсов под именем 'JPEG', скомпилированного в EXE-файл приложения:

```
procedure TWebModule1.WebModule1WebActionItem4Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.ContentStream := TResourceStream.Create(hInstance,
    ➥'JPEG', RT_RCDATA);
  Response.ContentType := 'image/jpeg';
  Response.SendResponse;
  Handled := True;
end;
```

В этой подпрограмме данные посылаются клиенту несколько иным способом. Этот вариант намного проще, но, тем не менее, также является прекрасным примером использования полиморфизма. Сначала создается экземпляр класса TResourceStream, который назначается свойству ContentStream. Поскольку конструктор класса TResourceStream сам загружает ресурс в поток, предпринимать никаких дальнейших действий по формированию потока не требуется. Для отправки данных потока клиенту остается только вызвать метод Response.SendResponse().

В заключительном примере клиенту отправляется WAV-файл, который хранится в виде RCDATA-ресурса. Здесь для отправки созданного потока используется метод Response.SendStream. Тем самым иллюстрируется еще один способ пересылки потоковых

данных. Он заключается в том, что создается поток, который при необходимости можно настроить или обработать, а затем отослать клиенту с помощью метода SendStream. В нашем случае это действие приведет к тому, что броузер воспроизведет WAV-файл с лаем собаки. Вот этот код:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  RS: TResourceStream;
begin
  RS := TResourceStream.Create(hInstance, 'BARK', RT_RCDATA);
  try
    Response.ContentType := 'audio/wav';
    Response.SendStream(RS);
    Handled := True;
  finally
    RS.Free;
  end;
end;
```

Резюме

В этой главе показано, как строить приложения Web-сервера с помощью расширений ISAPI/NSAPI. Приведенные сведения вполне применимы и для создания CGI-приложений в среде Delphi. Мы обсудили протокол HTTP и узнали, как Delphi инкапсулирует его в своих классах TWebRequest и TWebResponse. Было также показано, как строить приложения, используя класс TWebModule и динамически создавая HTML-код в его событиях OnAction. Затем были продемонстрированы варианты создания настраиваемых HTML-документов с помощью потомков класса TContentPageProducer и рассмотрены вопросы, связанные с доступом к данным и построением HTML-таблиц на базе класса TQueryTableProducer. Мы обсудили, как работать с элементами cookies и обрабатывать данные HTML-форм. Наконец, было продемонстрировано, как для передачи данных клиенту можно использовать потоки. В следующей главе мы познакомимся с многоуровневой технологией MIDAS.

Глава

32 Разработка приложений MIDAS

Дан Мизер (Dan Miser)

Механизм построения многоуровневого приложения 733

Преимущества многоуровневой архитектуры 734

Типичная архитектура приложения MIDAS 736

Использование технологии MIDAS для создания приложений 741

Дополнительные параметры, используемые для повышения устойчивости приложения 749

Примеры из реальной жизни 753

Дополнительные возможности наборов данных клиента 763

Установка MIDAS-приложений 772

Резюме 776

О многоуровневых приложениях сегодня говорят так же много, как и о любой другой области программирования. На это есть свои причины. Многоуровневые приложения имеют массу преимуществ перед традиционными приложениями клиент/сервер. Технология MIDAS (Middle-Tier Distributed Application Services — Средства разработки распределенных приложений среднего уровня) компании Borland предоставляет в распоряжение разработчика один из способов создания многоуровневых приложений, основанный на приемах и опыте, полученных при изучении Delphi. В этой главе вы познакомитесь с общей информацией о разработке многоуровневых приложений и увидите, как все рассмотренные принципы можно использовать для создания полнофункциональных приложений MIDAS.

Механизм построения многоуровневого приложения

Поскольку в этой главе речь пойдет о многоуровневых приложениях, вначале будет полезно разобраться в том, что же на самом деле представляет собой уровень. *Уровень* в данном случае является слоем приложения, обеспечивающим некоторый специфический набор функций. В приложениях баз данных можно выделить три основных уровня.

- *Данные.* Уровень данных отвечает за хранение данных. Как правило, на этом уровне используется система управления базой данных, например, Microsoft SQL Server, Oracle или InterBase.
- *Промежуточный (бизнес) уровень.* Этот уровень используется для получения информации с уровня данных в формате, соответствующем приложению, и выполнения окончательной проверки данных. Этот уровень также известен как уровень бизнес-правил, или уровень серверных приложений.
- *Уровень представления.* Этот уровень известен также как уровень графического интерфейса пользователя — именно на этом уровне осуществляется отображение данных в формате, отвечающем клиентскому приложению. Уровень представления всегда взаимодействует с уровнем бизнес-правил и никогда не связывается напрямую с уровнем данных.

В традиционных приложениях клиент/сервер используется архитектура, подобная представленной на рис. 32.1. Обратите внимание на то, что библиотеки доступа к данным клиента должны располагаться на каждой машине клиента. Исторически сложилось так, что этот момент всегда являлся камнем преткновения при установке приложений клиент/сервер — из-за несовместимости версий используемых библиотек DLL. Кроме того, поскольку большинство бизнес-правил реализовано в клиентской части приложения, при их обновлении требуется производить обновление для каждого клиента в отдельности.

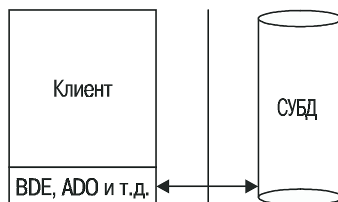


Рис. 32.1. Традиционная архитектура клиент/сервер

В многоуровневых приложениях используется архитектура, показанная на рис. 32.2. При использовании такой архитектуры появляется множество преимуществ по сравнению с эквивалентным приложением обычной архитектуры клиент/сервер.

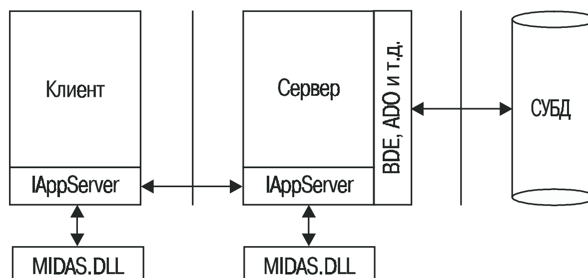


Рис. 32.2. Многоуровневая архитектура

Преимущества многоуровневой архитектуры

В следующих нескольких разделах будут рассмотрены основные преимущества многоуровневой архитектуры.

Централизованная поддержка бизнес-логики

В большинстве приложений клиент/сервер каждому клиентскому приложению для решения поставленной задачи требуется отслеживать собственные бизнес-правила. Это приводит не только к увеличению размера исполняемой части, но и вынуждает разработчика программного обеспечения осуществлять строгий контроль за совместимостью версий. Если пользователь А использует более раннюю версию приложения, чем пользователь В, то бизнес-правила могут выполняться несогласованно, вследствие чего могут возникнуть логические ошибки в данных. Реализация бизнес-правил на уровне серверного приложения требует создания и поддержки лишь одной его копии, поскольку любой пользователь, работающий с серверным приложением, будет использовать одну и ту же копию бизнес-правил. В приложениях клиент/сервер некоторые проблемы могут быть решены средствами СУБД, однако не все они обеспечивают одинаковые возможности. Кроме того, применение хранимых процедур делает приложения менее переносимыми. При использовании многоуровневого подхода бизнес-правила являются независимыми от СУБД, что упрощает решение задачи обеспечения независимости баз данных.

Архитектура „тонкого“ клиента

Помимо упомянутых бизнес-правил, типичное приложение клиент/сервер выполняет большинство функций уровня доступа к данным. Это приводит к увеличению размера клиентской выполняемой части приложения, получившей название “толстого” клиента. В частности, в случае приложений баз данных Delphi, предоставляющих доступ к SQL-серверу, на клиентской машине необходимо будет установить также BDE, драйверы SQL Links и/или ODBC и клиентские библиотеки, необходимые для взаимодействия с сервером. После установки все эти файлы

нужно соответствующим образом настроить. Все это существенно усложняет процесс установки приложения. При использовании технологии MIDAS доступ к данным контролируется серверной частью приложения, тогда как клиентской частью обеспечивается лишь представление данных. Это означает, что на клиентской машине достаточно установить лишь клиентское приложение и одну библиотеку DLL, обеспечивающую взаимодействие клиента с сервером. В этом и заключается особенность архитектуры “тонкого” клиента.

Автоматическое согласование ошибок доступа

В Delphi имеется встроенный механизм, облегчающий обработку ошибок доступа. В многоуровневых приложениях согласование ошибок доступа необходимо по тем же причинам, что и при использовании кэшированных обновлений. Данные копируются на машину клиента, где и осуществляется их изменение. Несколько клиентов могут одновременно работать с одной и той же записью. Механизм согласования ошибок доступа помогает пользователю определить, что следует сделать с записями, измененными со времени их последней загрузки с сервера. Согласно общей идеологии Delphi, если предоставляемые возможности вас не устраивают, их можно расширить или создать свои собственные правила.

Модель „портфеля“

Модель “портфеля” (briefcase model) основывается на концепции обычного портфеля. Вы можете поместить в него важные документы, переносить их с места на место и извлекать при необходимости. Delphi предоставляет возможность упаковывать все данные и сохранить их на диске переносного компьютера, что позволит работать с ними, находясь в пути, без реального соединения с серверной частью приложения или сервером базы данных.

Отказоустойчивость

Если компьютер, выполняющий функции сервера, станет в какой-то момент времени недоступным, то было бы неплохо иметь возможность динамически переключиться на резервный сервер без повторной компиляции клиентских и серверных приложений. Delphi обеспечивает и такую возможность.

Балансировка загрузки

При установке клиентского приложения на все возрастающем количестве пользовательских машин вы неизбежно столкнетесь с проблемой превышения предельно допустимой нагрузки на сервер. Существует два способа сбалансировать сетевой трафик: путем статической или динамической балансировки нагрузки. При статической балансировке нагрузки необходимо подключить еще одну серверную машину и переключить на нее половину пользователей, тогда как другая половина по-прежнему будет работать с исходным сервером. Однако при этом трудно обеспечить действительно равномерное распределение нагрузки между серверами, поскольку интенсивность работы с сервером разных групп пользователей различна. При использовании динамической балансировки нагрузки каждому клиентскому приложению можно указать конкретный сервер, к которому оно должно обращаться. Существует много различных алгоритмов динамической балансировки нагрузки (например, случайный, последовательный, с минимизацией пользователей сети и с минимизацией сетевого трафика). Начиная с версии 4, в Delphi содержится компонент, с помощью которого можно реализовать последовательную балансировку нагрузки серверов.

Классические ошибки

При создании многоуровневого приложения наиболее типичной ошибкой является размещение на уровне представления излишней информации об уровне данных. Как правило, проверки лучше производить на уровне представления, однако только способ выполнения такой проверки определяет удобство ее использования в многоуровневом приложении.

Например, при передаче динамических SQL-операторов от клиента серверу необходимо, чтобы клиентская часть была четко синхронизирована с уровнем данных. По этой причине возрастает количество перемещений данных, которые должны быть скоординированы во всем приложении. При изменении структуры одной таблицы на уровне данных потребуется обновить все клиентские приложения, в которых используются динамические SQL-операторы, чтобы они могли передавать на сервер корректные операторы. Это, естественно, ограничивает преимущества "тонких" клиентских приложений.

В качестве другого примера можно привести ситуацию, когда клиентское приложение пытается управлять временем жизни транзакции, вместо предоставления такой возможности приложению уровня бизнес-правил. Зачастую это реализуется с помощью публикации методов экземпляра класса `TDataBase` на сервере — `BeginTransaction()`, `Commit()`, `RollBack()` — и вызова их из клиентской части приложения. При этом код клиента становится более сложным для сопровождения, а главное, нарушается принцип, согласно которому лишь уровень представления должен отвечать за взаимодействие с уровнем данных. Никогда не следует полагаться на такой подход. Наоборот, обновления необходимо передавать на уровень бизнес-правил, который и будет производить обновление данных в транзакции.

Типичная архитектура приложения MIDAS

На рис. 32.3 показано, как выглядит типичное приложение MIDAS после его создания. Центральным элементом на диаграмме является модуль RDM (Remote Data Module — Удаленный модуль данных). RDM-модуль является потомком классического модуля данных, появившегося еще в версии Delphi 2. Подобный модуль данных представлял собой специальную форму, в которой можно было размещать только невидимые компоненты. Удаленный модуль данных в этом смысле ничем не отличается от своего предшественника. Кроме того, RDM-модуль является COM-объектом, или, если сказать более точно, объектом автоматизации. Службы, экспортируемые из RDM-модуля, будут доступны на всех клиентских машинах.

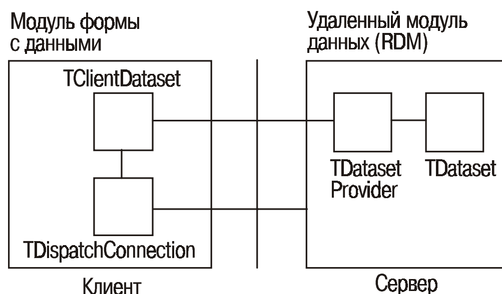


Рис. 32.3. Схема построения типичного приложения MIDAS

Рассмотрим некоторые параметры, доступные при создании модуля RDM. На рис. 32.4 показано диалоговое окно, появляющееся при выборе команды File⇨New⇨Remote Data Module.

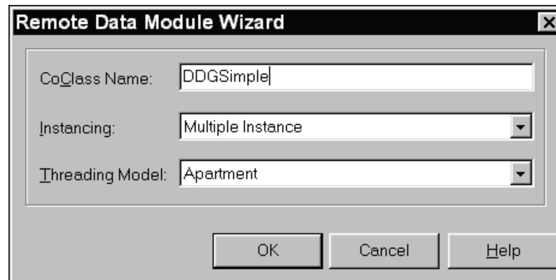


Рис. 32.4. Диалоговое окно мастера Remote Data Module

Сервер

Теперь, когда вы увидели, как выглядит схема типичного приложения MIDAS в целом, познакомимся с процессом его создания в Delphi. Сначала рассмотрим некоторые параметры, доступные при настройке сервера.

Способы создания экземпляров

Способ создания экземпляров определяет, сколько копий серверного процесса будет запущено. На рис. 32.5 показано, как выбранный параметр управляет поведением сервера.

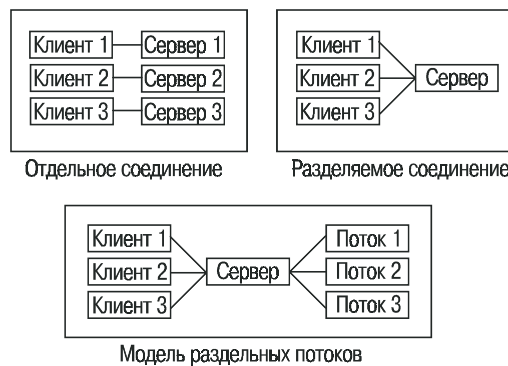


Рис. 32.5. Поведение сервера в зависимости от способа создания экземпляров

Для COM-сервера доступны следующие способы создания экземпляров удаленных модулей данных.

- `ciMultiInstance`. Каждый клиент, осуществляющий доступ к COM-серверу, использует один и тот же экземпляр сервера. По умолчанию это означает, что один клиент должен ожидать, пока предыдущий клиент не освободит COM-сервер. Для получения более подробной информации о том, как значение параметра Threading Choices влияет на поведение сервера в данном случае, читайте раздел “Выбор модели пото-

ков". Это решение эквивалентно последовательному доступу клиентов к серверу. Все клиенты должны использовать одно и то же соединение с базой данных, поэтому свойство `TDataBase.HandleShared` должно содержать значение `True`.

- `ciSingleInstance`. Каждый клиент, получая доступ к COM-серверу, использует отдельный экземпляр сервера. Это означает, что каждый клиент потребляет ресурсы отдельно загружаемого экземпляра сервера. Такой подход поддерживает параллельный доступ клиентов. Если вы решите выбрать этот параметр, не забывайте о существующих ограничениях ядра BDE, которые могут существенно снизить привлекательность данного варианта. Так, ядро BDE 5.01 позволяет создавать не более 48 процессов на одной машине. Поскольку каждый клиент порождает новый серверный процесс, то с каждым сервером одновременно установить соединение могут не более 48 клиентов.
- `ciInternal`. COM-сервер не может быть создан из внешних приложений. Этот подход оказывается полезным, если требуется управлять доступом к COM-объекту посредством промежуточного уровня защиты. Пример использования этого варианта можно найти по следующему адресу: `<`*`каталог DELPHI`*`>\DEMOS\MIDAS\POOLER`.

Обратите внимание также на то, что настройка DCOM-объекта напрямую влияет на режим создания экземпляров. Более подробно об этом рассказывается в разделе "Установка MIDAS-приложений".

Выбор модели потоков

Поддержка потоков в Delphi 5 претерпела коренные изменения. В версии 4 выбор модели потоков для EXE-сервера не имел особого смысла. В Registry просто устанавливался флажок, сообщающий подпрограммам поддержки COM, что функции данной библиотеки DLL могут выполняться с использованием указанной модели потоков. В Delphi 5 выбор модели потоков применим и для EXE-серверов, что позволяет подпрограммам поддержки COM распределять соединения по отдельным потокам без необходимости применения дополнительного кода. Для RDM-модулей допустимы следующие модели потоков.

- *Одиночная* (single). Выбор этой модели означает, что сервер одновременно может обрабатывать лишь один запрос. При использовании одиночной модели нет необходимости заботиться об управлении потоками, поскольку действия сервера ограничены одним потоком и подпрограммы COM осуществляют необходимую синхронизацию сообщений. Однако выбор такой модели окажется неудачным, если вы планируете создать многопользовательскую систему, поскольку клиент B, прежде чем начать свою работу, будет вынужден ожидать, пока не завершится обработка запроса клиента A. Этот подход чаще всего оказывается абсолютно неприемлемым, поскольку нет никаких гарантий, что клиент A не занят в данный момент формированием ежедневного отчета или не выполняет какие-либо другие операции, требующие значительных временных затрат.
- *Раздельная* (apartment). Выбор этой модели потоков дает самые высокие результаты в сочетании с присвоением параметру способа создания экземпляров значения `ciMultiInstance`. В этом случае благодаря параметру `ciMultiInstance` все клиенты будут совместно использовать один процесс сервера, однако при этом работа одного клиента с сервером не будет блокировать работу другого клиента, поскольку каждый из них будет работать с собственным потоком сервера. При выборе данной модели потоков гарантируется, что данные экземпляра RDM-модуля будут использоваться в безопасном режиме. Однако при этом необходимо защитить доступ к глобальным переменным, для чего можно воспользоваться любым из существующих способов синхронизации потоков — например, методом `PostMessage()`, критическими секциями, мьютексами, сема-

форами или классом-оболочкой Delphi TMultiReadExclusiveWriteSynchronizer. Именно эту модель потоков предпочтительнее использовать при работе с наборами данных BDE. Обратите внимание на то, что при использовании этой модели потоков наряду с наборами данных BDE вам потребуется поместить в RDM-модуль компонент TSession и установить значение его свойства AutoSessionName равным True. В этом случае BDE сможет адаптироваться к требованиям используемой модели потоков.

- *Свободная* (Free). Эта модель обеспечивает еще бо́льшую гибкость, позволяя передавать серверу несколько запросов клиентов. Однако при использовании этой модели необходимо обеспечить защиту от конфликтов потоков всех данных — как данных экземпляра, так и глобальных переменных. Эту модель предпочтительнее использовать совместно с объектами ADO компании Microsoft.
- *Обе модели* (Both). Эта модель также эффективна, как и свободная модель, но за одним исключением: при ее использовании обеспечивается автоматическая сериализация обратных вызовов.

Выбор способа доступа к данным

В Delphi 5 реализовано несколько различных способов доступа к данным. В частности, по-прежнему поддерживается BDE, благодаря чему можно использовать компоненты, производные от класса TDBDataSet, такие как TTable, TQuery и TStoredProc. Кроме того, теперь можно воспользоваться средствами технологии ADO и напрямую связываться с сервером InterBase посредством новых компонентов класса TDataSet.

Публикация служб

RDM-модуль отвечает за определение перечня служб, доступных клиенту. Если компонент TQuery удаленного модуля данных требуется сделать доступным клиенту, то наряду с ним в RDM необходимо поместить и компонент TDataSetProvider. Затем компонент TDataSetProvider связывается с компонентом TQuery через свойство TDataSetProvider.DataSet. Позже, когда клиенту потребуется использовать данные компонента TQuery, в этом ему поможет вновь созданный компонент TDataSetProvider. Чтобы указать, какие из провайдеров сервера должен видеть клиент, установите значение их свойства TDataSetProvider.Exported равным True или False.

В то же время, если клиенту не нужен весь набор данных сервера, а требуется лишь осуществить вызов метода, то можно обеспечить и такую возможность. Переместив фокус на удаленный модуль данных, выберите команду Edit⇨Add to Interface и введите в раскрывшемся диалоговом окне прототип стандартного метода. После обновления библиотеки типов можно будет определить реализацию этого метода — обычным способом.

Клиент

После построения сервера можно приступить к созданию клиента, который будет использовать службы, предоставляемые сервером. Рассмотрим параметры, доступные при построение клиента MIDAS.

Выбор соединения

Иерархия классов Delphi для соединения клиента с сервером начинается с класса TDispatchConnection. Этот базовый объект является родительским классом для всех типов соединений, которые будут рассмотрены ниже. В случае, если тип соединения не играет существенной роли, то мы будем ссылаться просто на объект TDispatchConnection.

Класс TDCOMConnection обеспечивает базовую защиту и аутентификацию, используя стандартную реализацию соответствующих служб в Windows. Этот тип соединения особенно полезен при использовании приложения в intranet/extranet (когда его применяют пользователи в пределах одного домена). В рамках технологии DCOM можно использовать раннее связывание, а также обратные вызовы и компоненты ConnectionPoints (обратные вызовы можно применять и при использовании сокетов, однако в этом случае вам придется ограничиться поздним связыванием). К недостаткам использования такого типа соединения можно отнести следующие.

- Во многих случаях усложняется настройка.
- Этот тип соединения плохо согласуется с концепцией брандмауэров.
- Для компьютеров под управлением Windows 95 требуется установка поддержки DCOM95.

Гораздо проще настроить соединение, предоставляемое компонентом TSocketConnection. Кроме того, оно использует лишь один порт для всего трафика MIDAS, поэтому задачи администратора брандмауэра будут гораздо проще, чем при работе с DCOM. Для поддержки такой схемы работы потребуется запустить программу ScktSrvr (находящуюся в каталоге <*каталог Delphi*>\BIN), т.е. для работы соединения на сервере потребуется лишь один дополнительный файл. В Delphi 4 также требовалось наличие пакета WinSock2, что для пользователей Windows 9x означает инсталляцию дополнительного программного обеспечения. Однако, если приложение работает в Delphi 5, причем обратные вызовы не используются, то можно установить значение свойства TSocketConnection.SupportCallbacks равным False. В этом случае на клиентских машинах достаточно использовать пакет WinSock1.

Компонент TOLEnterpriseConnection обеспечивает встроенные средства повышения устойчивости к сбоям и балансировки нагрузки. Кроме того, он упрощает использование в качестве сервера компьютера под управлением Windows 9x. Впервые компонент, обеспечивающий устойчивость к сбоям и простейшую балансировку загрузки (TSimpleObjectBroker), появился в Delphi 4. Теперь этот компонент допускает использование Windows 9x в качестве операционной системы сервера.

Начиная с версии Delphi 4 стало возможным использование компонента TCORBAConnection. Это эквивалент DCOM в рамках открытого стандарта. Технология CORBA применяется для адаптации приложений MIDAS к использованию межплатформных соединений. Например, клиент Java приложения MIDAS (поставляемый компанией Borland отдельно) позволяет клиенту JBuilder устанавливать соединение с сервером MIDAS даже в том случае, если он создан в Delphi.

В Delphi 5 появился компонент TWebConnection. Это компонент соединения позволяет поддерживать взаимодействие составных частей приложения MIDAS на основе протокола HTTP или HTTPS. На использовании этого типа соединения накладываются следующие ограничения.

- Не поддерживаются обратные вызовы любого типа.
- На машине клиента должна быть установлена библиотека WININET.DLL.
- На сервере должна быть установлена служба MS Internet Information Server (IIS) версии 4.0 (или броузер Netscape версии не ниже 3.6).

Однако эти ограничения являются оправданными, если приложение работает через Internet или должно проходить брандмауэр, который не находится под вашим управлением.

Помните, что использование всех этих транспортных компонентов подразумевает корректную установку в системе протокола TCP/IP. Существует лишь одно исключение, когда посредством DCOM связываются два компьютера под управлением Windows NT. В этом случае для определения типа используемого протокола DCOM нужно запустить утилиту DCOMCNFG, а затем переместить требуемый протокол в верхнюю часть списка, расположенного во вкладке Default Protocols. DCOM для Windows 9x поддерживает лишь протокол TCP/IP.

Соединение компонентов

Из рис. 32.3 видно, что составные части приложения MIDAS взаимодействуют через границы уровней. В этом разделе рассматриваются основные свойства и компоненты, благодаря которым клиент может взаимодействовать с сервером.

Для обеспечения взаимодействия клиента с сервером необходимо использовать один из потомков класса TDispatchConnection, рассмотренных в предыдущем разделе. У каждого компонента имеются свойства, специфические только для данного типа соединения, однако все они позволяют определить, где найти серверное приложение. Класс TDispatchConnection представляет собой аналог компонента TDatabase при использовании в приложениях клиент/сервер.

После установки соединения с сервером нужно выбрать способ использования служб, предоставляемых сервером. Для этого в клиентскую часть приложения следует поместить компонент TClientDataSet и связать его с компонентом TDispatchConnection. После этого в списке свойства ProviderNames можно будет просматривать перечень всех экспортируемых сервером провайдеров. Таким образом, в приложениях MIDAS компонент TClientDataSet функционально аналогичен компоненту TTable в приложениях клиент/сервер.

С помощью свойства TDispatchConnection.AppServer можно также вызывать пользовательские методы, существующие на сервере. Например, в следующей строке кода на сервере вызывается функция Login, которой передаются два строковых параметра, а возвращаемое значение имеет тип Boolean:

```
LoginSucceeded := DCOMConnection1.AppServer.Login(UserName, Password);
```

Использование технологии MIDAS для создания приложений

Теперь, ознакомившись со множеством параметров, доступных при разработке приложений с использованием технологии MIDAS, попробуем применить их на практике с целью создания реального приложения.

Установка сервера

Сначала познакомимся с механизмом построения серверного приложения. После этого мы обсудим, как создается клиентская часть приложения.

Удаленный модуль данных (RDM)

Основу серверного приложения составляет удаленный модуль данных. Чтобы создать RDM-модуль для нового приложения, выберите пиктограмму Remote Data Module во вкладке Multitier диалогового окна New Items (команда File⇨New). Появится диалоговое окно, в котором можно выполнить начальную настройку некоторых параметров удаленного модуля данных.

Самым важным параметром является имя RDM-модуля, поскольку идентификатор ProgID для данного приложения сервера будет построен с использованием имени проекта и имени удаленного модуля данных. Например, если проект (.DPR) имеет имя AppServer, а удаленный модуль данных — MyRDM, то идентификатор ProgID будет иметь вид AppServer.MyRDM. Убедитесь, что для сервера корректно выбран способ создания экземпляров и модель взаимодействия потоков.

Одним из важных изменений в Delphi 5 является модель обеспечения безопасности соединений, в которых используются протоколы TCP/IP и HTTP. Поскольку эти протоколы обходят процесс аутентификации, выполняемый в Windows по умолчанию, обязательно нужно удостовериться в том, что на сервере запускаются именно те объекты, которые вами определены. Для этого можно поместить в системный реестр определенные значения. Тем самым приложению MIDAS будут переданы сведения о том, какие именно объекты предполагается запускать. К счастью, все, что необходимо для этого сделать, так это переопределить метод класса UpdateRegistry. При создании удаленного модуля данных Delphi автоматически генерирует реализацию этого метода, показанную в листинге 32.1.

Листинг 32.1. Метод класса UpdateRegistry для удаленного модуля данных

```
class procedure TDDGSimple.UpdateRegistry(Register: Boolean;
 const ClassID, ProgID: string);
begin
  if Register then
  begin
    inherited UpdateRegistry(Register, ClassID, ProgID);
    EnableSocketTransport(ClassID);
    EnableWebTransport(ClassID);
  end
  else
  begin
    DisableSocketTransport(ClassID);
    DisableWebTransport(ClassID);
    inherited UpdateRegistry(Register, ClassID, ProgID);
  end;
end;
```

Этот метод вызывается каждый раз при регистрации или отмене регистрации сервера. Кроме записей в системном реестре, соответствующих COM-объектам и создаваемых при вызове наследованного метода UpdateRegistry, можно вызывать также методы Enable*XXX*Transport и Disable*XXX*Transport. При этом объект будет помечен как защищенный.

На заметку

Версия компонента TSocketConnection в Delphi 5 будет отображать в свойстве ServerName лишь зарегистрированные защищенные объекты. Если вам не требуется обеспечивать безопасность, сбросьте флажок Connections⇨Registered Objects Only в меню утилиты SCKTSRVR.

Провайдеры

Поскольку основной функцией сервера приложения является предоставление данных клиенту, нужно обеспечить возможность передачи данных с сервера в формате, понятном клиенту. К счастью, реализация технологии MIDAS в Delphi включает компонент TDataSetProvider, который существенно упрощает решение этой задачи.

Сначала поместите в удаленный модуль данных компонент TQuery. Если в приложении используется реляционная СУБД, то потребуется также и компонент TDatabase. Теперь нужно связать компоненты TQuery и TDatabase, а затем определить в свойстве SQL простой запрос — например, select * from customer. И, наконец, поместите в удаленный модуль данных компонент TDataSetProvider и свяжите его с компонентом TQuery через свойство DataSet. Свойство Exported объекта DataSetProvider определяет, будет ли этот провайдер видимым для клиентов. С помощью этого свойства можно легко управлять видимостью провайдеров и во время выполнения приложения.

На заметку

Хотя в данном разделе основное внимание уделяется компоненту TDBDataSet, ориентированному на работу с BDE, этими же принципами можно руководствоваться при организации доступа к данным с помощью любого другого компонента, производного от компонента TDataSet. В эту категорию входят также компоненты ADO и InterBase Express, рассмотрение которых выходит за рамки этой книги.

Регистрация сервера

После создания приложения сервера его нужно зарегистрировать как COM-объект, что сделает его доступным клиентским приложениям. Способы регистрации, обсуждавшиеся в главе 23, “COM-ориентированные технологии”, вполне применимы и в случае серверов MIDAS. Достаточно запустить приложение сервера — и требуемый параметр будет добавлен в системный реестр. Однако, прежде чем зарегистрировать сервер, необходимо сохранить проект. Это даст гарантию, что начиная с данного момента будет использоваться корректный идентификатор ProgID.

Если требуется запустить приложение лишь для регистрации, но не для реальной работы, то при запуске задайте в командной строке параметр /regserver. При этом будет выполнен процесс регистрации, после чего приложение немедленно завершится. Чтобы удалить параметры системного реестра, связанные с данным приложением, укажите в командной строке параметр /unregserver.

Создание клиента

Теперь, когда имеется работающее серверное приложение, рассмотрим основные задачи, выполняемые при создании клиента. Мы обсудим, как извлекать данные, как их редактировать, как обновить базу данных с учетом изменений, выполненных клиентом, и, наконец, как обработать ошибки, возникающие в процессе обновления.

Извлечение данных

В процессе работы приложения баз данных необходимо постоянно передавать данные от сервера клиенту с целью их редактирования. Помещение данных в локальный кэш снижает нагрузку на сеть и минимизирует время выполнения транзакций. В предыдущих версиях Delphi для решения этой задачи следовало использовать кэшированные обновления. В целом аналогичные решения применяются и в приложениях MIDAS.

Клиент взаимодействует с сервером посредством компонента TDispatchConnection. Можно легко решить эту задачу, поместив в компонент TDispatchConnection имя компьютера, на котором расположен сервер. При использовании компонента TDCOMConnection можно задать полностью квалифицированное имя домена (например, nt.dmiser.com), IP-адрес компьютера (например, 192.168.0.2) или его NetBIOS-имя (например, nt). В протоколе DCOM имеется ошибка, поэтому имя localhost использовать не рекомендуется. При использовании

компонента TSocketConnection в свойстве Address задаются IP-адреса, а в свойстве Host — полностью квалифицированное имя домена. О параметрах компонента TWebConnection мы поговорим немного позже.

После задания местоположения приложения сервера необходимо предоставить компоненту TDispatchConnection способ его идентификации. Это можно сделать с помощью свойства ServerName. При установке значения свойства ServerName автоматически заполняется свойство ServerGUID, которое является очень важным. В самом общем случае, если клиентское приложение подлежит распространению, следует удалить значение из свойства ServerName, оставив значение лишь в свойстве ServerGUID.

На заметку При использовании компонента TDCOMConnection в свойстве ServerName будет отображаться список серверов, зарегистрированных на *данном* компьютере. Однако компонент TSocketConnection является более "интеллектуальным" и отображает список серверов, зарегистрированных на удаленной машине.

Наконец, чтобы реально установить соединение с приложением сервера, установите значение свойства TDispatchConnection.Connected равным True.

Теперь, когда клиент соединен с сервером, необходимо определить способ использования провайдера сервера. Для этого воспользуйтесь компонентом TClientDataSet. Этот компонент применяется для связи с провайдером (а значит, и с компонентом TQuery, который тоже связан с провайдером) на сервере.

Прежде всего, необходимо связать компонент TClientDataSet с компонентом TDispatchConnection, установив соответствующее значение свойства RemoteServer компонента TClientDataSet. В результате в свойстве ProviderName будет содержаться список доступных провайдеров выбранного сервера.

С этого момента все готово для открытия набора данных клиента ClientDataSet.

Поскольку компонент TClientDataSet является виртуальным потомком компонента TDataSet, можно применять многие из приемов, рассмотренных нами при обсуждении использования компонента TDBDataSet в приложениях клиент/сервер. Например, установка значения свойства Active равным True приведет к открытию компонента TClientDataSet и отображению данных. Единственным отличием от установки свойства TTable.Active является то, что компонент TClientDataSet получает требуемые данные от приложения сервера.

Редактирование данных в клиентском приложении

Все записи, переданные сервером компоненту TClientDataSet, хранятся в его свойстве Data. Это свойство содержит представление пакета данных MIDAS с формате Variant. Компоненту TClientDataSet известно, как этот пакет данных преобразовать в более удобную форму. Использование типа Variant обусловлено тем, что для подсистемы COM доступно ограниченное число типов данных.

При выполнении операций с записями набора данных клиента копии вставленных, модифицированных или удаленных записей помещаются в свойство Delta. Этим обеспечивается высокая эффективность приложений MIDAS при передаче обновлений обратно в серверную часть приложения и, в конечном счете, в базу данных.

Для свойства Delta используется чрезвычайно эффективный формат. В этом свойстве для каждой операции вставки или удаления имеется одна запись, а для каждого обновления — две записи. Обновляемые записи хранятся тоже достаточно эффективно. Исходная запись содержится в первой записи, тогда как соответствующая ей модифицируемая запись хранится во второй записи обновления. Однако в модифицированной записи содержатся лишь измененные поля.

Интересно то, что свойство Delta совместимо со свойством Data. Другими словами, хранящееся в нем значение можно напрямую присвоить свойству Data другого компонента ClientDataSet. Благодаря этому текущее содержимое свойства Delta можно анализировать в любой момент.

Для редактирования данных можно использовать различные методы компонента TClientDataSet. В дальнейшем мы будем называться их *методами управления изменениями*. Методы управления изменениями позволяют различными способами модифицировать изменения, внесенные в набор данных клиента.

На заметку

Компонент TClientDataSet оказался гораздо более полезным, чем предполагалось изначально. Он обеспечивает прекрасный способ хранения таблиц в оперативной памяти (in-memory), что напрямую никак не связано с технологией MIDAS. Кроме того, поскольку компонент TClientDataSet может использоваться для передачи данных через свойство Data и другие свойства Delphi, он оказывается полезным и при реализации различных шаблонов объектно-ориентированного проектирования. Обсуждение этих приемов выходит за рамки излагаемого в этой главе материала. Однако для получения более подробной информации вы можете обратиться к Web-странице по адресу: http://xapware.com или http://xapware.com/ddg.

Отмена внесенных изменений

Большинство пользователей знакомы с текстовыми процессорами, в которых поддерживается команда отмены изменений. С ее помощью можно отменить последнее внесенное изменение и вернуться в исходное состояние. То же самое можно осуществить и с помощью вызова метода cds.Customer.UndoLastChange() компонента TClientDataSet. Используемый при этом стек имеет неограниченную длину, так что при необходимости можно вернуться к самому началу сеанса редактирования. Параметр, передаваемый этому методу, определяет, нужно ли позиционировать курсор на соответствующую запись.

Если требуется отменить все обновления одновременно, то последовательный вызов метода UndoLastChange() является не самым лучшим решением. Для отмены всех изменений, вносимых на протяжении одного сеанса редактирования, можно просто вызвать метод cdsCustomer.CancelUpdates().

Возврат к исходной версии

Для возвращения определенной записи в исходное состояние (т.е. в то состояние, в котором она находилась в момент первого извлечения из базы данных), можно воспользоваться другим методом. Для этого переместите курсор в наборе данных клиента на запись, которую нужно восстановить, и вызовите метод cdsCustomer.RevertRecord().

Транзакции клиента: свойство SavePoint

И, наконец, свойство SavePoint позволяет клиенту использовать транзакции. Это свойство идеально подходит для разработки сценариев типа “что-если”. Присвоив значение свойства SavePoint некоторой переменной, можно сохранить “моментальный снимок” данных. После этого пользователь может продолжать редактирование. Если в какой-либо момент пользователь решит, что в снимке содержатся именно те данные, которые ему требуются, можно присвоить свойству SavePoint сохраненное ранее значение. При этом набор данных клиента вернется в то состояние, в котором он находился до сохранения. Обратите внимание на то, что для реализации сложного сценария может понадобиться несколько уровней SavePoint.

Позволим себе одно предостережение относительно свойства SavePoint: при вызове метода UndoLastChange() значение этого свойства можно “испортить”. Например, предположим, пользователь выполнил редактирование двух записей и сохранил значение свойства SavePoint. Затем он внес изменения в следующую запись, после чего для отмены внесенных изменений дважды воспользовался методом UndoLastChange(). Поскольку компонент TClientDataSet перешел в состояние, предшествующее сохраненному в свойстве SavePoint, то значение этого свойства стало неопределенным.

Согласование данных

По завершении внесения изменений в локальную копию данных, содержащуюся в компоненте TClientDataSet, необходимо перенести эти изменения в базу данных. Это можно осуществить с помощью метода cdsCustomer.ApplyUpdates(). При вызове этого метода приложению сервера будет передано свойство Delta, после чего поступившие изменения будут внесены в базу данных в соответствии с механизмом согласования, установленным для обрабатываемого набора данных. Все обновления выполняются в рамках контекста одной транзакции. Кратко остановимся на обработке ошибок в ходе этого процесса.

Параметр, передаваемый методу ApplyUpdates(), определяет количество ошибок, появившихся в процессе обновления, после возникновения которых считается, что обновление завершилось неудачно. В этом случае все внесенные изменения последовательно отменяются. В данном контексте под *ошибками* подразумевается неудачный поиск по ключу, нарушение ссылочной целостности или любые другие ошибки базы данных. Если для этого параметра установлено нулевое значение, то тем самым задается недопустимость любых ошибок. Таким образом, при возникновении какой-либо ошибки, все внесенные изменения в базе данных зафиксированы не будут. Это значение используется чаще всего, поскольку оно наиболее точно соответствует основным принципам использования баз данных.

Однако при желании можно задать и ненулевое число допустимых ошибок. Тогда в базу данных будут перенесены все успешные записи. Предельным расширением этой концепции является передача в качестве параметра метода ApplyUpdates() значения -1. В этом случае приложением MIDAS в базе данных будет сохранена каждая отдельная запись, которую можно сохранить, без учета количества произошедших ошибок. Другими словами, при использовании этого значения транзакция всегда будет завершена успешно.

Если вам необходимо получить полный контроль над процессом обновления, в том числе возможность изменения операторов SQL, используемых для вставки, обновления или удаления, то можно воспользоваться обработчиком события TDataSetProvider.BeforeUpdateRecord(). Например, при удалении записи может потребоваться не удалять ее из базы данных. Вместо этого можно установить флажок, сообщающий приложениям, что данная запись недоступна. Позднее можно пересмотреть такие записи и выполнить операцию физического удаления. В следующем фрагменте кода демонстрируется, как это можно осуществить:

```
procedure TDataModule1.Provider1BeforeUpdateRecord(Sender: TObject;
  SourceDS: TDataSet; DeltaDS: TClientDataSet; UpdateKind: TUpdateKind;
  var Applied: Boolean);
begin
  if UpdateKind=ukDelete then
  begin
    Query1.SQL.Text:='update CUSTOMER set STATUS="DEL" where ID=:ID';
    Query1.Params[0].Value:=SourceDS.FieldByName('ID').Value;
    Query1.ExecSQL;
```

```
    Applied:=true;
  end;
end;
```

В процессе управления потоком и содержимым процесса обновления можно создать любое количество запросов, опираясь на различные факторы, например, на значение параметра `UpdateKind` и значения полей в наборе `DataSet`. При просмотре или модификации записей, содержащихся в параметре `DeltaDS`, удостоверьтесь, что используются свойства `OldValue` и `NewValue` соответствующего объекта `TField`. При использовании свойства `TField.Value` или `TField.As`*`XXX`* можно получить непредсказуемый результат.

Кроме того, при передаче обновлений в базу данных можно использовать бизнес-правила или вообще отменить передачу записи в базу данных. При генерировании исключительной ситуации она будет передана механизму обработки ошибок MIDAS, который рассматривается ниже.

По завершении транзакции можно проанализировать возникшие ошибки. При возникновении ошибки приостанавливается работа и сервера, и клиента. Вы получаете возможность устранить ошибки, зарегистрировать ошибку или выполнить любые другие необходимые действия.

Первая остановка осуществляется при обработке события `DataSetProvider.OnUpdateError`. Этот обработчик прекрасно подходит для обработки ожидаемых ошибок или для их устранения без вмешательства клиента.

Заключительным получателем ошибок является клиентское приложение, в котором пользователю можно дать возможность самому определить, как поступить с ошибочной записью. Для этого необходимо определить обработчик события `TClientDataSet.OnReconcileError`.

Этот подход оказывается исключительно полезным, поскольку MIDAS базируется на оптимистической стратегии блокировки записей. Данная стратегия позволяет нескольким пользователям одновременно работать с одной и той же записью. В общем случае это может быть причиной конфликтов, возникающих в том случае, если приложение MIDAS предпринимает попытку зафиксировать текущее состояние данных в базе, но обнаруживает, что со времени извлечения некоторая запись оказалась модифицированной. Ниже вы познакомитесь с определенными вариантами, альтернативными используемому по умолчанию процессу идентификации.

Использование стандартного диалогового окна согласования ошибок

К счастью, компания Borland предоставила в распоряжение разработчика стандартное диалоговое окно, которое можно использовать для отображения ошибок. Это окно показано на рис. 32.6. Имеется также и исходный код этого модуля, так что при необходимости его можно модифицировать. Для использования этого диалогового окна, выберите команду File⇨New, а затем — пиктограмму Reconcile Error Dialog во вкладке Dialogs. Не забудьте удалить этот модуль из списка Autocreate Forms, в противном случае при компиляции возникнут ошибки.

Почти все действия этого модуля реализованы в функции `HandleReconcileError()`. Эта функция тесно связана с событием `OnReconcileError`. На практике обработчик события `OnReconcileError` обычно вызывает функцию `HandleReconcileError`. Благодаря этому конечный пользователь на клиентской машине получает возможность взаимодействовать с процессом согласования ошибок на сервере и определять способ обработки этих ошибок. Обработчик события `OnReconcileError` может быть реализован следующим образом:

```
procedure TMyForm.CDSReconcileError(DataSet: TClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind;
```

```
  var Action: TReconcileAction);
begin
  Action:=HandleReconcileError(DataSet, UpdateKind, E);
end;
```

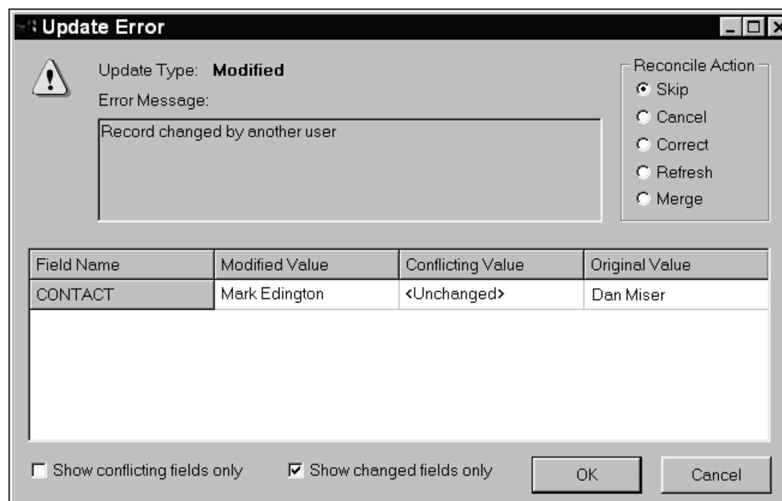


Рис. 32.6. Диалоговое окно Reconcile Error *в действии*

Значение параметра `Action` определяет действия, выполняемые приложением MIDAS над данной записью. Другие факторы, определяющие последовательность действий в этом методе, будут рассмотрены несколько позже. Перечень возможных действий содержится в следующем списке.

- `raSkip`. Не обновлять данную запись базы данных. Оставить измененную запись в буфере клиента.
- `raMerge`. Объединить поля записи с записью базы данных. Эта запись не будет применяться к уже вставленным записям.
- `raCorrect`. Обновить запись базы данных с использованием заданных значений. При выборе этого действия в диалоговом окне Reconcile Error значения можно редактировать в сетке. Этот метод нельзя применять, если запись базы данных редактировалась другим пользователем.
- `raCancel`. Не обновлять запись базы данных и удалить ее из буфера клиента.
- `raRefresh`. Обновить запись в буфере клиента с использованием данных текущей записи базы данных.
- `raAbort`. Полностью прервать операцию обновления.

Эти параметры имеют смысл (а, значит, и отображаются) не во всех случаях. Чтобы действия `raMerge` и `raRefresh` были доступны, запись должна была идентифицирована приложением с помощью первичного ключа базы данных. Это автоматически выполняется при использовании СУБД InterBase, однако в случае других реляционных СУБД для всех полей, указанных в первичном ключе, потребуется вручную установить равным `True` значение свойства `TField.ProviderFlags.pfInkey` компонента `TDataSet`.

Дополнительные параметры, используемые для повышения устойчивости приложения

После овладения основными принципами создания приложений MIDAS у читателя неизбежно возникнет вопрос: “Что дальше?” В этом разделе будут более подробно рассмотрены различные аспекты использования технологии MIDAS, а также новые средства управления работой приложения.

Методы оптимизации клиентской части приложения

Модель извлечения данных, применяемая в приложениях MIDAS, довольно изящна. Однако, поскольку все записи компонента TClientDataSet хранятся в оперативной памяти, необходимо проявлять осторожность при определении результирующих наборов данных, возвращаемых компоненту TClientDataSet. Обязательно следует удостовериться в том, что приложение сервера разработано правильно и возвращает лишь требуемые записи. Для ограничения количества записей, извлекаемых в каждый момент времени, можно использовать описанный ниже прием.

Ограничение размеров пакета данных

При открытии компонента TClientDataSet сервер извлекает из источника данных количество записей, указанное в свойстве TClientDataSet.PacketRecords. Однако приложение MIDAS исходно выбирает столько записей, сколько потребуется, чтобы их данными заполнить все доступные визуальные элементы управления. Например, если компонент TDBGrid в форме отображает одновременно до 10 записей, а значение свойства PacketRecords установлено равным 5, то в начальной выборке данных будет содержаться 10 записей. Во всех последующих пакетах данных будет содержаться по 5 записей. Если значение этого свойства равно -1, то передаются все записи. Если свойство PacketRecords принимает положительное значение, то оно определяет состояние приложения. Это обусловлено тем, что приложение сервера должно отслеживать перемещение курсора каждого клиента, чтобы сервер в ответ на запрос клиента мог вернуть соответствующий пакет записей. Однако состояние клиента можно отслеживать самостоятельно, передавая серверу позицию его последней записи. Именно это и осуществляется в следующем фрагменте кода:

```
Server RDM:
procedure TStateless.DataSetProvider1BeforeGetRecords(Sender: TObject;
  var OwnerData: OleVariant);
begin
  with Sender as TDataSetProvider do
  begin
    DataSet.Open;
    if not VarIsEmpty(OwnerData) then
      DataSet.Locate('au_id', OwnerData, []) else
    DataSet.First;
```

```
  end;
end;

procedure TStateless.DataSetProvider1AfterGetRecords(Sender: TObject;
  var OwnerData: OleVariant);
begin
  with Sender as TDataSetProvider do
  begin
    OwnerData := DataSet.FieldValues['au_id'];
    DataSet.Close;
  end;
end;

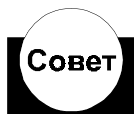
Client:
procedure TForm1.ClientDataSet1BeforeGetRecords(Sender: TObject;
  var OwnerData: OleVariant);
begin
  // KeyValue - закрытая переменная типа OleVariant
  if not (Sender as TClientDataSet).Active then
    KeyValue := Unassigned;
  OwnerData := KeyValue;
end;

procedure TForm1.ClientDataSet1AfterGetRecords(Sender: TObject;
  var OwnerData: OleVariant);
begin
  KeyValue := OwnerData;
end;
```

При использовании автоматической выборки метод TCLientDataSet.Last() извлекает остальные записи результирующего набора данных. Для получения этого же результата можно нажать в компоненте TDBGrid комбинацию клавиш <Ctrl+End>. Чтобы обойти данную проблему, установите значение свойства TCLientDataSet.FetchOnDemand равным False. Это свойство определяет, будет ли пакет данных извлекаться автоматически, если пользователь прочел все существующие записи в клиентском приложении. Если эту же возможность необходимо реализовать в коде, воспользуйтесь методом GetNextPacket(), возвращающим следующий пакет данных.

Использование модели „портфеля"

Другим видом оптимизации, позволяющим уменьшить трафик в сети, является использование модели "портфеля". Для этого присвойте свойству TClientDataSet.Filename имя некоторого файла. Если заданный файл уже существует, то компонент TCLientDataSet будет открывать локальную копию файла, а не считывать данные непосредственно из приложения сервера. Это чрезвычайно удобно для редко изменяемых элементов, таких как справочные таблицы.



Если в свойстве TClientDataSet.Filename задано имя файла с расширением .XML, то пакет данных будет сохранен в XML-формате. При этом для его обработки можно будет использовать любые инструменты XML, доступные при работе с файлом портфеля.

Передача на сервер динамических SQL-команд

В некоторых случаях может потребоваться модификация со стороны клиента основных свойств компонента TDataSet (например, свойства SQL компонента TQuery). Как следует из принципов создания многоуровневых приложений, это может оказаться достаточно эффективным и элегантным решением. В Delphi 5 решение этой задачи значительно упрощается.

Для формирования специальных запросов необходимо выполнить два действия. Во-первых, поместите требуемую SQL-команду в свойство TCLientDataSet.CommandText. Во-вторых, установите значение свойства DataSetProvider.Options равным poAllowCommandText. При открытии компонента TClientDataSet или вызове метода TClientDataSet.Execute() команда из свойства CommandText будет передана серверу. Этот же прием можно использовать для изменения имени таблицы или хранимой процедуры на сервере.

Методы оптимизации серверного приложения

При разработке приложений MIDAS поведение системы можно настраивать с помощью многочисленных событий. Практически для любого метода существуют события BeforeXXX и AfterXXX. Эти события могут оказаться особенно полезными, если необходимо придать приложению сервера полную независимость от состояния процессов пользователей.

Согласование разногласий записей

В предыдущих разделах при обсуждении механизма согласования разногласий кратко упоминалось о том, что если два пользователя работают с одной и той же записью, то при попытке второго пользователя передать запись на сервер будет сгенерирована ошибка. К счастью, можно осуществлять полный контроль над возникновением таких разногласий.

Свойство TDataSetProvider.UpdateMode применяется для генерации оператора SQL, который будет использоваться для проверки того, изменилась ли запись с момента ее последнего извлечения из базы данных. Вернемся к сценарию, в котором два пользователя редактируют одну и ту же запись. Ниже приведены возможные значения свойства TDataSetProvider.UpdateMode, определяющие возможности работы с записью каждого пользователя.

- upWhereAll. Этот параметр является самым ограничительным, однако обеспечивает наиболее высокую степень уверенности в том, что запись не изменилась со времени ее первоначального извлечения из базы данных. Если два пользователя редактируют одну и ту же запись, то первый пользователь может эту запись обновлять, тогда как второй пользователь получит сообщение об ошибке "Another user changed the record" ("Запись изменена другим пользователем"). Если в дальнейшем понадобится уточнить, какие поля следует использовать для данной проверки, удалите значение pfInWhere из соответствующих свойств TField.ProviderFlags.
- upWhereChanged. Если установлен этот параметр, то два пользователя смогут одновременно редактировать одну и ту же запись. Если оба пользователя редактируют различные поля одной и той же записи, то это не приведет к конфликту. Например, если пользователь A модифицирует поле Address и обновляет запись, то пользователь B, в свою очередь, может отредактировать поле BirthDate, а затем успешно обновить эту же запись.
- upWhereKeyOnly. Этот параметр является самым "нетребовательным". Запись базы данных может изменить каждый из пользователей. При этом существующая запись всегда замещается новой. Таким образом, при использовании этого параметра реализуется принцип "выигрывает последний".

Прочие параметры сервера

Для того чтобы определить, как приложение MIDAS будет управлять пакетами данных, можно использовать и другие допустимые значения свойства TDataSetProvider.Options. Например, при добавлении значения poReadOnly набор данных для клиента будет доступен только для чтения. Если указано значение poDisableInserts, poDisableDeletes или poDisableEdits, то клиент не сможет выполнить операцию вставки, удаления или редактирования соответственно, а в случае попытки выполнить запрещенную операцию, будут активизированы соответствующие обработчики событий OnEditError или OnDeleteError.

При использовании вложенных наборов данных из главной записи можно выполнять каскадные обновления или удаления в детальных записях, если добавить в свойство DataSetProvider.Options значения poCascadeUpdates или poCascadeDeletes. При использовании этого свойства необходимо, чтобы базовая СУБД поддерживала каскадную проверку ссылочной целостности.

Недостатком предыдущих версий технологии MIDAS была невозможность простого перенесения изменений, сделанных на сервере, в компонент TCLientDataSet клиентского приложения. Можно было лишь сортировать записи с помощью метода RefreshRecord (или в некоторых случаях повторно заполнить весь набор данных).

При добавлении в свойство DataSetProvider.Options значения poPropogateChanges все изменения данных, внесенные на сервере (например, в обработчике события DataSetProvider.BeforeUpdateRecord, предназначеном для реализации бизнес-правил), будут автоматически переданы обратно компоненту TClientDataSet. Более того, если в свойство DataSetProvider.Options добавить значение poAutoRefresh, то компоненту TClientDataSet будет автоматически передано значение свойства AutoIncrement, а также значения, используемые по умолчанию.

Режим poAutoRefresh в исходной версии Delphi 5 не работает. В более поздних версиях эта ошибка была устранена. Чтобы обойти эту проблему, необходимо либо воспользоваться методом Refresh() компонента TClientDataSet, либо самостоятельно управлять всем процессом обновления данных.

До сих пор при обсуждении процесса согласования противоречий речь шла о выполнении согласования стандартными средствами SQL, а события компонента TDataSet в процессе согласования не использовались. Для поддержки использования этих событий в процессе согласования конфликтов данных было создано свойство TDataSetProvider.ResolveToDataSet. Например, если значение этого свойства установлено равным True, то можно использовать большинство событий компонента TDataSet. Следите за тем, чтобы все используемые события вызывались лишь во время передачи обновлений обратно на сервер. Другими словами, если на сервере определено событие TQuery.BeforeInsert, то оно должно генерироваться только при вызове метода TClientDataSet.ApplyUpdates. События сервера нельзя объединить с соответствующими событиями компонента TClientDataSet.

Поддержка связи “главная-детальная”

Знакомство с приложениями баз данных оказалось бы неполным без упоминания о связях между таблицами типа “главная–детальная”. При использовании технологии MIDAS эту связь можно реализовать двумя способами. Один из приемов состоит в том, чтобы с сервера экспортировать два провайдера, а связь “главная–детальная” создать на стороне клиента. При этом в свойстве cdsDetail.PacketRecords по умолчанию будет содержаться нулевое значе-

ние. Не изменяйте это значение, поскольку в этом контексте оно используется для извлечения всех детальных записей для текущей главной записи. Недостатком создания связи "главная–детальная" на стороне клиента является невозможность обновления главного и детального наборов данных в контексте одной транзакции. Это серьезная проблема, однако, к счастью, данное ограничение можно обойти с помощью простого в использовании модуля, о котором речь пойдет несколько позже.

Вложенные наборы данных

Вложенные наборы данных впервые появились в Delphi 4. Вложенные наборы данных позволяют помещать детальные наборы данных непосредственно в главную таблицу. Помимо того, что это позволяет выполнять обновление главной и детальных записей за одну транзакцию, стало возможным хранить все главные и детальные записи в одном файле портфеля. Можно также использовать все преимущества компонента `DBGrid`, который теперь позволяет отображать детальные наборы данных в отдельных окнах. При использовании вложенных наборов данных не забывайте о следующем: при выборке главной записи извлекаются и все связанные с ней детальные записи, которые затем передаются клиенту. При использовании нескольких уровней вложенности детальных наборов данных это может послужить причиной существенного снижения производительности. Например, при извлечении одной главной записи, имеющей 10 детальных, каждая из которых связана, в свою очередь, с тремя детальными записями следующего уровня, из базы данных будет получена 41 запись. При использовании связи со стороны клиента сначала будут получены только 14 записей, а остальные вложенные записи будут извлекаться по мере перемещения по детальному клиентскому набору данных. Более подробно вложенные наборы данных будут рассматриваться ниже в этой главе.

Примеры из реальной жизни

Теперь, изучив основные принципы построения приложений MIDAS, ознакомимся, как можно применить эту технологию для решения практических задач.

Объединения

Существенное влияние на построение приложений реляционных баз данных оказывают способы организации связей между таблицами. Зачастую более удобно работать с представлениями, созданными на базе сильно нормализованных данных. В этом случае представления существенно сглаживают и упрощают представление данных, по сравнению с их реальною структурой. Однако при обновлении данных в подобных объединениях необходимо соблюдать дополнительные меры предосторожности.

Обновление одной таблицы

Применение обновлений к объединенному запросу представляет собой специальный случай в программировании баз данных, и приложения MIDAS не являются исключением. Проблема заключается в самом объединенном запросе. Хотя некоторые объединенные запросы предоставляют данные, которые могут быть обновлены автоматически, тем не менее существуют запросы, не допускающие автоматического извлечения, редактирования и обновления исходных данных. В Delphi задача разрешения обновлений объединенных запросов ложится на плечи разработчика.

При использовании объединений, требующих обновления лишь одной таблицы, большинство нужных действий Delphi может выполнить самостоятельно. При записи в базу данных информации из одной таблицы необходимо выполнить следующие действия.

1. Добавьте постоянные (persistent) поля в объединенный компонент TQuery.
2. Для каждого поля компонента TQuery, которое необходимо обновить, установите TField.ProviderFlags=[].
3. Чтобы сообщить приложению MIDAS, какую таблицу требуется обновить, поместите в обработчик события DataSetProvider.OnGetTableName код, приведенный ниже. Это новое событие упрощает способ задания имени таблицы, хотя в Delphi 4 то же самое можно было выполнить с помощью события DataSetProvider.OnGetDataSetProperties:

```
procedure TJoin1Server.prvJoinGetTableName(Sender: TObject;
  DataSet: TDataSet; var TableName: String);
begin
  TableName := 'Emp';
end;
```

Выполнив эти действия, вы сохраните имя таблицы в компоненте ClientDataSet. Теперь при вызове метода ClientDataSet1.ApplyUpdates() приложению MIDAS будет известно имя таблицы, используемое по умолчанию, и ему не придется выполнять его поиск.

Можно воспользоваться и другим способом — компонентом TUpdateSQL, выполняющим обновление лишь требуемой таблицы. Благодаря этой возможности в Delphi 5 во время процесса согласования можно использовать метод TQuery.UpdateObject. Такой подход наиболее точно соответствует процессу, применяемому в традиционных приложениях клиент/сервер.

Пример использования такого подхода можно найти на прилагаемом компакт-диске — в подкаталоге \Join1 каталога, соответствующего данной главе.

Обновление нескольких таблиц

В более сложных сценариях, когда редактируется и обновляется несколько таблиц, вам придется написать свой собственный код. При решении этой проблемы можно использовать два подхода.

- Подход, предложенный в Delphi 4, состоит в использовании метода DataSetProvider.BeforeUpdateRecord() для разбиения пакета данных и применения обновлений к базовым таблицам.
- Подход, предложенный в Delphi 5, состоит во внесении обновлений с помощью свойства UpdateObject.

Если для объединения нескольких таблиц используются кэшированные обновления, то для каждой обновляемой таблицы потребуется настроить один компонент TUpdateSQL. Поскольку свойство UpdateObject может быть присвоено только одному компоненту TUpdateSQL, то все свойства TUpdateSQL.DataSet необходимо связать с объединенным набором данных программно, в TQuery.OnUpdateRecord. После этого нужно вызвать метод TUpdateSQL.Apply, чтобы связать все параметры и выполнить исходный SQL-оператор. В рассматриваемом случае набор данных содержится в свойстве Delta. Этот набор данных передается в качестве параметра в обработчик события TQuery.OnUpdateRecord.

Однако при попытке реализации этого подхода в приложении MIDAS сразу же возникает одна проблема. Свойство TUpdateSQL.DataSet имеет тип TDBDataSet. Поскольку набор данных, содержащийся в свойстве Delta, имеет тип TDataSet, то присваивание является некор-

ректным. Мы не станем отказываться от выбранного подхода и обращаться для применения обновлений к методу Provider.BeforeUpdateRecord, а воспользуемся потомком компонента TUpdateSQL, который работает точно так же. Ключом к созданию этого компонента является переобъявление свойства DataSet с типом TDataSet и статическое переопределение метода SetParams для связывания параметров с целевым набором данных TDataSet. Кроме того, организуется предоставление свойств SessionName и DatabaseName с целью разрешить выполнение обновления в том же контексте, что и в случае других транзакций. Результирующий код обработчика события TQuery.OnUpdateRecord содержится в листинге 32.2.

Листинг 32.2. Объединение с использованием компонента TUpdateSQL

```
procedure TJoin2Server.JoinQueryUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  usqlEmp.SessionName := JoinQuery.SessionName;
  usqlEmp.DatabaseName := JoinQuery.DatabaseName;
  usqlEmp.Dataset := Dataset;
  usqlEmp.Apply(UpdateKind);

  usqlFTEmp.SessionName := JoinQuery.SessionName;
  usqlFTEmp.DatabaseName := JoinQuery.DatabaseName;
  usqlFTEmp.Dataset := Dataset;
  usqlFTEmp.Apply(UpdateKind);

  UpdateAction := uaApplied;
end;
```

Поскольку мы действовали в соответствии с правилами обновления данных, используемыми в технологии MIDAS, весь процесс обновления в точности соответствует вызову метода MIDAS ClientDataSet1.ApplyUpdates(0);.

На заметку

В Delphi 5 при согласовании ошибок доступа теперь может использоваться свойство UpdateObject. Вполне логично предположить, что метод применения обновлений к многотабличным объединениям, предназначенный для кэшированных обновлений, будет доступен и в приложениях MIDAS. Однако, во время написания этой книги такая возможность отсутствовала.

Этот пример можно найти на прилагаемом компакт-диске — в подкаталоге \Join2 каталога, соответствующего данной главе.

Приложения MIDAS и Web

Delphi тесно связана с платформой Windows. Поэтому любые клиентские приложения должны запускаться на компьютере под управлением Windows. Однако это не всегда желательно. Например, может понадобиться обеспечить простой доступ к базе данных через Internet. Поскольку ранее было создано приложение сервера, которое, помимо того что обеспечивает бизнес-правила, функционирует и в качестве брокера, очень желательно было бы использовать его в этом случае повторно, а не заново переписывать уровень бизнес-правил и доступа к данным для другой среды выполнения.

Простой HTML

Этот раздел посвящен вопросам адаптации приложения сервера и создания нового уровня представления, на котором используется простой язык HTML. Прежде чем приступить к изучению этого раздела, необходимо ознакомиться с материалом главы 31, “Компоненты WebBroker открывают двери в Internet”. При использовании такого подхода в архитектуру приложения вводится еще один уровень. По отношению к приложению сервера модуль WebBroker функционирует как клиент, преобразующий данные в формат HTML с целью отображения их в окне броузера. При этом теряются некоторые преимущества работы в интегрированной среде разработки Delphi, например, такие как возможность использования многочисленных управляющих элементов. Однако это совершенно необходимо, если требуется предоставить доступ к данным в простом HTML-формате.

Создав модуль `WebModule`, просто поместите в эту форму компоненты `TDispatchConnection` и `TClientDataSet`. После заполнения требуемых свойств можно использовать различные методы преобразования предоставляемых клиенту данных в формат HTML.

Одним из эффективных приемов является добавление компонента `TDataSetTableProducer`, связанного с компонентом `TClientDataSet`. В этом случае пользователь сможет щелкнуть на ссылке и перейти на страницу редактирования, на которой можно будет модифицировать и обновлять данные. В листингах 32.3 и 32.4 представлен пример реализации этого приема.

Листинг 32.3. Код HTML для редактирования и применения обновлений

```
<form action="<#SCRIPTNAME>/updaterecord" method="post">
<b>EmpNo: <#EMPNO></b>
<input type="hidden" name="EmpNo" value=<#EMPNO>>
<table cellspacing="2" cellpadding="2" border="0">
<tr>
    <td>Last Name:</td>
         <td><input type="text" name="LastName" value=<#LASTNAME>></td>
</tr>
<tr>
    <td>First Name:</td>
         <td><input type="text" name="FirstName" value=<#FIRSTNAME>></td>
</tr>
<tr>
<td>Hire Date:</td>
         <td><input type="text" name="HireDate" size="8" value=<#HIREDATE>></td>
</tr>
<tr>
    <td>Salary:</td>
    <td><input type="text" name="Salary" size="8" value=<#SALARY>></td>
</tr>
<tr>
    <td>Vacation:</td>
    <td><input type="text" name="Vacation" size="4" value=<#VACATION>></td>
</tr>
</table>
<input type="submit" name="Submit" value="Apply Updates">
<input type="Reset">
</form>
```

Листинг 32.4. Код для редактирования и внесения обновлений

```
unit WebMain;

interface

uses
  Windows, Messages, SysUtils, Classes, HTTPApp, DBWeb, Db, DBClient,
  MConnect, DSProd;

type
  TWebModule1 = class(TWebModule)
    dcJoin: TDCOMConnection;
    cdsJoin: TClientDataSet;
    dstpJoin: TDataSetTableProducer;
    dsppJoin: TDataSetPageProducer;
    ppSuccess: TPageProducer;
    ppError: TPageProducer;
    procedure WebModuleBeforeDispatch(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
    procedure WebModule1waListAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure dstpJoinFormatCell(Sender: TObject; CellRow,
      CellColumn: Integer; var BgColor: THTMLBgColor;
      var Align: THTMLAlign; var VAlign: THTMLVAlign; var CustomAttrs,
      CellData: String);
    procedure WebModule1waEditAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure dsppJoinHTMLTag(Sender: TObject; Tag: TTag;
      const TagString: String; TagParams: TStrings;
      var ReplaceText: String);
    procedure WebModule1waUpdateAction(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
  private
    { Закрытые объявления }
    DataFields : TStrings;
  public
    { Открытые объявления }
  end;

var
  WebModule1: TWebModule1;

implementation

{$R *.DFM}

procedure TWebModule1.WebModuleBeforeDispatch(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
```

```
  with Request do
    case MethodType of
      mtPost: DataFields:=ContentFields;
      mtGet: DataFields:=QueryFields;
    end;
end;

function LocalServerPath(sFile : string = '') : string;
var
  FN: array[0..MAX_PATH- 1] of char;
  sPath : shortstring;
begin
  SetString(sPath, FN, GetModuleFileName(hInstance, FN, SizeOf(FN)));
  Result := ExtractFilePath( sPath ) + ExtractFileName( sFile );
end;

procedure TWebModule1.WebModule1waListAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  cdsJoin.Open;
  Response.Content := dstpJoin.Content;
end;

procedure TWebModule1.dstpJoinFormatCell(Sender: TObject; CellRow,
CellColumn: Integer; var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if (CellRow > 0) and (CellColumn = 0) then
    CellData := Format('<a href="%s/getrecord?empno=%s">%s</a>',
      [Request.ScriptName, CellData, CellData]);
end;

procedure TWebModule1.WebModule1waEditAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  dsppJoin.HTMLFile := LocalServerPath('join.htm');
  cdsJoin.Filter := 'EmpNo = ' + DataFields.Values['empno'];
  cdsJoin.Filtered := true;
  Response.Content := dsppJoin.Content;
end;

procedure TWebModule1.dsppJoinHTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if CompareText(TagString, 'SCRIPTNAME')=0 then
   ReplaceText:=Request.ScriptName;
end;

procedure TWebModule1.WebModule1waUpdateAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
```

```
var
  EmpNo, LastName, FirstName, HireDate, Salary, Vacation: string;
begin
  EmpNo:=DataFields.Values['EmpNo'];
  LastName:=DataFields.Values['LastName'];
  FirstName:=DataFields.Values['FirstName'];
  HireDate:=DataFields.Values['HireDate'];
  Salary:=DataFields.Values['Salary'];
  Vacation:=DataFields.Values['Vacation'];

  cdsJoin.Open;
  if cdsJoin.Locate('EMPNO', EmpNo, []) then
  begin
    cdsJoin.Edit;
    cdsJoin.FieldByName('LastName').AsString:=LastName;
    cdsJoin.FieldByName('FirstName').AsString:=FirstName;
    cdsJoin.FieldByName('HireDate').AsString:=HireDate;
    cdsJoin.FieldByName('Salary').AsString:=Salary;
    cdsJoin.FieldByName('Vacation').AsString:=Vacation;
    if cdsJoin.ApplyUpdates(0)=0 then
      Response.Content:=ppSuccess.Content else
      Response.Content:=pPError.Content;
  end;
end;

end.
```

Обратите внимание: данный метод требует написания большого объема кода. Поэтому в примере реализован не полный набор возможностей приложений MIDAS, в частности отсутствует согласование ошибок доступа. При активном использовании рассмотренного подхода этот пример можно усовершенствовать, сделав его более устойчивым.

При создании модуля WebModule и приложения сервера использовалась концепция учета состояния. Поскольку протокол HTTP не зависит от состояния, нельзя гарантировать, что по окончании соединения значения свойств останутся прежними.

Пример использования этого подхода можно найти на прилагаемом компакт-диске — в подкаталоге \WebBrok каталога, соответствующего данной главе.

Компоненты InternetExpress

С помощью компонентов InternetExpress можно расширить функциональные возможности простого модуля WebModule и улучшить разрабатываемое клиентское приложение. Это возможно благодаря использованию в них открытых стандартов XML и JavaScript. С помощью компонентов InternetExpress можно создавать серверные приложения MIDAS, взаимодействующие лишь с броузером. Со стороны клиента не потребуется загружать никакие управляющие элементы ActiveX или выполнять какую-либо предварительную установку и настройку программного обеспечения — достаточно будет просто адресовать Web-броузер на соответствующую страницу Web-сервера.

При использовании компонентов InternetExpress на Web-сервере потребуется запустить дополнительное программное обеспечение. В данном примере будет использоваться приложение ISAPI, однако эту роль может выполнять также приложение CGI или ASP. Web-брокер получает запросы от броузера и передает их на сервер приложений. Это легко осуществить, поместив компоненты InternetExpress в приложение Web-брокера.

В рассматриваемом примере используется стандартное приложение MIDAS, в котором имеются наборы данных `Customers`, `Orders` и `Employees`. Наборы данных `Customers` и `Employees` являются вложенными (для получения более подробной информации о вложенных наборах данных читайте следующий раздел), а набор данных `Employees` будет использоваться в качестве справочной таблицы. Воспользуйтесь прилагаемым исходным кодом определения сервера приложений. После создания и регистрации сервера приложений можно будет сосредоточиться на построении приложения Web-брокера, которое будет взаимодействовать с сервером приложений.

Создайте новое ISAPI-приложение, выбрав команду File⇨New⇨Web Server Application в хранилище объектов. Поместите в модуль `WebModule` компонент `TDCOMConnection`. Он будет обеспечивать связь с сервером приложений. Поэтому задайте в качестве значения его свойства `ServerName` идентификатор `ProgID` сервера приложений.

Затем поместите в `WebModule` компонент `TXMLBroker`, расположенный во вкладке InternetExpress палитры компонентов, и установите значения его свойств `RemoteServer` и `ProviderName` равными `CustomerProvider`. Компонент `TXMLBroker` функционирует аналогично компоненту `TCLientDataSet`. Он будет использоваться для получения пакетов данных с сервера приложений и передачи их броузеру. Основным отличием между пакетами данных компонентов `TXMLBroker` и `TClientDataSet` является то, что компонент `TXMLBroker` преобразует пакеты данных MIDAS в формат XML. Добавим также в `WebModule` компонент `TCLientDataSet` и свяжем его с провайдером `Employees` на сервере приложений. Позже этот компонент будет использоваться как источник данных выборки.

Компонент `TXMLBroker` отвечает за соединение с приложением сервера и за навигацию по страницам HTML. Для настройки поведения приложения InternetExpress можно воспользоваться множеством параметров. Например, можно ограничить количество записей, передаваемых клиенту, или задать число допустимых ошибок во время обновления.

Теперь необходимо определить способ передачи данных в броузер. Используя компонент `TMidasPageProducer`, можно применить технологию WebBroker для обработки HTML-страниц в броузере. Однако компонент `TMidasPageProducer` также допускает визуальное создание Web-страниц в редакторе Web Page Editor.

Дважды щелкните на компоненте `TMidasPageProducer` — на экране раскроется окно редактора Web-страниц. Этот визуальный редактор позволяет определить, какие элементы будут содержаться на данной Web-странице. Одной из самых интересных особенностей компонентов InternetExpress является их абсолютная открытость. Можно создавать свои собственные компоненты, которые будут использоваться в редакторе Web-страниц в соответствии с четко определенными правилами. Примеры компонентов InternetExpress содержатся в каталоге `<`*`каталог Delphi`*`>\DEMOS\MIDAS\INTERNET-EXPRESS\INETCUSTOM`.

Компонент `TMidasPageProducer` имеет свойство `IncludePathURL`. Содержащееся в нем значение существенно влияет на работоспособность всего приложения InternetExpress. Установите для него значение, соответствующее виртуальному каталогу с файлами сценариев JavaScript, использующимися в приложении. Например, если файлы находятся в каталоге `c:\inetpub\wwwroot\jscript`, то свойство `IncludePathURL` должно иметь значение `/jscript/`.

Активизируйте редактор Web-страниц, а затем щелкните на кнопке New Item (Ins), чтобы вывести на экран диалоговое окно Add Web Component (рис. 32.7). В списке этого диалогового окна содержится перечень компонентов Web, которые могут быть добавлены на HTML-страницу. Содержимое этого списка зависит от родительского компонента, выделенного в текущий момент в верхней левой области диалогового окна. Например, добавление в корневой узел Web-компонента DataForm позволит конечным пользователям просматривать и редактировать информацию из базы данных, представленную в виде некоторой формы.

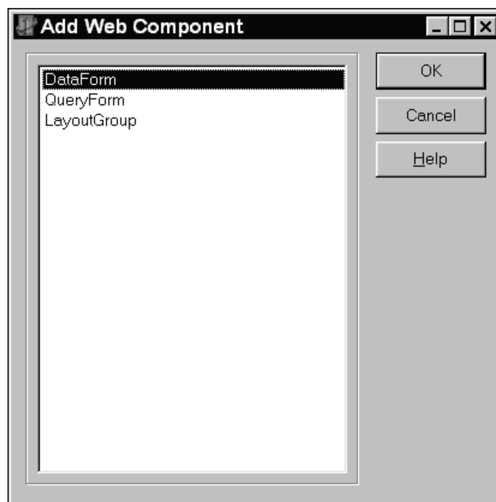


Рис. 32.7. Диалоговое окно Add Web Component *редактора Web-страниц*

Выделив в редакторе Web-страниц элемент DataForm, можно снова щелкнуть на кнопке New Item (Ins). Обратите внимание, что перечень доступных компонентов в этом случае будет отличаться от списка, выведенного на предыдущем шаге. После выбора компонента FieldGroup во вкладке предварительного просмотра появится предупреждение, сообщающее о том, что свойству `XMLBroker` компонента `FieldGroup` не присвоено никакое значение. Заданные в редакторе свойств значения свойства `XMLBroker` сразу же отобразятся в окне предварительного просмотра редактора Web-страниц. Если продолжить редактировать свойства или добавлять компоненты, состояние HTML-страницы будет постоянно изменяться (рис. 32.8).

Стандартные компоненты Web обеспечивают практически неограниченные возможности настройки. С помощью свойств легко изменить заголовки полей, тип выравнивания и цвет, добавить собственный HTML-код, и даже использовать листы стилей. Более того, если компонент не полностью удовлетворяет вашим потребностям, то можно создать его потомка и использовать его в редакторе Web-страниц. Возможности среды ограничиваются лишь воображением разработчика.

При вызове динамической библиотеки ISAPI ее необходимо поместить в виртуальный каталог, использующийся для запуска сценариев. Необходимо также переместить файлы JavaScript, содержащиеся в каталоге `<`*`каталог DELPHI`*`>\SOURCE\WEBMIDAS`, в корректное местоположение на Web-сервере и модифицировать свойство `TMidasPageProducer.IncludePathURL` таким образом, чтобы в качестве его значения использовался адрес URL файлов JavaScript. После выполнения всех перечисленных действий страница будет готова к просмотру.

Для доступа к странице необходимо иметь лишь JavaScript-совместимый броузер. Укажите в броузере адрес `http://localhost/inetx/inetxisapi.dll` — и данные будут отображены, как показано на рис. 32.9.

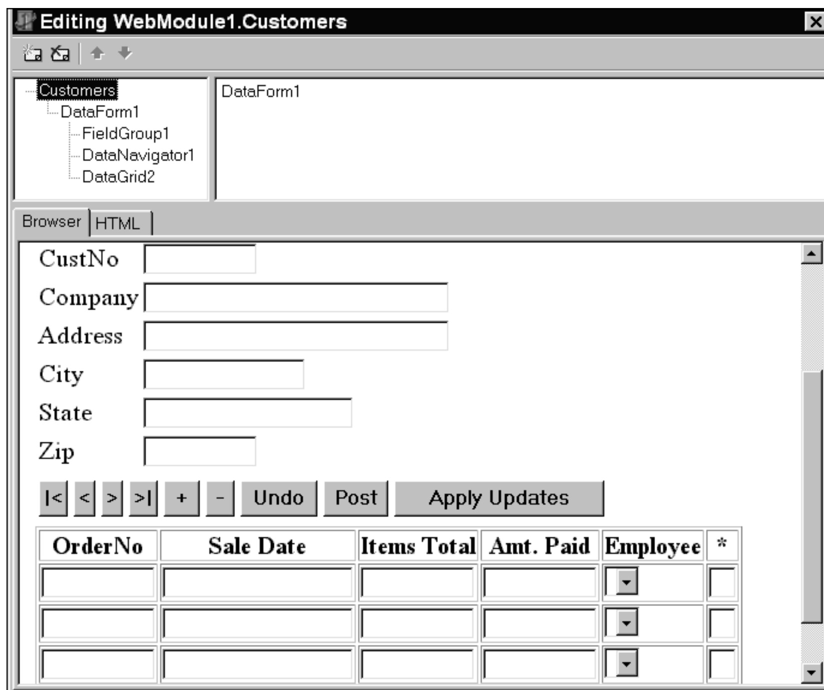


Рис. 32.8. Редактор Web-страниц после разработки HTML-страницы

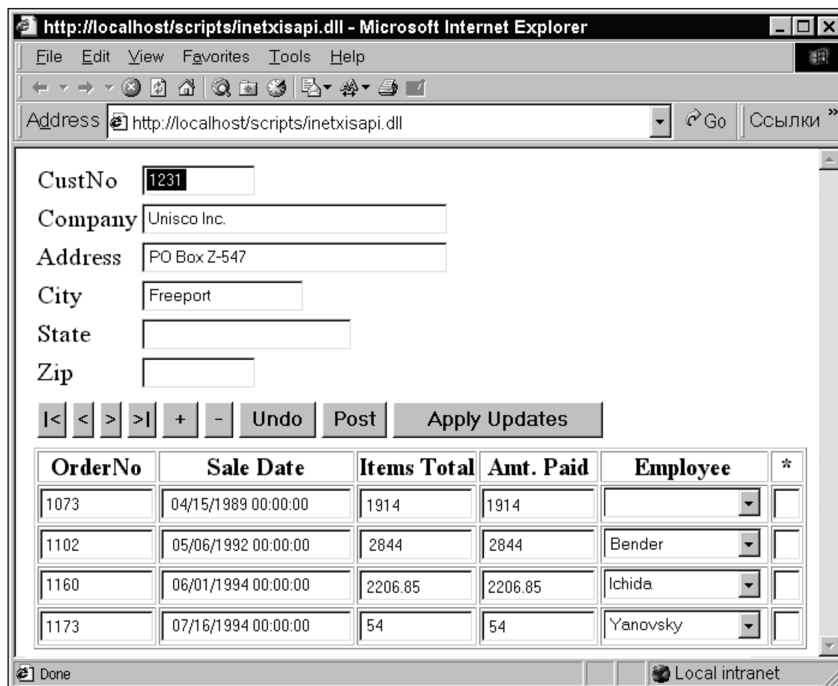


Рис. 32.9. Internet Explorer с загруженной Web-страницей приложения InternetExpress

И, наконец, в процессе применения обновлений можно выполнять согласование ошибок доступа подобно тому, как это производилось в автономном приложении MIDAS (рис. 32.10). Эту возможность можно использовать, если значение свойства TXMLBroker.ReconcileProducer равно TPageProducer. При возникновении ошибки этому свойству будет присвоено значение свойства Content компонента TPageProducer, которое затем будет возвращено конечному пользователю.

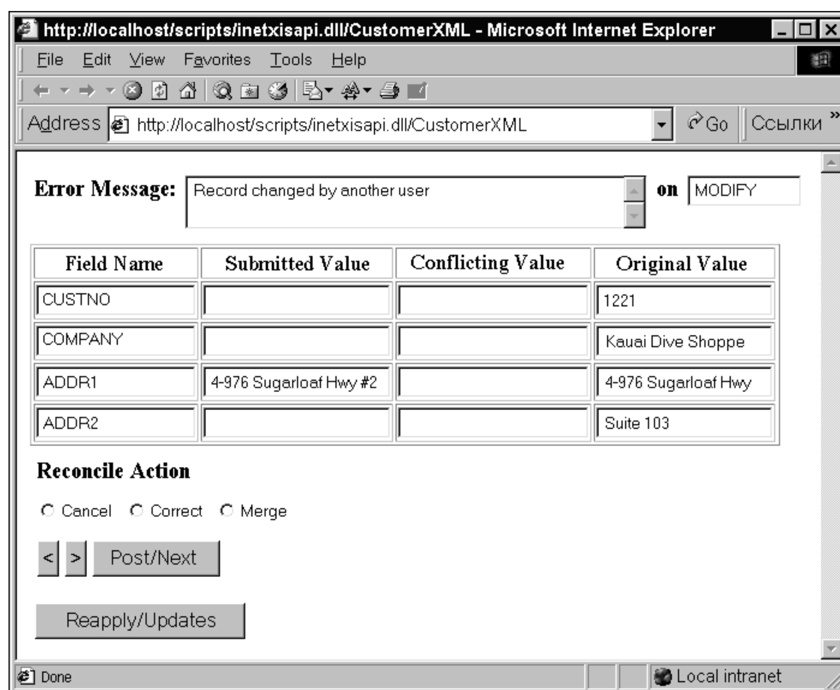


Рис. 32.10. HTML-страница, сгенерированная компонентом TReconcilePageProducer

После установки пакета InetXCustom.dpk, содержащегося в каталоге <*каталог DELPHI*>\DEMOS\MIDAS\INTERNETEXPRESS\INETXCUSTOM, будет доступен компонент TReconcilePageProducer, представляющий собой специализированный тип компонента TPageProducer. Этот компонент генерирует код HTML, который функционирует аналогично диалоговому окну Reconciliation Error в стандартном приложении MIDAS.

Пример, иллюстрирующий изложенный материал, можно найти на прилагаемом компакт-диске — в подкаталоге \InetX каталога, соответствующего данной главе.

Дополнительные возможности наборов данных клиента

Управление компонентом TClientDataSet осуществляется с помощью установки множества различных параметров. В данном разделе будут рассмотрены способы использования компонента TClientDataSet, упрощающие код в сложных приложениях.

Вложенные наборы данных

Вложенные наборы данных уже кратко упоминались в предыдущих разделах. Теперь познакомимся с ними поближе.

Для создания вложенных наборов данных в приложении сервера необходимо определить отношение “главная–детальная”. Это можно осуществить точно так же, как и в приложениях клиент/сервер, а именно — определить SQL-оператор для объекта детального запроса TQuery, который включал бы параметр связи. Например:

```
"select * orders where custno=:custno"
```

Затем присвойте свойству TQuery.Datasource детального запроса TQuery значение, указывающее на компонент TDatasource, связанный с компонентом TDataSet главного набора данных. После установки связи останется лишь экспортировать компонент TDataSetProvider, связанный с главным набором данных. Приложение MIDAS достаточно интеллектуально, чтобы понять, что у главного набора данных имеются детальные наборы данных, связанные с ним и передаваемые клиенту как экземпляры класса TDataSetField.

В клиентской части приложения присвойте имя главного провайдера свойству TCLientDataSet.ProviderName. Затем добавьте в компонент TClientDataSet постоянные поля. Обратите внимание на последнее поле в окне редактора полей. Его имя совпадает с именем детального набора данных на сервере и имеет тип TDataSetField. На данный момент имеется уже достаточно информации, чтобы использовать вложенный набор данных в коде. Однако для простоты можно добавить компонент TClientDataSet для детального набора данных и присвоить его свойству DataSetField имя соответствующего компонента TDataSetField главного набора данных. При этом важно помнить, что в детальном наборе данных не нужно устанавливать никаких других свойств, таких, например, как RemoteServer, ProviderName, MasterSource, MasterFields или PacketRecords. Необходимо лишь установить значение свойства DataSetField. Теперь с детальным набором данных можно связать необходимые управляющие элементы.

После завершения работы со вложенными наборами данных, внесенные обновления необходимо зафиксировать в базе данных. Это можно осуществить с помощью вызова метода ApplyUpdates главного компонента TCLientDataSet. В результате приложение MIDAS одной транзакцией внесет все изменения в главный набор данных, содержащийся на сервере, в который включены и детальные наборы данных.

Соответствующий пример можно найти на прилагаемом компакт-диске — в подкаталоге \NestCDS каталога, соответствующего данной главе.

Связь “главная-детальная” в клиентской части приложения

Напомним о нескольких предостережениях, которые упоминались при рассмотрении вложенных наборов данных. Альтернативой использованию вложенных наборов данных является создание связи “главная–детальная” в клиентской части приложения. В этом случае на сервере необходимо создать компоненты TDataSet и TDataSetProvider для главного и детального наборов данных.

В клиентской части свяжите с наборами данных два компонента TClientDataSet, экспортируемых на сервер. Затем создайте связь “главная–детальная”, присвоив свойству TClientDataSet.MasterSource детального набора данных источник данных TDataSource, связанный с главным набором данных.

При установке свойства MasterSource компонента TClientDataSet свойству PacketRecords присваивается нулевое значение. Это означает, что в данном случае приложение MIDAS должно извлекать с сервера только метаданные. Однако, если свойство PacketRecords имеет нулевое значение в контексте связи "главная–детальная", то смысл несколько меняется. В этом случае приложение MIDAS для каждой главной записи будет извлекать весь набор связанных детальных записей. Одним словом, оставьте в свойстве PacketRecords значение, используемое по умолчанию.

Для передачи данных, связанных по принципу "главная–детальная", в базу данных за одну транзакцию разработчику необходимо написать свою собственную реализацию метода ApplyUpdates. Эта задача является далеко не такой простой, как большинство задач, решаемых в Delphi. Однако в этом случае разработчику предоставляется полный контроль над процессом обновления.

Применение обновлений к единственной таблице обычно осуществляется с помощью метода TCLientDataSet.ApplyUpdates. Это метод передает измененные записи от компонента ClientDataSet его провайдеру на среднем уровне, который, в свою очередь, записывает изменения в базу данных. При этом все необходимые действия выполняются в рамках одной транзакции и могут быть выполнены без вмешательства программиста. Чтобы то же самое выполнить при использовании связи "главная–детальная", необходимо понимать, какие действия выполняются Delphi при вызове метода TClientDataSet.ApplyUpdates.

Все изменения, внесенные в компонент TClientDataSet, хранятся в свойстве Delta. В этом свойстве содержится вся информация, которая в конечном счете будет записана в базу данных. В следующем фрагменте кода иллюстрируется процесс обновления, при котором содержимое свойств Delta передается обратно в базу данных. В листингах 32.5 и 32.6 приведены фрагменты кода клиента и сервера для выполнения обновлений при использовании связи "главная–детальная".

В исходной версии Delphi 5 содержалась ошибка, из-за которой в контексте одной транзакции было невозможно обновить данные на сервере с использованием содержимого нескольких свойств Delta. Если такая возможность вам все же необходима, замените соответствующий метод в модуле DBTABLES.PAS следующим кодом:

```
function TDBDataSet.PSInTransaction: Boolean;
var
  InProvider: Boolean;
begin
  InProvider := SetDBFlag(dbfProvider, True);
  try
    Result := Database.InTransaction;
  finally
    SetDBFlag(dbfProvider, InProvider);
  end;
end;
```

Соответствующий пример можно найти на прилагаемом компакт-диске — в подкаталоге \MDCDS каталога, относящегося к данной главе.

Листинг 32.5. Обновления связи "главная–детальная" в клиентской части

```
procedure TClientDM.ApplyUpdates;
var
  MasterVar, DetailVar: OleVariant;
begin
  Master.CheckBrowseMode;
```

```
  Detail_Proj.CheckBrowseMode;
  if Master.ChangeCount > 0 then
    MasterVar := Master.Delta else
    MasterVar := NULL;
  if Detail.ChangeCount > 0 then
    DetailVar := Detail.Delta else
    DetailVar := NULL;
  RemoteServer.AppServer.ApplyUpdates(DetailVar, MasterVar);
  { Согласование ошибок в пакетах данных. Поскольку предполагается
    отсутствие ошибок, то ошибки могут содержаться лишь в одном пакете.
    Обновление данных производится тогда, когда ни в одном пакете нет ошибок.}
  if not VarIsNull(DetailVar) then
    Detail.Reconcile(DetailVar) else
  if not VarIsNull(MasterVar) then
    Master.Reconcile(MasterVar) else
  begin
    Detail.Reconcile(DetailVar);
    Master.Reconcile(MasterVar);
    Detail.Refresh;
    Master.Refresh;
  end;
end;
```

Листинг 32.6. Обновления связи "главная–детальная" в серверной части

```
procedure TServerRDM.ApplyUpdates(var DetailVar, MasterVar: OleVariant);
var
  ErrCount: Integer;
begin
Database.StartTransaction;
  try
    if not VarIsNull(MasterVar) then
    begin
      MasterVar := cdsMaster.Provider.ApplyUpdates(MasterVar, 0, ErrCount);
      if ErrCount > 0 then
        SysUtils.Abort;    // Отмена внесенных изменений
    end;
    if not VarIsNull(DetailVar) then
    begin
      DetailVar := cdsDetail.Provider.ApplyUpdates(DetailVar, 0, ErrCount);
      if ErrCount > 0 then
        SysUtils.Abort;    // Отмена внесенных изменений
    end;
    Database.Commit;
  except
    Database.Rollback
  end;
end;
```

Хотя данный метод работает достаточно хорошо, тем не менее он не поддерживает возможность повторного использования его кода. Было бы неплохо такую возможность обеспечить. Для этого достаточно выполнить следующие действия.

1. Поместите свойства Delta каждого набора данных клиента в массив типа Variant.
2. Поместите провайдеры каждого набора данных клиента в массив типа Variant.
3. Используйте все свойства Delta в одной транзакции.
4. Обеспечьте согласование ошибок доступа в пакетах данных, возвращенных на предыдущем шаге, и обновите данные.

Эти рекомендации реализованы в утилите, текст которой представлен в листинге 32.7.

Листинг 32.7. Модуль, содержащий процедуры утилиты и требуемую степень абстракции

```
unit CDSUtil;

interface

uses
  DbClient, DbTables;

function RetrieveDeltas(const cdsArray : array of TClientDataSet): Variant;
function RetrieveProviders(const cdsArray : array of TClientDataSet): Variant;
procedure ReconcileDeltas(const cdsArray : array of TClientDataSet;
  vDeltaArray: OleVariant);

procedure CDSApplyUpdates(ADatabase : TDatabase; var vDeltaArray: OleVariant;
  const vProviderArray: OleVariant);

implementation

uses
  SysUtils, Provider,
  {$IFDEF VER130}Midas{$ELSE}StdVcl{$ENDIF};

type
  PArrayData = ^TArrayData;
  TArrayData = array[0..1000] of Olevariant;

{ На входе свойство Delta принимает значение CDS.Delta. На выходе Delta
  будет содержать пакеты данных, содержащие все записи, которые не удалось
  занести в базу данных. Помните, что в Delphi 5 необходимо указать
  имя провайдера, поэтому оно передается в первом элементе AProvider.}
procedure ApplyDelta(AProvider: OleVariant; var Delta : OleVariant);
var
  ErrCount : integer;
  OwnerData: OleVariant;
begin
  if not VarIsNull(Delta) then
  begin
    // ScktSrvr не поддерживает раннее связывание
```

```
{$IFDEF VER130}
    Delta := (IDispatch(AProvider[0]) as IAppServer).AS_ApplyUpdates(
      AProvider[1], Delta, 0, ErrCount, OwnerData);
{$ELSE}
    Delta := OleVariant(IDispatch(AProvider)).ApplyUpdates(Delta, 0, ErrCount);
{$ENDIF}
    if ErrCount > 0 then
      SysUtils.Abort;  // В вызове процедуры это приведет к отмене изменений
  end;
end;

{Вызов из сервера}
procedure CDSApplyUpdates(ADatabase : TDatabase; var vDeltaArray: OleVariant;
  const vProviderArray: OleVariant);
var
  i : integer;
  LowArr, HighArr: integer;
  P: PArrayData;
begin
  { Все обновления в одной транзакции. Если некоторый шаг приводит к ошибке,
   то генерируется исключительная ситуация, приводящая к отмене транзакции. }
  ADatabase.Connected:=true;
  ADatabase.StartTransaction;
  try
    LowArr:=VarArrayLowBound(vDeltaArray,1);
    HighArr:=VarArrayHighBound(vDeltaArray,1);
    P:=VarArrayLock(vDeltaArray);
    try
      for i:=LowArr to HighArr do
        ApplyDelta(vProviderArray[i], P^[i]);
    finally
      VarArrayUnlock(vDeltaArray);
    end;
    ADatabase.Commit;
  except
    ADatabase.Rollback;
  end;
end;

{Вызовы из клиентской части}
function RetrieveDeltas(const cdsArray : array of TClientDataSet): Variant;
var
  i : integer;
  LowCDS, HighCDS : integer;
begin
  Result:=NULL;
  LowCDS:=Low(cdsArray);
  HighCDS:=High(cdsArray);
  for i:=LowCDS to HighCDS do
    cdsArray[i].CheckBrowseMode;
```

```
  Result:=VarArrayCreate([LowCDS, HighCDS], varVariant);
  { Задание массива Variant с изменениями
   (или NULL, если изменения отсутствуют) }
  for i:=LowCDS to HighCDS do
  begin
    if cdsArray[i].ChangeCount>0 then
      Result[i]:=cdsArray[i].Delta else
      Result[i]:=NULL;
  end;
end;

{ В Delphi 5 эта функция возвращает имя провайдера и имя приложения.
  Позже для вызова AS_ApplyUpdates в функции CDSApplyUpdates
  будет использоваться ProviderName.}
function RetrieveProviders(const cdsArray : array of TClientDataSet): Variant;
var
  i: integer;
  LowCDS, HighCDS: integer;
begin
  Result:=NULL;
  LowCDS:=Low(cdsArray);
  HighCDS:=High(cdsArray);

  Result:=VarArrayCreate([LowCDS, HighCDS], varVariant);
  for i:=LowCDS to HighCDS do
{$IFDEF VER130}
    Result[i]:=VarArrayOf([cdsArray[i].AppServer, cdsArray[i].ProviderName]);
{$ELSE}
    Result[i]:=cdsArray[i].Provider;
{$ENDIF}
end;

procedure ReconcileDeltas(const cdsArray : array of TClientDataSet;
  vDeltaArray: OleVariant);
var
  bReconcile : boolean;
  i: integer;
  LowCDS, HighCDS : integer;
begin
  LowCDS:=Low(cdsArray);
  HighCDS:=High(cdsArray);

  { Если предыдущий шаг завершился с ошибками, то
    выполняется согласование ошибок пакетов данных.}
  bReconcile:=false;
  for i:=LowCDS to HighCDS do
    if not VarIsNull(vDeltaArray[i]) then begin
      cdsArray[i].Reconcile(vDeltaArray[i]);
      bReconcile:=true;
      break;
    end;
```

```
  {При необходимости обновление наборов данных}
  if not bReconcile then
    for i:=HighCDS downto LowCDS do begin
      cdsArray[i].Reconcile(vDeltaArray[i]);
      cdsArray[i].Refresh;
    end;
end;

end.
```

В листинге 32.8 предыдущий пример модифицирован с помощью модуля `CDSUtil`.

Листинг 32.8. Предыдущий пример, но с использованием модуля `CDSUtil.pas`

```
procedure TForm1.btnApplyClick(Sender: TObject);
var
  vDelta: OleVariant;
  vProvider: OleVariant;
  arrCDS: array[0..1] of TClientDataSet;
begin
  arrCDS[0]:=cdsMaster;  // Задание массива ClientDataSet
  arrCDS[1]:=cdsDetail;

  vDelta:=RetrieveDeltas(arrCDS);                 // Шаг 1
  vProvider:=RetrieveProviders(arrCDS);           // Шаг 2
  DCOMConnection1.ApplyUpdates(vDelta, vProvider); // Шаг 3
  ReconcileDeltas(arrCDS, vDelta);                // Шаг 4
end;

procedure TServerRDM.ApplyUpdates(var vDelta, vProvider: OleVariant);
begin
  CDSApplyUpdates(Database1, vDelta, vProvider);    // Шаг 3
end;
```

Этот модуль можно использовать и в двух- или трехуровневых приложениях. Для того чтобы применяемый подход перенести с двух- на трехуровневую модель, вместо того чтобы вызывать эту функцию в клиентской части, на сервере необходимо экспортировать функцию, в которой вызывается метод `CDSApplyUpdates`. Весь остальной код клиента остается без изменений.

Двухуровневые приложения

Теперь вы знаете, как в трехуровневном приложении компоненту `ClientDataSet` поставить в соответствие провайдера, а значит, и данные. Однако зачастую достаточно создать простое двухуровневое приложение. Как выполнить подобные действия в этом случае? Для этого существует четыре возможности.

- Присвоить данные во время выполнения.
- Присвоить данные во время разработки.
- Назначить провайдера во время выполнения.
- Назначить провайдера во время разработки.

При работе с компонентом ClientDataSet существует две основные возможности: либо использовать свойство AppServer, либо обращаться к данным напрямую. В первом случае между компонентами TClientDataSet и TDataSetProvider необходимо установить связь, что позволит им взаимодействовать друг с другом. Если же потребуется работать непосредственно с самими данными, в вашем распоряжении будет эффективный механизм локального хранения данных. При этом для получения данных нет необходимости обращаться к компоненту TDataSetProvider.

Для получения компонентом TCLientDataSet данных непосредственно от компонента TDataSet во время выполнения программы используйте код, приведенный в листинге 32.9.

Листинг 32.9. Код для получения данных непосредственно от компонента TDataSet

```
function GetData(ADataSet: TDataSet): OleVariant;
begin
  with TDatasetProvider.Create(nil) do
  try
    Dataset:=ADataset;
    Result:=Data;
  finally
    Free;
  end;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  ClientDataset1.Data:=GetData(ADOTable1);
end;
```

При использовании такого метода в Delphi 5 требуется больше усилий и кода, чем в предыдущих версиях Delphi, в которых достаточно было свойству ClientDataSet1.Data присвоить свойство Table1.Provider.Data. Однако функция GetData() позволяет скрыть дополнительный код.

Для того чтобы из компонента TDataSet извлечь данные с помощью компонента TClientDataSet в процессе разработки, выберите в контекстном меню компонента TClientDataSet команду Assign Local Data. Затем задайте компонент TDataSet, в котором содержатся требуемые данные, и они будут помещены в свойство TCLientDataSet.Data.

Если сохранить файл в этом состоянии и сравнить размер DFM-файла с размером данных до выполнения этой команды, то можно заметить, что размер DFM-файла увеличился. Это происходит потому, что в DFM-файле Delphi сохраняет все метаданные и записи, ассоциированные с компонентом TDataSet. Эти данные будут переданы в файл DFM только в том случае, если значение свойства Active компонента TClientDataSet установлено равным True. Объем файла также можно сократить, выбрав в контекстном меню компонента TClientDataSet команду Clear Data.

Для того чтобы воспользоваться всей гибкостью, предоставляемой провайдером, необходимо обратиться к свойству AppServer. Нужное значение этому свойству можно присвоить во время выполнения. Это можно сделать в методе FormCreate примерно так:

```
ClientDataset1.AppServer:=TLocalAppServer.Create(Table1);
ClientDataset1.Open;
```

И, наконец, установить значение свойства AppServer можно во время разработки. Если значение свойства RemoteServer компонента TCLientDataSet не установлено, то свойству TClientDataSet.ProviderName можно присвоить значение TDataSetProvider.

Основным отличием между использованием компонента TDataSet и компонента TClientDataSet является то, что при использовании компонента TClientDataSet в качестве посредника между запросами к данным компонента TDataSet используется интерфейс IAppServer. Это позволяет манипулировать свойствами, методами, событиями и полями компонента TClientDataSet, но не компонента TDataSet. Можно считать, что этот компонент содержится в отдельном приложении и, следовательно, им нельзя управлять непосредственно из кода. Поместите все "серверные" компоненты в отдельный модуль DataModule. Размещение компонентов TDatabase, TDataSet и TCDSProvider в отдельном модуле DataModule позволяет заранее подготовить создаваемое приложение к предстоящему развертыванию его в многоуровневой среде. Такая организация приложения обладает еще одним преимуществом: при ее использовании модуль DataModule можно рассматривать как объект, доступ к которому для клиента будет затруднен. Это также будет способствовать подготовке приложения к переносу в многоуровневую среду, поскольку затруднит разработчикам создание таких связей, которые впоследствии будут препятствовать переносу.

На заметку

Во время разработки свойство TClientDataSet.ProviderName не может быть связано с провайдерами, расположенными в другой форме или модуле. Поэтому во время выполнения в коде нужно установить значение свойства TClientDataSet.AppServer.

Установка MIDAS-приложений

После создания MIDAS-приложения, последним барьером, который нужно преодолеть, является его установка и развертывание. В этом разделе мы обсудим, что необходимо сделать для безболезненной установки приложения, созданного при использовании технологии MIDAS.

Предоставление лицензии

С момента появления технология MIDAS в Delphi 3, многих пользователей постоянно интересует вопрос предоставления лицензий. Распространение приложений, созданных с применением этой технологии, усложняется огромным количеством нюансов. В данном разделе описываются общие требования, определяющие необходимость приобретения лицензии MIDAS. Однако единственный законный документ, связанный с лицензированием, содержится в файле DEPLOY.TXT, находящемся в корневом каталоге Delphi 5. И, наконец, последней инстанцией, которая может ответить на вопросы, возникшие в конкретной ситуации, является региональный отдел продаж компании Borland. Дополнительные руководства и примеры можно найти по адресу:

```
http://www.borland.com/midas/papers/lisensing
```

или на нашем Web-узле (http://www.xapware.com/ddg).

Информация в этом документе подготовлена таким образом, чтобы можно было получить ответы на вопросы, возникающие в некоторых общих случаях использования технологии MIDAS. В этом документе содержится также информация о ценах и условиях приобретения лицензии.

В процессе принятия решения о приобретении лицензии MIDAS основным критерием является вопрос о том, будут ли пакеты данных MIDAS передаваться за пределы компьютера. Если это так, то лицензию вам приобрести необходимо. В противном случае (как, например, в случае одно- и двухуровневых приложений, рассмотренных ранее) вы можете использовать технологию MIDAS, не приобретая лицензии.

Настройка DCOM

Настройка протокола DCOM является и наукой, и искусством. Полная и безопасная настройка DCOM определяется очень многими аспектами, поэтому в данном разделе мы познакомимся лишь с основами этой “черной магии”.

После регистрации приложение сервера можно настроить с помощью утилиты `DCOMCNFG` компании Microsoft. Эта утилита входит в комплект поставки системы Windows NT, однако для компьютеров под управлением Windows 9x ее потребуется загрузить отдельно. Сразу же следует отметить, что в утилите `DCOMCNFG` имеется множество ошибок. Наиболее примечательной из них является то, что утилита может быть запущена лишь на тех компьютерах под управлением Windows 9x, на которых применяется режим совместного использования ресурсов на уровне пользователей. А это, безусловно, требует наличия домена, что не всегда возможно или желательно в одноранговых сетях (например, в сети из двух компьютеров под управлением Windows 9x). В результате, многие пользователи склонны думать, что для запуска утилиты настройки протокола DCOM требуется компьютер под управлением Windows NT.

Если утилита `DCOMCNFG` запущена, то можно выделить зарегистрированное приложение сервера и щелкнуть на кнопке Properties, чтобы вывести на экран информацию о сервере. Изучение утилиты `DCOMCNFG` удобно начать с вкладки Identity. Для зарегистрированного объекта сервера по умолчанию используется режим Launching User. Компания Microsoft при всем своем желании не могла бы принять худшего решения.

При создании сервера подпрограммами DCOM обычно используется контекст безопасности пользователя, указанного на странице Identity. Если выбран режим Launching User, то для каждого отдельного зарегистрированного пользователя будет запущен новый процесс. Во многих случаях пользователи выбирают режим создания экземпляров `ciMultiInstance`, а потом удивляются, почему создается несколько копий сервера. Например, если с сервером соединяется пользователь А, а затем — пользователь В, то для пользователя В будет запущен новый процесс. Кроме того, если пользователь регистрируется на машине под именем, отличным от используемого сервером в данный момент, его графический интерфейс будет этому пользователю недоступен. Это обусловлено концепцией системы Windows NT, известной под названием *станций Windows* (Windows stations). Осуществлять запись в графический пользовательский интерфейс на станции Windows может лишь тот, кто имеет статус Interactive User. Таким пользователем является тот, кто зарегистрирован на сервере в текущий момент. В общем, при настройке сервера приложения никогда не используйте режим Launching User.

Следующим интересным вариантом является имеющийся во вкладке Identity режим Interactive User. Это означает, что пользователь, создавший сервер, будет работать в контексте пользователя, зарегистрированного на сервере в текущий момент времени. При этом вы сможете визуально взаимодействовать с сервером. Но, к сожалению, большинство системных администраторов запрещают, зарегистрировавшись на компьютере Windows NT, бездействовать. Кроме того, если зарегистрировавшийся пользователь решит закончить работу, то приложение сервера перестанет нормально работать.

Во вкладке Identity остался еще один режим — This User. Если выбран этот режим, все клиенты будут создавать одно приложение сервера и пользоваться правами доступа и контекстом пользователя, определенного во вкладке Identity. Это также означает, что для запуска сервера компьютер Windows NT не потребует регистрации пользователя. Побочным эффектом такого подхода является то, что при использовании данного параметра не будет отображаться пользовательский графический интерфейс сервера. Однако чтобы обеспечить правильную работу приложения сервера, этот режим следует предпочесть всем остальным.

После корректной настройки объекта сервера обратите внимание на вкладку Security. Убедитесь, что пользователь, который будет запускать этот объект, имеет соответствующие права доступа. Удостоверьтесь также в том, что доступ к серверу предоставлен и пользователю SYSTEM, в противном случае возникнут ошибки.

В процессе настройки протокола DCOM имеется множество нюансов. Для получения самой свежей информации об этом протоколе, особенно в случае его использования совместно с Windows 9x, Delphi и MIDAS, посетите наш Web-узел по адресу:

http://www.DistribuCon.com/dcom95.htm

Файлы, необходимые для установки приложения

Требования к установке приложения MIDAS изменяются с выходом каждой новой версии Delphi. По сравнению с предыдущими версиями, в Delphi 5 этот процесс существенно упрощен. В предыдущих версиях и на сервере, и машине клиента нужно было устанавливать файл DBCLIENT.DLL. В этом файле содержалась реализация компонента TCLientDataSet. Библиотеку DBCLIENT.DLL нужно было зарегистрировать и на компьютере клиента. Требовались также другие файлы, такие, например, как STDVCL32.DLL, STDVCL40DLL и IDPROV32.DLL. Если хотя бы один файл отсутствовал или был неправильно зарегистрирован, приложение работало некорректно.

В Delphi 5 минимальный набор файлов, требуемый для установки приложения MIDAS, указан в приведенных ниже перечнях рекомендуемых действий.

Для установки серверной части приложения выполните следующие действия.

1. Скопируйте приложение сервера в каталог с необходимыми привилегиями NTFS.
2. Установите требуемый уровень доступа к данным, чтобы приложение сервера могло взаимодействовать в качестве клиента с СУБД (например, BDE, MDAC, специфическими для клиента библиотеками базами данных, и т.д.).
3. Скопируйте файл MIDAS.DLL в каталог %SYSTEM%. Для компьютеров, работающих под управлением Windows NT, таким каталогом по умолчанию является C:\Winnt\System32, а для компьютеров, на которых установлена система Windows 9x, — C:\Windows\System.
4. Запустите приложение сервера, чтобы зарегистрировать его как COM-объект.

Для установки клиентской части выполните следующие действия.

1. Скопируйте клиентское приложение в каталог вместе с любыми другими внешними файлами, используемыми клиентом (например, пакетами времени выполнения, библиотеками DLL, управляющими элементами ActiveX и т.д.).
2. Скопируйте файл MIDAS.DLL в каталог %SYSTEM%.
3. Дополнительно. Если в клиентском приложении задается свойство ServerName компонента TDispatchConnection или используется раннее связывание, необходимо зарегистрировать файл библиотеки типов сервера (TLB). Для этого воспользуйтесь утилитой <*каталог DELHI*>\BIN\TREGSVR.EXE (при желании это можно осуществить и программно).

Соглашения по установке приложений в Internet

Для установки приложения в локальной сети не существует никаких ограничений. Можно выбрать любой тип соединения, лучше всего соответствующий данному приложению. Однако при установке приложения в Internet может возникнуть множество препятствий, среди которых в первую очередь следует отметить брандмауэры.

Протокол DCOM является далеко не самым дружественным по отношению к брандмауэрам. Он требует открытия на брандмауэре нескольких портов. Большинство системных администраторов не торопятся предоставлять целый диапазон портов, поскольку это подталкивает хакеров к активным действиям. При использовании компонента TSocketConnection ситуация несколько улучшается, поскольку требуется открыть лишь один порт. Однако некоторые системные администраторы могут отказаться и от этого на том основании, что при этом нарушается система безопасности.

Компонент TWebConnection является производным от компонента TSocketConnection и позволяет преобразовать трафик MIDAS в поток протокола HTTP, использующего самый открытый порт в мире — порт HTTP (по умолчанию порт под номером 80). На самом деле этот компонент поддерживает даже протокол SSL, благодаря чему можно пользоваться безопасными соединениями и избежать проблем с брандмауэрами. В конце концов, если какая-либо корпорация не предоставит возможности соединения по протоколу HTTP, то связаться с ней не удастся никакими другими средствами.

Это небольшое чудо обеспечивается за счет использования ISAPI-расширения компании Borland, с помощью которого трафик HTTP можно преобразовать в поток MIDAS и наоборот. В этом отношении динамическая библиотека ISAPI будет работать точно так же, как и библиотека ScktSrvr для соединений типа сокетов. ISAPI-расширение, библиотека httpsrvr.dll, должна содержаться в каталоге, доступном для выполняемого кода. Например, в случае IIS4 для этого файла по умолчанию будет использоваться каталог C:\Inetpub\Scripts.

Еще одним преимуществом использования компонента TWebConnection является поддержка буферизации объектов. Благодаря этому экономятся ресурсы сервера, затрачиваемые на создание объектов при соединении с ним клиентов. Более того, механизм буферизации позволяет установить максимально возможное количество таких объектов. После достижения заданного максимального количества объектов, очередному клиенту будет послано сообщение об ошибке. В этом сообщении содержится информация о том, что сервер занят и не может обработать поступивший запрос клиента. Такой подход является гораздо более гибким, чем простое создание случайного числа потоков для каждого клиента, который хочет соединиться с сервером.

Если в приложении MIDAS используется буферизация удаленного модуля данных, то в его методе UpdateRegistry необходимо вызывать методы RegisterPooled и UnregisterPooled (пример реализации метода UpdateRegistry содержится в листинге 32.1). Метод RegisterPooled должен вызываться приблизительно так:

```
RegisterPooled(ClassID, 16, 30);
```

В этой строке приложению MIDAS сообщается, что в буфере может располагаться 16 объектов и любой экземпляр объекта будет удален, если он не был активен в течение 30 минут. Если вам не нужно удалять объекты, задайте этому параметру нулевое значение.

В клиентскую часть приложения не нужно вносить никаких коренных изменений. Просто используйте компонент TWebConnection вместо компонента TDispatchConnection, заполните его соответствующие свойства, и клиент будет готов взаимодействовать с сервером по протоколу HTTP. При использовании компонента TWebConnection вместо задания имени или адреса компьютера, на котором установлен сервер, нужно указать полный адрес URL файла httpsrvr.dll. На рис. 32.11 представлен фрагмент типичной установки с использованием компонента TWebConnection.

Еще одним преимуществом использования транспортного протокола HTTP является то, что операционная система типа Windows NT Enterprise позволяет объединять серверы в кластеры. При этом для серверной части приложения обеспечивается реальная отказоустойчивость и балансировка загрузки. Для получения более подробной информации о кластеризации обращайтесь к Web-странице по следующему адресу:

http://www.microsoft.com/ntserver/ntserverenterprise/exec/overview/clustering

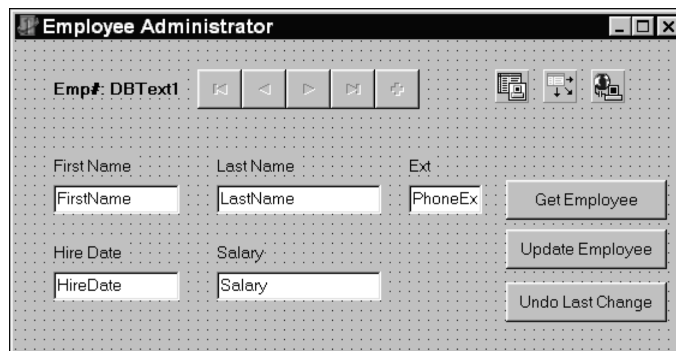


Рис. 32.11. Установка компонента `TWebConnection` *во время разработки*

Ограничения, вызываемые использованием компонента `TWebConnection` относительно невелики, при этом они с лихвой окупаются увеличением числа клиентов, которые могут обращаться к приложению сервера. Эти ограничения состоят в том, что на компьютере клиента необходимо установить файл `wininet.dll`, а при использовании компонента `TWebConnection` нельзя применять обратные вызовы. Кроме того, приложение сервера нужно зарегистрировать с помощью функции `EnableWebTransport` в переопределенном методе `UpdateRegistry`.

Резюме

В этой главе представлены краткие сведения о технологии MIDAS. Здесь лишь обозначены неисчерпаемые возможности этой технологии, рассмотрение всего разнообразия которых выходит за рамки одной главы. Даже изучив все нюансы MIDAS, можно постоянно совершенствовать свои знания и расширять возможности создаваемых приложений, используя C++ Builder и JBuilder. В среде JBuilder с использованием все той же технологии MIDAS и концепций, изученных в этой главе, можно реализовать доступ к серверу приложения с различных платформ и, тем самым, достичь истинного совершенства.

Технология MIDAS быстро развивается, и каждый программист теперь может выбрать средства создания многоуровневых приложений. Однажды ощутив подлинную мощь приложений MIDAS, вы уже никогда не вернетесь к разработке обычных приложений баз данных.

Быстрая разработка приложений баз данных

ЧАСТЬ

V

INVENTORY MANAGER: ПРИМЕР РАЗРАБОТКИ ПРИЛОЖЕНИЯ С АРХИТЕКТУРОЙ КЛИЕНТ/СЕРВЕР 778

ДИСПЕТЧЕР КЛИЕНТА: РАЗРАБОТКА ПРИЛОЖЕНИЯ ПО ТЕХНОЛОГИИ MIDAS 828

РАЗРАБОТКА НАСТОЛЬНОГО ПРИЛОЖЕНИЯ: СБОР СВЕДЕНИЙ ОБ ОШИБКАХ 850

ПРИЛОЖЕНИЕ С ИСПОЛЬЗОВАНИЕМ КОМПОНЕНТОВ WEBBROKER: СБОР СВЕДЕНИЙ ОБ ОШИБКАХ 877

Глава

33 Inventory Manager: пример разработки приложения с архитектурой клиент/сервер

Проектирование внутреннего интерфейса 779

Централизованный доступ к базе данных: реализация бизнес-правил 788

Создание пользовательского интерфейса 802

Резюме 827

В этой главе описываются процесс разработки приложения базы данных на основе концепций, рассмотренных в главе 29, “Разработка приложений архитектуры клиент/сервер”, и методы создания двухуровневого приложения типа клиент/сервер. Логика этого приложения (или бизнес-правила) разделена между клиентом и сервером. Кроме того, здесь обсуждаются методы централизации доступа к данным в модуле данных, позволяющие полностью отделить пользовательский интерфейс от логики работы с базой данных.

В главе 4, “Строение приложения и концепции конструирования” (том I), мы обсудили принципы построения форм, которые можно выводить либо независимо либо в качестве дочерних окон других элементов управления. В этой главе подобная структура применена к пользовательскому интерфейсу обсуждаемого приложения.

В качестве базовой СУБД в приложении используется Local InterBase. Приложение разработано с применением типичной бизнес-модели компании, занимающейся поставкой запчастей для автомобилей. Эта бизнес-модель требует, чтобы в приложении отслеживалось состояние трех основных наборов данных.

- *Ведомость товаров на складе.* В этом наборе данных фиксируются объемы каждого вида товара на складе и его стоимость.
- *Продажи.* Этот набор данных содержит информацию о проданных товарах и их покупателях.
- *Покупатель.* В этом наборе данных хранятся сведения о покупателях, в частности, их имена и адреса.

Данное приложение никоим образом не следует рассматривать как полный программный продукт, предназначенный для учета запасов на складе. Цель данной главы — сосредоточить ваше внимание на методах разработки баз данных типа клиент/сервер. Именно этим мы и руководствовались, принимая решение предоставить вам в качестве иллюстрации вполне работающее приложение.

Глава разделена на три раздела. В первом разделе, “Проектирование внутреннего интерфейса”, рассматривается проектирование внутреннего интерфейса на основе объектов базы данных, которые обсуждались в главе 29. Во втором разделе, “Централизованный доступ к базе данных: реализация бизнес-правил”, речь пойдет об использовании класса Delphi `TDataModule` для централизации доступа к базе данных. И наконец, в третьей части, “Создание пользовательского интерфейса”, обсуждаются вопросы проектирования реального пользовательского интерфейса для приложения учета запасов на складе.

Проектирование внутреннего интерфейса

В качестве базовой СУБД для обсуждаемого в этой главе приложения Inventory Manager используется продукт Local InterBase Server, созданный компанией InterBase Software Corporation. Это позволило разработать применяемую в приложении базу данных исключительно с помощью средств языка структурированных запросов (SQL). Благодаря высокой гибкости этого инструмента оказалось возможным переместить определенную часть обработки данных на сервер — за счет использования триггеров, генераторов и хранимых процедур. Кроме всего прочего, подобное решение более эффективно с точки зрения поддержки целостности данных. Еще одно существенное преимущество построения внутреннего интерфейса с использованием языка SQL заключается в том, что подобное решение существенно упрощает перенос приложения в реальную распределенную среду клиент/сервер.

На заметку

Некоторые нюансы работы с данными, присутствующие в этой главе, являются специфическими для СУБД InterBase и могут оказаться не применимыми в других SQL-средах разработки реляционных баз данных, таких как Oracle или Microsoft SQL. Однако рассматриваемые здесь концепции вполне приемлемы — просто для их реализации потребуется использовать иные средства.

Итак, для создания различных объектов базы данных, необходимых приложению Inventory Manager, будет использован язык SQL. Сюда относятся такие объекты, как домены, таблицы, генераторы, триггеры, хранимые процедуры и права доступа.

Существует несколько способов создания внутреннего интерфейса с помощью инструментов *моделирования данных*. Использование приложений xCase, RoboCase, Erwin и SQL-Designer значительно упрощает процесс моделирования данных. С их помощью можно моделировать данные визуально, т.е. без ввода громоздких операторов языка SQL. После создания базовой модели данных в нее можно будет внести изменения (при необходимости).

На рис. 33.1 изображена модель данных нашего приложения.

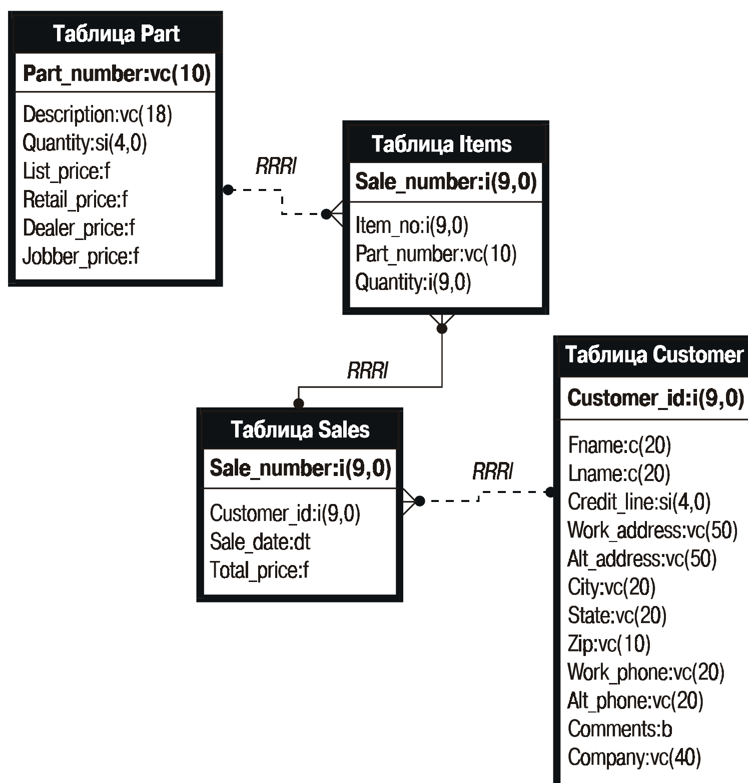


Рис. 33.1. Модель данных приложения учета движения товаров на складе

Определение доменов

Прежде чем определять любые таблицы, триггеры и тому подобные объекты, следует определить домены, которые будут использоваться в операторах языка SQL, определяющих *метаданные* базы данных.

На заметку

Под *метаданными* понимаются все объекты (таблицы, индексы и т.п.), предоставляемые как часть определения базы данных.

Представьте себе *домен* как элемент, аналогичный определяемому пользователем типу данных в языке Object Pascal. С помощью доменов можно определять специальные типы данных, структура которых отличается от встроенных типов данных СУБД.

Домены помогают упростить объявление данных и установку ограничений, позволяя присвоить удобные имена тем типам данных, которые широко используются в конкретной базе данных. Следует отметить, что после применения домена в хотя бы одном из столбцов любой таблицы, изменить его будет уже нельзя.

Перечислим некоторые домены, обычно используемые в метаданных, связанных с продажами.

- `CREATE DOMAIN DCUSTOMERID AS INTEGER;`

  Это пример очень простого домена. Определяется новый домен под именем `DCUSTOMERID` как тип, идентичный обычному стандартному целому.

- `CREATE DOMAIN DCREDITLINE AS SMALLINT default 0 CHECK (VALUE BETWEEN 0 AND 3000);`

  Здесь определяется новый домен типа `SMALLINT`, но к нему применяется дополнительное ограничение, состоящее в том, что значение должно лежать в диапазоне чисел от `0` до `3000`.

- `CREATE DOMAIN DNAME AS CHAR(20);`

  Определяется домен под именем `DNAME`, который представляет собой строку фиксированной длины, рассчитанной ровно на 20 символов.

- ```
  CREATE DOMAIN DADDRESS AS VARCHAR(50);
  CREATE DOMAIN DCITY AS VARCHAR(20);
  CREATE DOMAIN DSTATE AS VARCHAR(20);
  CREATE DOMAIN DZIP AS VARCHAR(10);
  CREATE DOMAIN DPHONE AS VARCHAR(20);
  ```

  Здесь определено несколько доменов в виде строк переменной длины, максимальная длина которых составляет 50, 20, 20, 10 и 20 символов соответственно.

- `CREATE DOMAIN DPRICE AS NUMERIC(15, 2) default 0.00;`

  Создается домен, представляющий собой десятичное число. Первое число (15) задает точность сохранения, а второе (2) — количество дробных десятичных знаков. По умолчанию значение для столбцов этого домена будет равно `0.00`.

На заметку

Тип данных `CHAR(n)` используется для хранения в базе данных точно *n* символов. Если значение, содержащееся в конкретном поле, равно меньше чем *n* символам, то неиспользуемые позиции заполняются символами пробела.

Тип данных `VARCHAR(n)` хранит количество символов, соответствующее размеру строки, не превышающему максимального значения *n*. Хотя этот тип данных более рационально использует память, за это приходится расплачиваться более медленным выполнением операций.

Дополнительную информацию о доменах можно найти в справочном руководстве "InterBase Language Reference Guide" или в файле `IB32.Hlp` интерактивной справочной системы.
  ```

Определение таблиц

Определив домены, можно приступать к созданию таблицы. Каждая таблица создается с помощью SQL-оператора CREATE TABLE, содержащего перечисление полей таблицы с указанием их типа данных или доменов.

Таблица CUSTOMER

Таблица CUSTOMER представляет собой объект данных, предназначенный для хранения информации о покупателях, и определяется следующим образом:

```
/* Таблица: CUSTOMER, Владелец: SYSDBA */
CREATE TABLE CUSTOMER (CUSTOMER_ID INTEGER NOT NULL,
        FNAME DNAME NOT NULL,
        LNAME DNAME NOT NULL,
        CREDIT_LINE DCREDITLINE NOT NULL,
        WORK_ADDRESS DADDRESS,
        ALT_ADDRESS DADDRESS,
        CITY DCITY,
        STATE DSTATE,
        ZIP DZIP,
        WORK_PHONE DPHONE,
        ALT_PHONE DPHONE,
        COMMENTS BLOB SUB_TYPE TEXT SEGMENT SIZE 80,
        COMPANY VARCHAR(40),
CONSTRAINT PCUSTOMER_ID PRIMARY KEY (CUSTOMER_ID));
```

Указание спецификатора NOT NULL в определении поля означает, что пользователь должен обязательно ввести его значение, прежде чем запись можно будет поместить в таблицу. Иными словами, эти поля никогда не должны оставаться пустыми.

Поле COMMENTS требует некоторых пояснений. Оно имеет тип BLOB (Binary Large Object — Двоичный объект большого размера), который допускает сохранение в этом поле данных любого типа, представленных в произвольном формате. Однако наличие определения подтипа SUB_TYPE TEXT указывает на то, что помещаемые в это поле данные всегда будут представлять собой текст ASCII и, следовательно, совместимы с компонентом Delphi TDBMemo.

С помощью инструкции CONSTRAINT создается первичный ключ на основе поля CUSTOMER_ID, который гарантирует, что значение этого поля в каждой записи таблицы будет уникальным. Это также является первым шагом к обеспечению ссылочной целостности создаваемой базы данных. Как будет показано ниже, поле типа PRIMARY KEY используется как поле подстановки для поля типа FOREIGN KEY, определенного в другой таблице.

Таблица PART

Таблица PART представляет собой ведомость товаров на складе. Ее структура довольно проста:

```
/* Таблица: PART, Владелец: SYSDBA */
CREATE TABLE PART (PART_NUMBER VARCHAR(10) NOT NULL,
        DESCRIPTION VARCHAR(18),
        QUANTITY SMALLINT NOT NULL,
```

```
        LIST_PRICE DPRICE NOT NULL,
        RETAIL_PRICE DPRICE NOT NULL,
        DEALER_PRICE DPRICE NOT NULL,
        JOBBER_PRICE DPRICE NOT NULL,
CONSTRAINT PPART_NUMBER PRIMARY KEY (PART_NUMBER));
```

Каждая запись содержит информацию об одной уникальной запасной части, включая описание, количество и сведения о стоимости. Обратите внимание, что эта таблица также имеет первичный ключ — на этот раз им служит поле PART_NUMBER.

Таблица SALES

В таблице SALES содержатся сведения о каждой операции продажи товаров некоторому покупателю. Она определяется следующим образом:

```
/* Таблица: SALES, Владелец: SYSDBA */
CREATE TABLE SALES (SALE_NUMBER INTEGER,
        CUSTOMER_ID INTEGER,
        SALE_DATE DATE,
        TOTAL_PRICE DOUBLE PRECISION);

ALTER TABLE SALES ADD FOREIGN KEY (CUSTOMER_ID)
      REFERENCES CUSTOMER(CUSTOMER_ID);
```

Обратите внимание на инструкцию ALTER TABLE, которая добавляет в таблицу SALES внешний ключ. *Внешний ключ* (foreign key) — это столбец или набор столбцов в одной таблице, который в точности соответствует столбцу (или набору столбцов), определенному в качестве первичного ключа в другой таблице. Использование внешнего ключа в таблице SALES позволяет обеспечить ее ссылочную целостность, а это гарантирует, что любая запись в данной таблице будет создана только в том случае, если значение ее поля CUSTOMER_ID имеет соответствие в поле CUSTOMER_ID некоторой записи таблицы CUSTOMER.

Таблица ITEMS

В таблице ITEMS содержатся сведения о товарах (запчастях), отпущенных покупателю по определенной накладной (отдельной операции продажи). Таблица SALES связана с таблицей ITEMS отношением типа "один ко многим", и эта связь реализуется через поля SALE_NUMBER и SALE_NO в каждой таблице. Определение таблицы ITEMS имеет следующий вид:

```
/* Таблица: ITEMS, Владелец: SYSDBA */
CREATE TABLE ITEMS (SALE_NUMBER INTEGER,
        ITEM_NO INTEGER,
        PART_NO VARCHAR(10),
        QTY SMALLINT);

ALTER TABLE ITEMS ADD FOREIGN KEY (PART_NO)
      REFERENCES PART(PART_NUMBER);
```

Подобно таблице SALES, таблица ITEMS имеет внешний ключ, гарантирующий, что в нее можно будет ввести только такие записи, в которых значение поля PART_NO будет соответствовать значению поля PART_NUMBER некоторой записи таблицы PART.

Определение генераторов

Генератор можно представить в виде механизма автоматической генерации уникальных последовательных чисел, предназначенных для вставки в таблицу. Генераторы часто используются для создания уникальных значений, вставляемых в ключевое поле таблицы. В базе данных SALES генераторы будут использоваться для автоматической генерации нового уникального идентификатора (ID) покупателя для таблиц CUSTOMER, SALES и ITEMS. Эти генераторы определяются следующим образом:

```
CREATE GENERATOR GEN_CUSTID;
CREATE GENERATOR GEN_ITEMNO;
CREATE GENERATOR GEN_SALENO;
```

На заметку

После добавления в базу данных генератора, просто так удалить его будет нельзя. Проще всего удалить или так модифицировать триггеры или хранимые процедуры, чтобы в них не вызывался соответствующий GEN_ID(). Затем можно будет удалить генератор из системной таблицы RDB$GENERATORS.

Определение триггеров

Триггер — это процедура, которая автоматически выполняет некоторое действие при вставке, обновлении или удалении записи из таблицы. Благодаря триггерам в базах данных могут автоматически выполняться некоторые повторяющиеся операции, запускаемые при фиксации изменений в таблице. В результате упрощается алгоритм приложений, используемых для доступа или модификации данных.

На заметку

Триггеры и генераторы — это специфические функции СУБД InterBase. Несмотря на то что большинство изготовителей SQL-серверов предлагают подобные возможности, в своих реализациях они, скорее всего, используют другой синтаксис и семантику. При этом вы должны иметь в виду, что, хотя триггеры и генераторы — очень хорошие средства, они могут оказаться "узким местом" при переводе приложения на SQL-сервер, отличный от InterBase.

Для начала нам понадобятся триггеры, которые с помощью созданных ранее генераторов будут добавлять в соответствующие таблицы заготовки записей о новом покупателе и очередной торговой сделке. Триггер, предназначенный для вставки записи с новым уникальным идентификатором покупателя, может выглядеть так:

```
CREATE TRIGGER TCUSTOMER_ID FOR CUSTOMER
ACTIVE BEFORE INSERT POSITION 0
as begin
  new.customer_id = gen_id(gen_custid, 1);
end
```

Следующий триггер аналогичным образом работает с таблицей ITEMS:

```
CREATE TRIGGER TITEM_NO FOR ITEMS
ACTIVE BEFORE INSERT POSITION 0
as begin
  new.item_no = gen_id(gen_itemno, 1);
end
```

На заметку

В рассматриваемой базе данных используются и другие триггеры, с помощью которых двухбуквенная аббревиатура штата преобразуется в его полное название. Эти триггеры можно найти на прилагаемом компакт-диске, в файле `Sales.ddl`, расположенном в каталоге, соответствующем данной главе.

Определение хранимых процедур

Хранимая процедура — это самостоятельная программа на языке SQL, сохраняемая на сервере как часть метаданных базы данных.

Хранимую процедуру можно вызывать, причем в результате своего выполнения она может возвращать некоторый набор данных — аналогично обычному запросу. Преимущества хранимых процедур состоят в том, что они сокращают объем обработки, требуемой со стороны клиента, уменьшают нагрузку на сеть и позволяют централизовать выполнение некоторых функций. Кроме того, хранимые процедуры могут способствовать повышению производительности, поскольку они представляют собой заранее откомпилированный SQL-код, выполняемый непосредственно на сервере, без необходимости пересылки его по сети. Более подробно хранимые процедуры обсуждались в главе 29, “Разработка приложений архитектуры клиент/сервер”.

В базе данных `SALES` используются две хранимые процедуры. Первая, `INSERT_SALE`, предназначена для вставки записи о продаже в таблицу `SALES`. Этой хранимой процедуре передается три параметра: идентификатор покупателя, дата продажи и общая стоимость проданного товара. Процедура возвращает идентификатор операции продажи, сгенерированный внутри хранимой процедуры. Клиентское приложение передает это возвращаемое значение в другую хранимую процедуру, где оно используется в качестве внешнего ключа для таблицы `ITEMS`. Текст хранимой процедуры `INSERT_SALE` представлен в листинге 33.1.

Листинг 33.1. Хранимая процедура `INSERT_SALE`

```
CREATE PROCEDURE INSERT_SALE AS BEGIN EXIT; END ^
...
ALTER PROCEDURE INSERT_SALE (
  ICUSTOMER_ID  INTEGER,
  ISALE_DATE    DATE,
  ITOTAL_PRICE  DOUBLE PRECISION)
RETURNS(
  RSALE_NUMBER  INTEGER)
AS
BEGIN
  /* Сначала получаем новый идентификатор Sale из генератора   */
  /* GEN_SALENO. Это значение сохраняется в параметре rSale,   */
  /* определенном как возвращаемое значение, и, следовательно, */
  /* оно будет возвращено вызвавшему процедуру клиенту.        */
  rSALE_NUMBER = gen_id(GEN_SALENO, 1);
  /* Теперь вставляем запись в таблицу SALES */
  INSERT INTO SALES(
    SALE_NUMBER,
    CUSTOMER_ID,
    SALE_DATE,
    TOTAL_PRICE)
  VALUES(
```

```
    :rSALE_NUMBER,
    :iCUSTOMER_ID,
    :iSALE_DATE,
    :iTOTAL_PRICE);
END
```

Эта хранимая процедура выполняет очень простую последовательность SQL-операторов. Сначала из генератора GEN_SALENO считывается новый идентификатор для записи операции продажи. Затем выполняется простой SQL-оператор INSERT INTO, предназначенный для вставки данных, переданных процедуре через параметры.

Вторая хранимая процедура (INSERT_SALE_ITEM) сложнее и используется приложением несколько по-другому — для вставки в таблицу ITEMS данных об отдельном элементе операции продажи. По всей вероятности, для одной типичной операции продажи эта хранимая процедура будет вызываться несколько раз. Таким образом, клиент начинает обработку каждой операции продажи с вызова хранимой процедуры INSERT_SALE, которая создает для этой операции новую записи и возвращает клиенту ее идентификатор. Затем для каждого продаваемого товара клиент будет вызывать хранимую процедуру INSERT_SALE_ITEM. При каждом вызове он должен передать процедуре информацию о конкретном товаре, включая и полученный ранее идентификатор операции продажи.

В хранимую процедуру INSERT_SALE_ITEM передаются три параметра: идентификатор продажи, номер запчасти и количество продаваемых запчастей. В этой хранимой процедуре выполняется несколько операций, обеспечивающих целостность данных. Прежде всего по таблице PART проверяется наличие запрашиваемого числа элементов на складе (если их недостаточно, генерируется исключительная ситуация). Если имеющегося на складе количества запчастей указанного вида достаточно для удовлетворения заказа покупателя, то из количества запчастей в таблице PART вычитается значение параметра Qty. После этого информация о проданном элементе добавляется в таблицу ITEMS.

Текст хранимой процедуры INSERT_SALE_ITEM представлен в листинге 33.2.

Листинг 33.2. Хранимая процедура INSERT_SALE_ITEM

```
CREATE PROCEDURE INSERT_SALE_ITEM AS BEGIN EXIT; END ^
...
ALTER PROCEDURE INSERT_SALE_ITEM (
  ISALE_NUMBER  INTEGER,
  IPART_NO      VARCHAR(10),
  IQTY          SMALLINT)
AS
  DECLARE VARIABLE Actual_Qty VARCHAR(10);
BEGIN
/* Проверяем наличие необходимого количества деталей iQTY в таблице PARTS */
  SELECT QUANTITY FROM PART
    WHERE PART_NUMBER = :iPART_NO
    INTO Actual_Qty;
  IF (Actual_Qty < iQTY) THEN
    EXCEPTION EXP_EXCESS_ORDER;
  ELSE BEGIN
    /* Удаляем количество проданных запчастей из таблицы PART */
    UPDATE PART
```

```
    SET QUANTITY = (:Actual_Qty - :iQty)
    WHERE PART_NUMBER = :iPART_NO;
    /* Теперь размещаем новый заказ */
    INSERT INTO ITEMS(
      SALE_NUMBER,
      PART_NO,
      QTY)
    VALUES(
      :iSALE_NUMBER,
      :iPART_NO,
      :iQTY);
  END
END
```

На заметку

Если для ввода метаданных вы используете инструмент ISQL, то вам придется изменить символ завершения. Поскольку все команды внутри процедуры должны завершаться точкой с запятой (;), которая является также и символом завершения команды в языке ISQL, во избежание конфликта следует изменить символ завершения ISQL-команд. Это осуществляется с помощью команды SET TERM.

В базе данных SALES в качестве символа завершения используется символ ^, и следующая ISQL-команда обеспечит соответствующую замену:

```
SET TERM ^ ;
```

Предоставление прав доступа

Заключительный этап определения базы данных — предоставление прав доступа к таблицам и хранимым процедурам отдельным пользователям. С помощью следующей инструкции можно предоставить всем пользователям право на выполнение команд SELECT и UPDATE в отношении таблицы CUSTOMER:

```
GRANT SELECT, UPDATE ON CUSTOMER TO PUBLIC WITH GRANT OPTION;
```

Используя следующую инструкцию, можно предоставить все права на работу с таблицей SALE:

```
GRANT ALL ON SALE TO PUBLIC WITH GRANT OPTION;
```

Фраза GRANT OPTION означает, что тем, кому этим оператором разрешается доступ к указанной таблице, одновременно предоставляется и право разрешать доступ к этой таблице другим пользователям. Инструкции GRANT, используемые для таблиц и хранимых процедур приложения Inventory Manager, имеют следующий вид:

```
/* Предоставление прав доступа к объектам базы данных */
GRANT SELECT, UPDATE ON CUSTOMER TO PUBLIC WITH GRANT OPTION;
GRANT ALL ON SALES TO PUBLIC WITH GRANT OPTION;
GRANT ALL ON PART TO PUBLIC WITH GRANT OPTION;
GRANT ALL ON ITEMS TO PUBLIC WITH GRANT OPTION;
GRANT EXECUTE ON PROCEDURE INSERT_SALE TO PUBLIC;
GRANT EXECUTE ON PROCEDURE INSERT_SALE_ITEM TO PUBLIC;
```

В следующем разделе показано, как связать между собой объекты этой базы данных.

Централизованный доступ к базе данных: реализация бизнес-правил

В этом разделе вы узнаете, как отделить логику доступа к базе данных и поддержку бизнес-правил от функций пользовательского интерфейса. Это позволит достичь сразу нескольких целей. Во-первых, помещение всей поддержки бизнес-правил в один модуль данных упрощает ее сопровождение — поскольку она располагается в одном месте, отпадает необходимость поиска отдельных фрагментов по всему приложению. Во-вторых, такой подход позволяет легко перейти от двух- к трехуровневой модели путем добавления соответствующих компонентов в модуль данных, который уже содержит всю логику поддержки бизнес-правил. Здесь мы этого не делаем, но хотим обратить ваше внимание на данный факт, поскольку он безусловно заслуживает серьезного рассмотрения при проектировании двухуровневых систем.

Класс `TDataModule` всегда следует использовать таким образом, чтобы охватить как можно больше элементов, связанных с самой базой данных. Как это сделать, будет показано на примере приложения Inventory Manager.

В нашем демонстрационном приложении использован один экземпляр класса `TDataModule`. Для небольших приложений этого вполне достаточно; для больших же приложений имеет смысл использовать несколько экземпляров класса `TDataModule` и распределить между ними функции, руководствуясь логикой и здравым смыслом.

В листинге 33.3 представлен исходный код класса `TDDGSalesDataModule`, определенный в модуле `SalesDM.pas`.

Листинг 33.3. Класс `TDDGSalesDataModule`

```
unit SalesDM;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  DBTables, Db;

type
  TDDGSalesDataModule = class(TDataModule)
    qryCustomer: Tquery;
    dbSales: Tdatabase;
    usqlCustomer: TUpdateSQL;
    qryCustomerCUSTOMER_ID: TIntegerField;
    qryCustomerFNAME: TStringField;
    qryCustomerLNAME: TStringField;
    qryCustomerCREDIT_LINE: TSmallintField;
    qryCustomerWORK_ADDRESS: TStringField;
    qryCustomerALT_ADDRESS: TStringField;
    qryCustomerCITY: TStringField;
    qryCustomerSTATE: TStringField;
    qryCustomerZIP: TStringField;
    qryCustomerWORK_PHONE: TStringField;
    qryCustomerALT_PHONE: TStringField;
```

```
    qryCustomerCOMMENTS: TMemoField;
    qryCustomerCOMPANY: TStringField;
    qryParts: Tquery;
    usqlParts: TUpdateSQL;
    qryPartsPART_NUMBER: TStringField;
    qryPartsDESCRIPTION: TStringField;
    qryPartsQUANTITY: TSmallintField;
    qryPartsLIST_PRICE: TFloatField;
    qryPartsRETAIL_PRICE: TFloatField;
    qryPartsDEALER_PRICE: TFloatField;
    qryPartsJOBBER_PRICE: TFloatField;
    spInsertSaleItem: TStoredProc;
    spInsertSale: TStoredProc;
    qryTotalPrice: Tquery;
    tblTempItems: Ttable;
    tblTempItemsPART_NUMBER: TStringField;
    tblTempItemsDESCRIPTION: TStringField;
    tblTempItemsQUANTITY: TSmallintField;
    tblTempItemsRETAIL_PRICE: TFloatField;
    tblTempItemsTOTAL_PRICE: TFloatField;
    qryTotalPriceSUMOFTOTAL_PRICE: TFloatField;
    qrySale: Tquery;
    dsCustomer: TDataSource;
    qryItems: Tquery;
    dsSale: TDataSource;
    qrySaleSALE_NUMBER: TIntegerField;
    qrySaleSALE_DATE: TDateTimeField;
    qrySaleTOTAL_PRICE: TFloatField;
    qryItemsDESCRIPTION: TStringField;
    qryItemsQTY: TSmallintField;
    qryCustomerSearch: Tquery;
    procedure tblTempItemsBeforePost(DataSet: TDataSet);
    procedure dbSalesLogin(Database: Tdatabase; LoginParams: Tstrings);
  protected
    procedure SetAfterTempItemsChange(Value: TDataSetNotifyEvent);
    function GetAfterTempItemsChange: TDataSetNotifyEvent;
  public

    // Методы подключения

    procedure Logout;
    function Login: Boolean;
    function Connect: Boolean;
    procedure Disconnect;

    // Методы, связанные с покупателями
    procedure FirstCustomer;
    procedure LastCustomer;
    procedure NextCustomer;
    procedure PrevCustomer;
    procedure EditCustomer;
```

```
    procedure NewCustomer;
    procedure AcceptCustomer;
    procedure CancelCustomer;
    procedure DeleteCustomer;
    function IsFirstCustomer: Boolean;
    function IsLastCustomer: Boolean;
    function GetCustomerName: String;
    function SearchForCustomer: Boolean;

    // Методы, связанные с запчастями
    procedure FirstPart;
    procedure LastPart;
    procedure NextPart;
    procedure PrevPart;
    procedure EditPart;
    procedure NewPart;
    procedure AcceptPart;
    procedure CancelPart;
    procedure DeletePart;
    function IsFirstPart: Boolean;
    function IsLastPart: Boolean;
    function SearchForPart: Boolean;

    // Методы, связанные с продажами

    procedure AddItemToSale;
    procedure SaveSale;
    procedure CancelSale;
    function SaleItemsTotalPrice: double;
    procedure OpenTempItems;
    procedure CloseTempItems;

    // Внешние свойства
    property AfterTempItemsChange: TDataSetNotifyEvent
        read GetAfterTempItemsChange
      write SetAfterTempItemsChange;

  end;

var
  DDGSalesDataModule: TDDGSalesDataModule;

implementation

uses CustomerSrchFrm, LoginFrm;

{ $R *.DFM}

procedure TDDGSalesDataModule.SetAfterTempItemsChange(Value:
               TDataSetNotifyEvent);
begin
```

```
  { Этот метод записи добавляет параметр Value к событиям AfterPost и
    AfterDelete временной таблицы элементов. Это гарантирует, что при любом
    изменении данных обработчик событий будет обязательно вызван. }
  tblTempItems.AfterPost   := Value;
  tblTempItems.AfterDelete := Value;
end;

function TDDGSalesDataModule.GetAfterTempItemsChange: TDataSetNotifyEvent;
begin
  Result := tblTempItems.AfterPost;
end;

// Методы регистрации в системе

procedure TDDGSalesDataModule.dbSalesLogin(Database: Tdatabase;
  LoginParams: Tstrings);
begin
  { Вызывает следующий метод, чтобы заполнить список строк LoginParams
    регистрационными данными пользователя. Метод GetLoginParams определен
    в модуле LoginFrm.pas. }
  GetLoginParams(LoginParams);
end;

procedure TDDGSalesDataModule.Logout;
begin
  Disconnect;
end;

function TDDGSalesDataModule.Login: Boolean;
begin
  Result := Connect;
end;

function  TDDGSalesDataModule.Connect: Boolean;
begin
  { Подключает пользователя к базе данных. Если свойство dbSales.Connected
    установлено равным True, будет вызван обработчик события OnLogon,
    который активизирует диалоговое окно регистрации покупателя,
    определенное в модуле LoginFrm.pas. }
  try
    dbSales.Connected  := True;
    qryCustomer.Active := True;
    qryParts.Active    := True;
    qrySale.Active     := True;
    qryItems.Active    := True;
    Result := True;
  except
    MessageDlg('Invalid Password or login information, cannot login.',
       mtError, [mbok], 0);
     { Неверный пароль или информация регистрации, поэтому вход не разрешен. }
    dbSales.Connected := False;
```

```
    Result := False;
  end;
end;

procedure TDDGSalesDataModule.Disconnect;
begin
  // Отключение от базы данных
  dbSales.Connected := False;
end;

// Методы, связанные с покупателями

procedure TDDGSalesDataModule.AcceptCustomer;
begin
  dbSales.ApplyUpdates([qryCustomer]);
end;

procedure TDDGSalesDataModule.CancelCustomer;
begin
  qryCustomer.CancelUpdates;
end;

procedure TDDGSalesDataModule.DeleteCustomer;
begin
  qryCustomer.Delete;
end;

procedure TDDGSalesDataModule.EditCustomer;
begin
  qryCustomer.Edit;
end;

procedure TDDGSalesDataModule.FirstCustomer;
begin
  qryCustomer.First;
end;

procedure TDDGSalesDataModule.LastCustomer;
begin
  qryCustomer.Last;
end;

procedure TDDGSalesDataModule.NewCustomer;
begin
  qryCustomer.Insert;
end;

procedure TDDGSalesDataModule.NextCustomer;
begin
  qryCustomer.Next;
end;
```

```
procedure TDDGSalesDataModule.PrevCustomer;
begin
  qryCustomer.Prior;
end;

function TDDGSalesDataModule.IsFirstCustomer: Boolean;
begin
  Result := qryCustomer.Bof;
end;

function TDDGSalesDataModule.IsLastCustomer: Boolean;
begin
  Result := qryCustomer.Eof;
end;

function TDDGSalesDataModule.GetCustomerName: String;
begin
  { Обычно возвращается имя компании. Если имени компании не существует,
    возвращается имя покупателя. }
  if qryCustomerCOMPANY.AsString <> EmptyStr then
    Result := qryCustomerCOMPANY.AsString
  else
    Result := Format('%s %s', [qryCustomerFNAME.AsString,
               qryCustomerLNAME.AsString]);
end;

function TDDGSalesDataModule.SearchForCustomer: Boolean;
var
  CustID: Integer;
  SearchQry: String;
begin
  // Предполагаем неудачу
  Result := False;
  { Вызываем функцию SearchCustomer, которая определена в модуле
    CustomerSrchFrm.pas. Эта функция возвращает строку запроса, которая
    добавляется к экземпляру qryCustomerSearch компонента Tquery. }
  SearchQry := SearchCustomer;
  if SearchQry <> EmptyStr then
  begin
    Screen.Cursor := crSQLWait;
    try
      qryCustomerSearch.Close;
      qryCustomerSearch.SQL.Clear;
      qryCustomerSearch.SQL.Add(SearchQry);
      qryCustomerSearch.Open;
      try

        // Если запись не найдена, выходим из этого метода
        if qryCustomerSearch.FieldByName('CUSTOMER_ID').IsNull then
        begin
          Screen.Cursor := crDefault;
```

```
        Exit;
      end;

      { Если запись найдена, получаем ID покупателя, который используется
        для поиска записи в реальном экземпляре qryCustomer компонента
        Tquery. При этом курсор устанавливается в позицию найденной записи. }
      CustID := qryCustomerSearch.FieldByName('CUSTOMER_ID').AsInteger;

      { Если запись не найдена в запросе qryCustomer, значит, в этой базе
       данных существует некоторое несоответствие, поэтому генерируем
       исключительную ситуацию. }
      if not qryCustomer.Locate('CUSTOMER_ID', CustID, []) then
        raise Exception.Create('Inconsistency in database.')
          // Несоответствие в базе данных
      else
        Result := True;
    finally
      qryCustomerSearch.Close;
    end;
  finally
    Screen.Cursor := crDefault;
  end;
 end
 else
   Result := False;;
end;

// Методы, связанные с запчастями

function TDDGSalesDataModule.IsFirstPart: Boolean;
begin
  Result := qryParts.Bof;
end;

function TDDGSalesDataModule.IsLastPart: Boolean;
begin
  Result := qryParts.Eof;
end;

procedure TDDGSalesDataModule.AcceptPart;
begin
  dbSales.ApplyUpdates([qryParts]);
end;

procedure TDDGSalesDataModule.CancelPart;
begin
  qryParts.CancelUpdates;
end;

procedure TDDGSalesDataModule.DeletePart;
begin
```

```
  qryParts.Delete;
end;

procedure TDDGSalesDataModule.EditPart;
begin
  qryParts.Edit;
end;

procedure TDDGSalesDataModule.FirstPart;
begin
  qryParts.First;
end;

procedure TDDGSalesDataModule.LastPart;
begin
  qryParts.Last;
end;

procedure TDDGSalesDataModule.NewPart;
begin
  qryParts.Insert;
end;

procedure TDDGSalesDataModule.NextPart;
begin
  qryParts.Next;
end;

procedure TDDGSalesDataModule.PrevPart;
begin
  qryParts.Prior;
end;

function TDDGSalesDataModule.SearchForPart: Boolean;
{  Этот метод ищет запасную часть на основе ID, заданного пользователем. }
var
  PartNumber: string;
begin
  Result := False;
  PartNumber := '';
  if InputQuery('Part Search', 'Enter a Part Number', PartNumber) then
    if not qryParts.Locate('PART_NUMBER', PartNumber, []) then
      Exit
    else
      Result := True;
end;

// Методы, связанные с операциями продажи

procedure TDDGSalesDataModule.AddItemToSale;
begin
```

```
  { tblTempItems - это временная таблица, используемая для хранения
    элементов, добавляемых в продажу. Если пользователь сохраняет строку со
    сведениями о продаже, эти записи будут использованы в вызовах хранимых
    процедур, которые реально сохранят информацию о продаже в базе данных. }
  if not tblTempItems.Locate('PART_NUMBER',
    qryParts.FieldByName('PART_NUMBER').AsString, []) then
  begin
    tblTempItems.Insert;
    try
      tblTempItems['PART_NUMBER']  := qryParts['PART_NUMBER'];
      tblTempItems['DESCRIPTION']  := qryParts['DESCRIPTION'];
      tblTempItems['QUANTITY']     := 1;
      tblTempItems['RETAIL_PRICE'] := qryParts['RETAIL_PRICE'];
      tblTempItems.Post;
    except
      tblTempItems.Cancel;
    end;
  end
  else
    MessageDlg('Item already in list', mtWarning, [mbok], 0);
    // Элемент уже есть в списке
end;

procedure TDDGSalesDataModule.CancelSale;
begin
  { Если пользователь отменяет продажу, элементы, которые были
    добавлены в таблицу tblTempItems, нужно удалить. }
  tblTempItems.Close;
  tblTempItems.EmptyTable;
  tblTempItems.Open;
end;

procedure TDDGSalesDataModule.SaveSale;
var
  SaleNo: Integer;
begin
  { Если пользователь сохраняет сведения о продаже, то сначала создается
    запись в таблице продаж, ключ которой будет помещен в переменную
    SaleNo. Он используется в качестве ключа для связи с данными об элементах
    продажи, которые добавляются на следующем этапе. Элементы продажи
    считываются из временной таблицы tblTempItems. }
  dbSales.StartTransaction;
  try
    { Сначала создаем общую запись о продаже. }
    with spInsertSale do
    begin
      ParamByName('iCUSTOMER_ID').AsInteger := qryCustomer['CUSTOMER_ID'];
      ParamByName('iSALE_DATE').AsDateTime  := Now;
      ParamByName('iTOTAL_PRICE').AsFloat   := SaleItemsTotalPrice;
      ExecProc;
      // Получаем значение ключа в переменной SaleNo
```

```
      SaleNo := ParamByName('rSALE_NUMBER').AsInteger;
    end;

    { Теперь обрабатываем все записи в таблице tblTempItems для продажи,
    номер которой задан переменной SaleNo. }
    tblTempItems.First;
    while not tblTempItems.Eof do
    begin
      with spInsertSaleItem do
      begin
        ParamByName('IPART_NO').AsString      := tblTempItems['PART_NUMBER'];
        ParamByName('IQTY').AsInteger         := tblTempItems['QUANTITY'];
        ParamByName('ISALE_NUMBER').AsInteger := SaleNo;
        ExecProc;
      end;
      tblTempItems.Next;
    end;

    dbSales.Commit;

    // Обновляем модифицированные таблицы
    qryParts.Close;
    qryParts.Open;

    tblTempItems.Close;
    tblTempItems.EmptyTable;
    tblTempItems.Open;

  except
    dbSales.Rollback;
  end;
end;

function TDDGSalesDataModule.SaleItemsTotalPrice: double;
begin
  { Запрос qryTotalPrice считывает общую стоимость для всех записей,
    добавленных в таблицу tblTempItems. С помощью этого модуля данных
    метод может быть вызван из любой формы. }
  qryTotalPrice.Close;
  qryTotalPrice.Open;
  try
    Result := qryTotalPrice.FieldByName('SUM OF TOTAL_PRICE').AsFloat;
  finally
    qryTotalPrice.Close;
  end;
end;

procedure TDDGSalesDataModule.tblTempItemsBeforePost(DataSet: TDataSet);
begin
  { Перед передачей записи во временную таблицу вычисляем общую стоимость
    для поля TOTAL_PRICE по всем элементам, добавленным пользователем. }
```

```
  tblTempItemsTOTAL_PRICE.ReadOnly := False;
  try
    tblTempItems['TOTAL_PRICE'] := tblTempItems['RETAIL_PRICE'] *
      tblTempItems['QUANTITY'];
  finally
    tblTempItemsTOTAL_PRICE.ReadOnly := True;
  end;
end;

procedure TDDGSalesDataModule.OpenTempItems;
begin
  tblTempItems.Close;
  tblTempItems.EmptyTable;
  tblTempItems.Open;
end;

procedure TDDGSalesDataModule.CloseTempItems;
begin
  tblTempItems.Active := False;
end;

end.
```

Класс TDDGSalesDataModule включает экземпляр компонента Tdatabase под именем dbSales, всевозможные запросы (экземпляры компонента Tquery), а также компоненты типа TUpdateSQL и TStoredProc, необходимые для функционирования приложения Inventory Manager.

Компонент dbSales — это главное связующее звено с внутренним SQL-интерфейсом в базе данных Sales.gdb. Эта связь реализуется через псевдоним DDGSALES, который устанавливается с помощью программы DBExplorer. Компонент dbSales назначает псевдоним DDGSalesDB на уровне приложения. Первоначально его свойство Connected устанавливается равным False, чтобы все таблицы, принадлежащие этому компоненту, также были закрыты при первом запуске приложения. Компонент dbSales содержит обработчик события OnLogin, о котором речь пойдет чуть ниже.

Нетрудно заметить, что определения методов класса TDDGSalesDataModule разделены на группы по функциональному признаку. Эти группы следующие.

Группа методов	Выполняемые функции
Методы подключения	Методы, позволяющие пользователю войти и выйти из приложения
Методы, связанные с покупателями	Методы обработки данных о покупателях
Методы, связанные с запчастями	Методы обработки данных о запчастях
Методы, связанные с продажами	Методы создания и обработки записей о результатах проведения операции продажи

Чтобы понять назначение различных методов, достаточно прочитать комментарии, приведенные в листинге. В частности, в методе SaveSale() используется компонент TStoredProc, который создает новую запись, относящуюся к операции продажи, и добавляет в нее соответствующие элементы. Хранимые процедуры в этом методе подключаются к хранимым процедурам, представленным в листингах 33.1 и 33.2.

Методы Login/Logout

Методы, предназначенные для входа и выхода из приложения, называются Login() и Logout() соответственно. Метод Login(), в свою очередь, вызывает метод Connect(), который устанавливает соединение с базой данных через компонент dbSales. Это осуществляется посредством установки свойства dbSales.Connected равным значению True. В этом случае вызывается обработчик события dbSales.OnLogin, если, конечно, он существует. Этот обработчик события dbSalesLogin() вызывает метод GetLoginParams(), определенный в модуле LoginFrm.pas, который с помощью диалогового окна принимает от пользователя регистрационные данные. Текст этого метода представлен в листинге 33.4.

Листинг 33.4. Класс TLoginForm — пользовательская форма регистрации

```
unit LoginFrm;

interface

uses WinTypes, WinProcs, Classes, Graphics, Forms, Controls, StdCtrls,
  Buttons, ExtCtrls;

type
  TLoginForm = class(Tform)
    lblEnterPassword: Tlabel;
    lblEnterName: Tlabel;
    edtName: Tedit;
    edtPassword: Tedit;
    btnOK: Tbutton;
    btnCancel: Tbutton;
  public
  end;

function GetLoginParams(ALoginParams: Tstrings): Boolean;

implementation

{$R *.DFM}

function GetLoginParams(ALoginParams: Tstrings): Boolean;
var
  LoginForm: TLoginForm;
begin
  Result := False;
  LoginForm := TLoginForm.Create(Application);
  try
    if LoginForm.ShowModal = mrOk then
    begin
      ALoginParams.Values['USER NAME'] := LoginForm.edtName.Text;
      ALoginParams.Values['PASSWORD'] := LoginForm.edtPassWord.Text;
      Result := True;
```

```
    end;
  finally
    LoginForm.Free;
  end;
end;

end.
```

Метод Logout() просто закрывает компонент базы данных dbSales, который, в свою очередь, закрывает все соединения Tquery/Ttable.

Методы работы с таблицей CUSTOMER

Класс DDGSalesDataModule содержит несколько методов, предназначенных для работы с таблицей CUSTOMER: NewCustomer(), AcceptCustomer(), EditCustomer(), DeleteCustomer() и CancelCustomer(). Все перечисленные методы, чтобы выполнить нужное действие, просто вызывают соответствующие методы компонента Tquery. Остальные методы требуют дополнительных пояснений.

Функция GetCustomerName() считывает имя компании, к которой "принадлежит" покупатель. Если же таковое отсутствует, функция возвращает имя и фамилию покупателя, уникальный идентификатор (ID) которого задается параметром CustID.

С помощью метода SearchForCustomer() пользователь может выполнить поиск информации, которая хранится в таблице CUSTOMER для определенного покупателя. Поиск основан на значениях полей, задаваемых пользователем с помощью формы поиска покупателя. Эта форма создает строку запроса, которая передается серверу. О работе этой формы поговорим чуть позже, а пока предположим, что она формирует строку запроса, которая присваивается свойству qryCustomerSearch.SQL. Если в таблице CUSTOMER содержится информация об искомом покупателе, то текущей становится запись, соответствующая данному запросу.

Методы работы с таблицей PART

Методы работы с таблицей PART аналогичны методам работы с таблицей CUSTOMER. Методы NewPart(), EditPart(), AcceptPart(), DeletePart() и CancelPart() имеют простую организацию и для выполнения конкретной операции вызывают соответствующие методы компонента Tquery.

Метод SearchForPart() несколько проще метода SearchForCustomer(). С помощью функции InputQuery() он считывает номер запчасти, а затем, используя функцию Locate(), выполняет операцию поиска заданной запчасти.

Методы работы с таблицей SALES

Методы работы с таблицей SALES (в частности, метод SaveSale()) решают уже более интересные задачи, связанные с выполнением различных операций с базой данных клиент/сервер.

С помощью метода AddItemToSale() пользователь может указать товары, заказанные в новой операции продажи (как показано на рис. 33.2).

Метод CancelSale() завершает операцию по вставке данных о продаже.

Метод SaveSale() является самым сложным методом класса DDGSalesDataModule. В нем для добавления в базу данных сведений о продаже используются функции поддержки транзакций компонента dbSales. Сюда входят операции начала транзакции, создания новой записи о продаже, добавления *x* номеров продаваемых элементов и завершения транзакции или отмены изменений, внесенных всем процессом в целом.

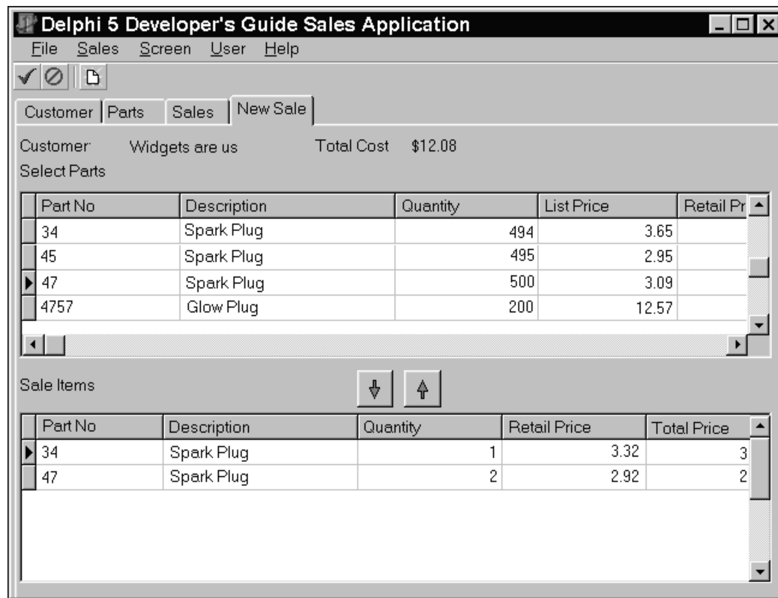


Рис. 33.2. Выбор товаров, заказанных в новой операции продажи

Запись с данными о продаже добавляется с помощью хранимой процедуры spInsertSale. Обратите внимание на то, как с помощью следующей инструкции клиенту возвращается номер продажи, который генерируется внутри настоящей хранимой процедуры:

```
SaleNo := ParamByName('rSALE_NUMBER').AsInteger;
```

Затем это значение используется для каждой записи, добавляемой в таблицу ITEMS с помощью экземпляра spInsertSaleItems компонента TStoredProc. Теперь вам должно быть понятно, как отдельные записи об отпускаемых товарах связываются с общими данными о продаже.

Методы работы с временной таблицей

Для выполнения операций с данными временной таблицы TEMPPART, используемой для хранения элементов для продажи, разработано несколько методов. В табл. 33.1 показано определение этой рабочей таблицы.

Таблица 33.1. Поля таблицы TEMPPART.DB

Имя поля	Тип	Размер	Значение
PART_NO	A	10	Номер запчасти для данного элемента
DESCRIPTION	A	18	Описание этого элемента

Окончание табл. 33.1

Имя поля	Тип	Размер	Значение
QUANTITY	S		Количество проданных запчастей
RETAIL_PRICE	N	50	Розничная цена проданного элемента
RETAIL_PRICE	N	50	Общая стоимость проданных запчастей

Метод AddItemToSale() предназначен для добавления сведений о запчасти в текущую операцию продажи.

Метод SaleItemsTotalPrice() возвращает общую стоимость элементов, существующих в таблице tblTempItems. В этом методе компонент qryTotalPrice используется для выполнения запроса к таблице формата Paradox с целью вычисления общей стоимости. Выполняемый при этом SQL-оператор имеет следующий вид:

```
select SUM(RETAIL_PRICE) from temppart.db
```

Этот оператор возвращает сумму числовых значений для заданного столбца (в данном случае — RETAIL_PRICE).

Метод tblTempItemsBeforePost() является обработчиком события tblTempParts.BeforePost. Этот обработчик гарантирует, что отправляемая запись отображает правильную стоимость на основе количества продаваемых элементов. Это возможно благодаря тому, что событие BeforePost происходит до реальной отправки записи в таблицу.

Оповещение пользователей класса TDataModule о событиях, связанных с компонентами доступа к данным

Одна из проблем, связанная с централизацией доступа к базе данных, состоит в том, что каждый компонент доступа к данным может иметь собственное событие, о котором вы хотите поставить в известность пользовательский интерфейс. Весьма желательно, чтобы пользовательский интерфейс реагировал на все события компонентов доступа к данным. Но так как эти компоненты располагаются в классе TDataModule, то формы не могут автоматически перехватывать эти события. Необходимо также иметь в виду, что класс TDataModule может попасть к вам в виде скомпилированного модуля.

Оповестить о возникновении определенных событий проще всего путем ввода в классе TDataModule собственного события, которому любые формы смогут назначить собственный обработчик событий. Это событие, в свою очередь, может произойти в результате события определенного компонента. Именно таким способом — благодаря свойству AfterTempItemsChange — и происходит оповещение о событиях AfterPost и AfterDelete для таблицы tblTempParts. Для этого свойства определены и метод чтения, и метод записи, которые имеют прямой доступ к действительным свойствам класса tblTempParts.

Создание пользовательского интерфейса

Определившись с централизованным доступом к данным, можно переходить к построению пользовательского интерфейса, положив в основу его функционирования методы, свойства и события объекта TDataModule. В следующих разделах обсуждаются различные формы, которыми можно наполнить подобное приложение.

В основу нашего приложения мы положим такую структуру (см. главу 4, “Строение приложения и концепции конструирования”, том I), при которой некоторая форма может стать дочерним окном другого окна.

Общая модель создаваемого приложения показана на рис. 33.3.

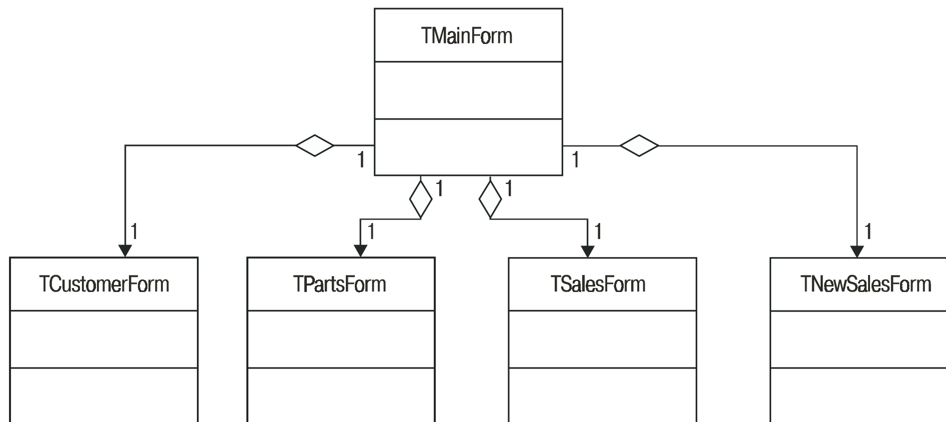


Рис. 33.3. Структура приложения Inventory Manager

Главная форма приложения (`TMainForm`) может содержать четыре дочерние формы.

- `Customer`. Используется для добавления, редактирования и просмотра информации о покупателях в системе.
- `Parts`. Используется для добавления, редактирования и просмотра записей о наличии запчастей на складе.
- `Sales`. Используется для просмотра сведений об операциях продажи.
- `New Sales`. Используется для добавления новой операции продажи.

Существует несколько других вспомогательных форм, которые не вызываются в качестве дочерних форм главной формы. Эти формы рассматриваются несколько ниже, а пока обратимся к главной форме и каждой из ее дочерних форм.

Класс `TMainForm` — главная форма приложения

Главная форма приложения содержит компонент `TTabControl`, который служит в качестве родительского для дочерних форм. Пользователь переходит в новую дочернюю форму, либо выбирая команду из главного меню, либо щелкая на корешке вкладки требуемого экземпляра `tcMain` компонента `TTabControl`. Логика работы формы реализована таким образом, чтобы обеспечить синхронизацию элементов меню и вкладок формы. Большая часть логики направлена на гарантированное создание только одной формы и ее отображение, сопровождаемое освобождением всех других форм.

В листинге 33.5 представлен исходный текст главной формы `TMainForm`.

Листинг 33.5. `TMainForm` — главная форма приложения Inventory Manager

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
```

```
  Menus, StdCtrls, ComCtrls, ExtCtrls, ChildFrm;

type

  { Существует четыре типа дочерних форм, которые могут быть отображены
    в этом приложении. Объект TActiveScreenType предназначен для того,
    чтобы знать, какие из четырех типов форм активны в данный момент. }

  TActiveScreenType = (acCustomer, acParts, acSales, acNewSales);

  TMainForm = class(TForm)
    mmSales: TMainMenu;
    mmiScreen: TMenuItem;
    mmiCustomer: TMenuItem;
    mmiParts: TMenuItem;
    mmiNewSale: TMenuItem;
    mmiSales: TMenuItem;
    mmiFile: TMenuItem;
    mmiExit: TMenuItem;
    mmiHelp: TMenuItem;
    tcMain: TTabControl;
    mmiUser: TMenuItem;
    mmiLogon: TMenuItem;
    mmiLogoff: TMenuItem;
    imgCar: TImage;
    procedure ScreenClick(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure tcMainChange(Sender: TObject);
    procedure tcMainChanging(Sender: TObject; var AllowChange: Boolean);
    procedure mmiLogonClick(Sender: TObject);
    procedure mmiLogoffClick(Sender: TObject);
  private
    // Экземпляр ActiveScreenType хранит тип активной формы.
    ActiveScreenType: TActiveScreenType;
    // Объект ActiveScreen представляет собой ссылку на активную дочернюю форму
    ActiveScreen: TChildForm;
    procedure SetActiveScreen;
  public
    { Открытые объявления }
  end;

var
  MainForm: TMainForm;

implementation

uses CustomerFrm, PartsFrm, NewSalesFrm, SalesFrm, SalesDM;

{$R *.DFM}
```

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  // Устанавливаем выравнивание вкладок главной формы.
  tcMain.Align := alClient;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.ScreenClick(Sender: TObject);
{ Этот метод вызывается, если пользователь решил перейти в другую
  форму с помощью главного меню.
  Данный метод определяет, можно ли перейти к другой дочерней форме.
  Это реализуется путем проверки, возвращает ли метод CanChange() каждой
  дочерней формы значение True. Если это подтверждается, то изменяется
  глобальное значение ActiveScreenType и вызывается метод SetActiveScreen(),
  чтобы выполнить реальное изменение логики. }
  if Sender is TMenuItem then
  begin
    if ActiveScreen <> nil then
    begin
      if ActiveScreen.CanChange then
      begin

        TMenuItem(Sender).Checked := True;
        if Sender = mmiCustomer then
          ActiveScreenType := acCustomer
        else if Sender = mmiParts then
          ActiveScreenType := acParts
        else if Sender = mmiSales then
          ActiveScreenType := acSales
        else if Sender = mmiNewSale then
          ActiveScreenType := acNewSales;

        { Обеспечиваем синхронизацию элемента управления вкладками
          TTabControl с элементом меню, на котором щелкнул пользователь.}
        tcMain.TabIndex := ord(ActiveScreenType);
        SetActiveScreen;
      end
    end;
  end;
end;

procedure TMainForm.tcMainChange(Sender: TObject);
begin
{ Этот метод изменяет экран, когда пользователь переключает вкладки.
  Он синхронизирует установки главного меню и элемента управления вкладками.
  Данный метод вызывает метод SetActiveScreen(), чтобы на самом деле
```

```
  изменить активный экран.}
  if ActiveScreen <> nil then
  begin
    case tcMain.TabIndex of
      0: mmiCustomer.Checked := True;
      1: mmiParts.Checked    := True;
      2: mmiSales.Checked    := True;
      3: mmiNewSale.Checked  := True;
    end;
    ActiveScreenType := TActiveScreenType(tcMain.TabIndex);
    SetActiveScreen;
  end;
end;

procedure TMainForm.SetActiveScreen;
{ Этот метод заменяет активный экран одной из четырех дочерних форм.
  Каждая из этих форм выводится как дочерняя форма экземпляра tcMain
  компонента TTabControl. }
var
  TempScreen: TChildForm;
begin
  { Определяем, создан ли у нас экземпляр дочерней формы. Если создан,
    отсоединяем меню этой дочерней формы и освобождаем ее. }

  TempScreen := ActiveScreen;

  // Отсоединяем меню.
  if Assigned(ActiveScreen) then
  begin
    if ActiveScreen.GetFormMenu <> nil then
      mmSales.UnMerge(ActiveScreen.GetFormMenu);
  end;

  { Определяем, какой активный экран (дочернюю форму) нужно создать,
    и устанавливаем (при необходимости) ее панели инструментов так,
    чтобы главная форма служила родительской. }
  case ActiveScreenType of
    acCustomer:
      begin
        ActiveScreen := TCustomerForm.Create(Application, tcMain);
        TCustomerForm(ActiveScreen).SetToolBarParent(self);
      end;
    acParts:
      begin
        ActiveScreen := TPartsForm.Create(Application, tcMain);
        TPartsForm(ActiveScreen).SetToolBarParent(self);
      end;
    acSales:
        ActiveScreen := TSalesForm.Create(Application, tcMain);
    acNewSales:
```

```
      begin
        ActiveScreen := TNewSalesForm.Create(Application, tcMain);
        TPartsForm(ActiveScreen).SetToolBarParent(self);
      end;
  end;

  // Объединяем меню дочерней формы с меню главной формы
  if ActiveScreen <> nil then
  begin
    if ActiveScreen.GetFormMenu <> nil then
      mmSales.Merge(ActiveScreen.GetFormMenu);
    ActiveScreen.Show;
  end;

  if Assigned(TempScreen) then
    TempScreen.Free;
end;

procedure TMainForm.tcMainChanging(Sender: TObject;
  var AllowChange: Boolean);
begin
  { Замена экрана выполняется только в том случае, если дочерняя форма
    находится в режиме, который позволяет замену. }
  AllowChange := ActiveScreen.CanChange;
end;

procedure TMainForm.mmiLogonClick(Sender: TObject);
begin
  // Регистрируем пользователя в системе
  if DDGSalesDataModule.Login then
  begin
    tcMain.Align := alClient;
    tcMain.Visible := True;
    ActiveScreenType := acCustomer;
    SetActiveScreen;
    mmiScreen.Enabled := True;
    mmiLogon.Enabled  := False;
    mmiLogoff.Enabled := True;
  end;
end;

procedure TMainForm.mmiLogoffClick(Sender: TObject);
begin
  // Выводим пользователя из системы
  if Assigned(ActiveScreen) then
  begin
    if ActiveScreen.GetFormMenu <> nil then
      mmSales.UnMerge(ActiveScreen.GetFormMenu);
    ActiveScreen.Free;
    ActiveScreen := nil;
```

```
  end;

  tcMain.Visible := False;
  DDGSalesDataModule.Logout;

  mmiScreen.Enabled := False;
  mmiLogon.Enabled  := True;
  mmiLogoff.Enabled := False;

end;

end.
```

За подробностями о каждом методе обращайтесь к комментариям, приведенным в листинге 33.5. Бо́льшая часть кода этого приложения содержится в уже рассмотренном модуле DDGSalesDataModule. Оставшаяся часть логики относится к пользовательскому интерфейсу и сосредоточена в дочерних формах. Ими мы сейчас и займемся.

Форма TCustomerForm — ввод данных о покупателе

С помощью формы TCustomerForm пользователь может добавлять, редактировать и удалять из базы данных информацию о покупателях. Эта форма показана на рис. 33.4. Поскольку бо́льшая часть логики пользовательского интерфейса сосредоточена в классах, являющихся предками класса TCustomerForm, исходный текст этой формы, показанный в листинге 33.6, невелик и легко доступен для понимания.

Рис. 33.4. TCustomerForm — форма ввода данных о покупателях

Листинг 33.6. `TCustomerForm` **— форма ввода данных о покупателях**

```
unit CustomerFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  DBNAVSTATFRM, StdCtrls, DBCtrls, Mask, Menus, ImgList, ComCtrls, ToolWin,
  Db, DBModeFrm;

type
  TCustomerForm = class(TDBNavStatForm)
    lblFirstName: TLabel;
    dbeFirstName: TDBEdit;
    lblLastName: TLabel;
    dbeLastName: TDBEdit;
    lblCreditLine: TLabel;
    dbeCreditLine: TDBEdit;
    lblWorkAddress: TLabel;
    dbeWorkAddress: TDBEdit;
    lblHomeAddress: TLabel;
    dbeHomeAddress: TDBEdit;
    lblCity: TLabel;
    dbeCity: TDBEdit;
    lblState: TLabel;
    dbeState: TDBEdit;
    lblZipCode: TLabel;
    dbeZip: TDBEdit;
    lblWorkPhone: TLabel;
    dbeWorkPhone: TDBEdit;
    lblHomePhone: TLabel;
    dbeHomePhone: TDBEdit;
    lblComments: TLabel;
    dbmmComments: TDBMemo;
    lblCompany: TLabel;
    dbeCompany: TDBEdit;
    dsCustomer: TDataSource;
    EXit1: TMenuItem;
    procedure sbFirstClick(Sender: TObject);
    procedure sbPrevClick(Sender: TObject);
    procedure sbNextClick(Sender: TObject);
    procedure sbLastClick(Sender: TObject);
    procedure sbInsertClick(Sender: TObject);
    procedure sbEditClick(Sender: TObject);
    procedure sbDeleteClick(Sender: TObject);
    procedure sbCancelClick(Sender: TObject);
    procedure sbAcceptClick(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure sbFindClick(Sender: TObject);
```

```
    procedure sbBrowseClick(Sender: TObject);
  private
    procedure SetNavButtons;
  public
    function GetFormMenu: TMainMenu; override;
    function CanChange: Boolean; override;
  end;

var
  CustomerForm: TCustomerForm;

implementation

uses SalesDM;

{$R *.DFM}

procedure TCustomerForm.SetNavButtons;
begin
{ Обеспечиваем настройку кнопок навигации в соответствии с режимом формы. }
  sbFirst.Enabled := not DDGSalesDataModule.IsFirstCustomer;
  sbLast.Enabled  := not DDGSalesDataModule.IsLastCustomer;
  sbPrev.Enabled  := not DDGSalesDataModule.IsFirstCustomer;
  sbNext.Enabled  := not DDGSalesDataModule.IsLastCustomer;

  { Синхронизируем элементы меню с кнопками панели инструментов.}
  mmiFirst.Enabled    := sbFirst.Enabled;
  mmiLast.Enabled     := sbLast.Enabled;
  mmiPrevious.Enabled := sbPrev.Enabled;
  mmiNext.Enabled     := sbNext.Enabled;
end;

procedure TCustomerForm.sbFirstClick(Sender: TObject);
begin
  // Переходим к первой записи в результирующем наборе
  inherited;
  DDGSalesDataModule.FirstCustomer;
  SetNavButtons;
end;

procedure TCustomerForm.sbPrevClick(Sender: TObject);
begin
  // Переходим к предыдущей записи в результирующем наборе
  inherited;
  DDGSalesDataModule.PrevCustomer;
  SetNavButtons;
end;

procedure TCustomerForm.sbNextClick(Sender: TObject);
begin
```

```
  // Переходим к следующей записи в результирующем наборе
  inherited;
  DDGSalesDataModule.NextCustomer;
  SetNavButtons;
end;

procedure TCustomerForm.sbLastClick(Sender: TObject);
begin
  // Переходим к последней записи в результирующем наборе
  inherited;
  DDGSalesDataModule.LastCustomer;
  SetNavButtons;
end;

procedure TCustomerForm.sbInsertClick(Sender: TObject);
begin
  // Вставляем информацию о новом покупателе
  inherited;
  DDGSalesDataModule.NewCustomer;
end;

procedure TCustomerForm.sbEditClick(Sender: TObject);
begin
  // Редактируем запись о текущем покупателе
  inherited;
  DDGSalesDataModule.EditCustomer;
end;

procedure TCustomerForm.sbDeleteClick(Sender: TObject);
begin
  // Удаляем запись о текущем покупателе
  inherited;
  DDGSalesDataModule.DeleteCustomer;
end;

procedure TCustomerForm.sbCancelClick(Sender: TObject);
begin
  // Отменяем операцию редактирования или добавления.
  inherited;
  DDGSalesDataModule.CancelCustomer;
end;

procedure TCustomerForm.sbAcceptClick(Sender: TObject);
begin
  { Принимаем изменения, выполненные при редактировании или добавлении. }
  inherited;
  DDGSalesDataModule.AcceptCustomer;
end;

procedure TCustomerForm.FormShow(Sender: TObject);
```

```
begin
  // Инициализируем меню и кнопки
  inherited;
  SetNavButtons;
end;

function TCustomerForm.CanChange: Boolean;
begin
  { Разрешаем пользователю изменять формы только при просмотре записей. }
  Result := FormMode = fmBrowse;
end;

function TCustomerForm.GetFormMenu: TMainMenu;
begin
  { Возвращаемся в главное меню. Этого требует главная форма для слияния меню. }
  Result := mmFormMenu;
end;

procedure TCustomerForm.sbFindClick(Sender: TObject);
begin
  { Поиск заданного покупателя путем вызова соответствующей формы поиска. }
  inherited;
  DDGSalesDataModule.SearchForCustomer;
end;

procedure TCustomerForm.sbBrowseClick(Sender: TObject);
begin
  { Переводим форму в режим просмотра. При этом операция редактирования
   или добавления будет невозможна. }
  inherited;
  if not (FormMode = fmBrowse) then
    DDGSalesDataModule.CancelCustomer;
end;

end.
```

Комментарии, приведенные в этом листинге, помогут понять назначение отдельных методов. Небольшой объем кода, реализующего работу данной формы, объясняется тем, что львиная часть логики базы данных содержится в классе TDDGSalesDataModule, не говоря уже о том, какой объем функций реализован компонентами библиотеки VCL. Остальные дочерние формы мало отличаются от данной.

Форма TPartsForm — ввод данных о содержимом склада

Форма ввода данных о запчастях TPartsForm показана на рис. 33.5 , а ее исходный код представлен в листинге 33.7.

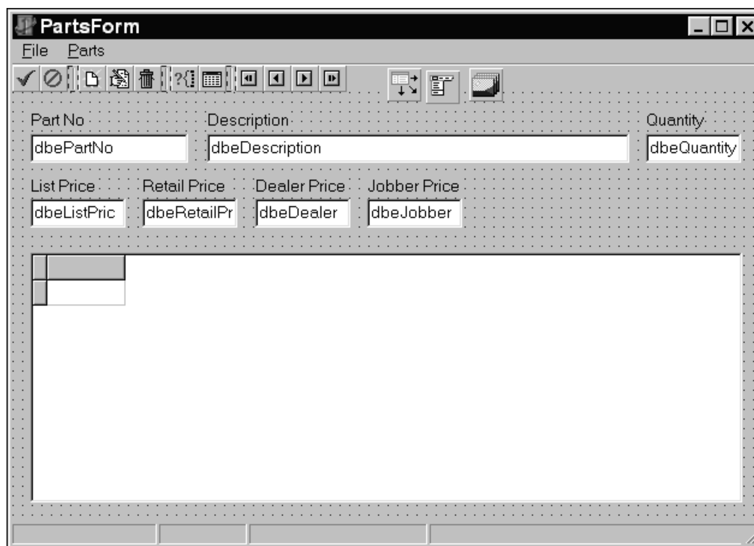


Рис. 33.5. `TPartsForm` *— форма ввода данных о запчастях*

Листинг 33.7. `TPartsForm` **— форма ввода данных о запчастях**

```
unit PartsFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  DBNAVSTATFRM, Menus, ImgList, ComCtrls, ToolWin, Grids, DBGrids, Db,
  StdCtrls, Mask, DBCtrls, DBModeFrm;

type
  TPartsForm = class(TDBNavStatForm)
    lblPartNo: TLabel;
    dbePartNo: TDBEdit;
    dsParts: TDataSource;
    lblDescription: TLabel;
    dbeDescription: TDBEdit;
    lblQuantity: TLabel;
    dbeQuantity: TDBEdit;
    lblListPrice: TLabel;
    dbeListPrice: TDBEdit;
    lblRetailPrice: TLabel;
    dbeRetailPrice: TDBEdit;
    lblDealerPrice: TLabel;
    dbeDealerPrice: TDBEdit;
    lblJobberPrice: TLabel;
    dbeJobberPrice: TDBEdit;
    dbgParts: TDBGrid;
```

```
    procedure sbAcceptClick(Sender: TObject);
    procedure sbCancelClick(Sender: TObject);
    procedure sbInsertClick(Sender: TObject);
    procedure sbEditClick(Sender: TObject);
    procedure sbDeleteClick(Sender: TObject);
    procedure sbFirstClick(Sender: TObject);
    procedure sbPrevClick(Sender: TObject);
    procedure sbNextClick(Sender: TObject);
    procedure sbLastClick(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure sbFindClick(Sender: TObject);
    procedure sbBrowseClick(Sender: TObject);
  private
    procedure SetNavButtons;
  public
    function GetFormMenu: TMainMenu; override;
    function CanChange: Boolean; override;
  end;

var
  PartsForm: TPartsForm;

implementation

uses SalesDM;

{$R *.DFM}

procedure TPartsForm.SetNavButtons;
begin
  { Обеспечиваем установку кнопок навигации в соответствии с режимом формы. }
  sbFirst.Enabled := not DDGSalesDataModule.IsFirstPart;
  sbLast.Enabled  := not DDGSalesDataModule.IsLastPart;
  sbPrev.Enabled  := not DDGSalesDataModule.IsFirstPart;
  sbNext.Enabled  := not DDGSalesDataModule.IsLastPart;

  { Синхронизируем элементы меню навигации с кнопками панели инструментов. }
  mmiFirst.Enabled := sbFirst.Enabled;
  mmiLast.Enabled  := sbLast.Enabled;
  mmiPrevious.Enabled  := sbPrev.Enabled;
  mmiNext.Enabled  := sbNext.Enabled;

end;

procedure TPartsForm.sbAcceptClick(Sender: TObject);
begin
  { Принятие изменений, связанных с добавлением или редактированием
    для данной запчасти. }
  inherited;
  DDGSalesDataModule.AcceptPart;
end;
```

```
procedure TPartsForm.sbCancelClick(Sender: TObject);
begin
  // Отменяем операцию редактирования или добавления
  inherited;
  DDGSalesDataModule.CancelPart;
end;

procedure TPartsForm.sbInsertClick(Sender: TObject);
begin
  // Вставляем запись о новой запчасти
  inherited;
  DDGSalesDataModule.NewPart;
end;

procedure TPartsForm.sbEditClick(Sender: TObject);
begin
  // Редактируем запись о текущей запчасти
  inherited;
  DDGSalesDataModule.EditPart;
end;

procedure TPartsForm.sbDeleteClick(Sender: TObject);
begin
  // Удаляем запись о текущей запчасти
  inherited;
  DDGSalesDataModule.DeletePart;
end;

procedure TPartsForm.sbFirstClick(Sender: TObject);
begin
  // Переходим к первой записи в результирующем наборе
  inherited;
  DDGSalesDataModule.FirstPart;
  SetNavButtons;
end;

procedure TPartsForm.sbPrevClick(Sender: TObject);
begin
  // Переходим к предыдущей записи в результирующем наборе
  inherited;
  DDGSalesDataModule.PrevPart;
  SetNavButtons;
end;

procedure TPartsForm.sbNextClick(Sender: TObject);
begin
  // Переходим к следующей записи в результирующем наборе
  inherited;
  DDGSalesDataModule.NextPart;
  SetNavButtons;
end;
```

```
procedure TPartsForm.sbLastClick(Sender: TObject);
begin
  // Переходим к последней записи в результирующем наборе
  inherited;
  DDGSalesDataModule.LastPart;
  SetNavButtons;
end;

procedure TPartsForm.FormShow(Sender: TObject);
begin
  { Инициализируем требуемым образом кнопки панели инструментов
    и элементы меню. }
  inherited;
  SetNavButtons;
end;

function TPartsForm.CanChange: Boolean;
begin
  { Разрешаем пользователю замену форм только в случае, если не
     выполняется операция добавления или редактирования записи. }
  Result := FormMode = fmBrowse;
end;

function TPartsForm.GetFormMenu: TMainMenu;
begin
  { Возвращаемся в главное меню. Этого требует главная форма для слияния
    меню дочерних форм. }
  Result := mmFormMenu;
end;

procedure TPartsForm.sbFindClick(Sender: TObject);
begin
  // Поиск запчасти по ее номеру
  inherited;
  DDGSalesDataModule.SearchForPart;
end;

procedure TPartsForm.sbBrowseClick(Sender: TObject);
begin
  { Переходим в режим просмотра - но только после отмены любых изменений,
    внесенных в текущую запись.
  inherited;
  if not (FormMode = fmBrowse) then
    DDGSalesDataModule.CancelPart;
end;

end.
```

Как видите, этот листинг практически идентичен листингу 33.6, что отнюдь не является недостатком — постоянство является желательным атрибутом и способствует пониманию.

Форма TSalesForm — просмотр данных о продажах

Форма TSalesForm используется для просмотра данных о совершенных операциях продажи (рис. 33.6). Исходный код этой формы содержит только один метод GetFormMenu(), который должен быть переопределен для возврата значения nil, чтобы главная форма не пыталась выполнить операцию слияния меню. Здесь не приводится листинг этой формы, поскольку ее код не содержит никаких моментов, заслуживающих особого внимания. Текст формы TSalesForm можно найти на прилагаемом компакт-диске, в файле SaleFrm.pas, расположенном в каталоге данной главы.

Рис. 33.6. TSalesForm — форма просмотра данных о продажах

Форма TNewSalesForm — ввод данных о продажах

Форма TNewSalesForm, пожалуй, самая сложная из всех четырех дочерних форм. Ее исходный текст представлен в листинге 33.8. И вместе с тем, это очень простая форма. Основные моменты логики программирования поясняются комментариями. В частности, обратите внимание на то, что мы вынуждены были создать метод, возвращающий компонент TToolBar. Этот метод уже существует в классе TDBNavStatForm, потомками которого служат другие дочерние формы, но данная форма является потомком только класса TChildForm. Поэтому нам пришлось создать подобный метод специально для этой формы. Внешний вид формы TNewSalesForm показан на рис. 33.7.

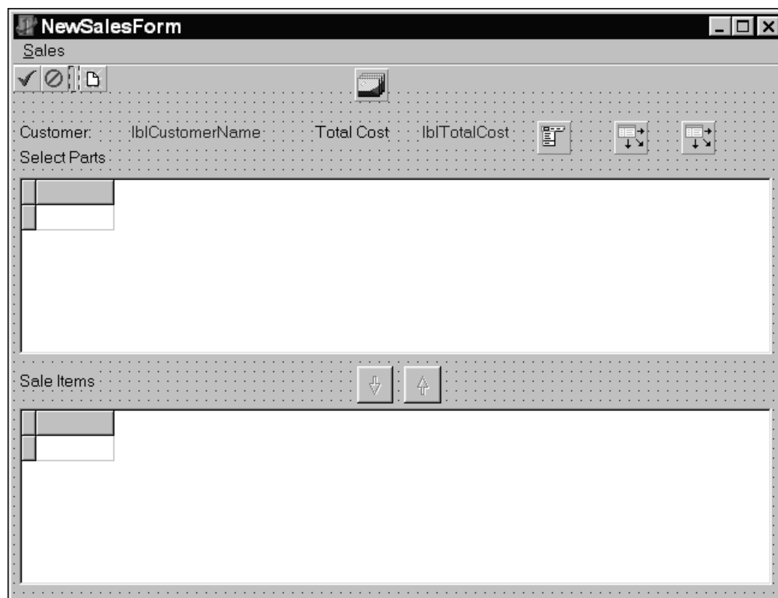


Рис. 33.7. `TNewSalesForm` *— форма ввода новых данных о продажах*

Листинг 33.8. `TNewSalesForm` **— форма ввода новых данных о продажах**

```
unit NewSalesFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
 CHILDFRM, Grids, DBGrids, Buttons, StdCtrls, Db, Menus, ToolWin, ComCtrls,
 ImgList;

type
  TNewSalesForm = class(TChildForm)
    dsParts: TDataSource;
    dsTempItems: TDataSource;
    lblCustomer1: TLabel;
    lblCustomerName: TLabel;
    lblTotCost: TLabel;
    lblTotalCost: TLabel;
    lblSelectParts: TLabel;
    sbAddPart: TSpeedButton;
    sbRemovePart: TSpeedButton;
    lblSaleItems: TLabel;
    dbgParts: TDBGrid;
    dbgSaleItems: TDBGrid;
    mmFormMenu: TMainMenu;
    mmiSales: TMenuItem;
```

```
    mmiNew: TMenuItem;
    mmiCancel: TMenuItem;
    mmiSave: TMenuItem;
    tbSales: TToolBar;
    sbAccept: TToolButton;
    sbCancel: TToolButton;
    tb1: TToolButton;
    sbInsert: TToolButton;
    ilNavigationBar: TImageList;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure sbAddPartClick(Sender: TObject);
    procedure mmiNewClick(Sender: TObject);
    procedure mmiCancelClick(Sender: TObject);
    procedure mmiSaveClick(Sender: TObject);
  private
    AddingSale: Boolean;

    procedure SetSaleMenus;
    procedure TempItemsAfterChange(DataSet: TDataSet);
  public
    function CanChange: Boolean; override;
    function GetFormMenu: TMainMenu; override;
    procedure SetToolBarParent(AParent: TWinControl);
  end;

var
  NewSalesForm: TNewSalesForm;

implementation

uses SalesDM;

{$R *.DFM}

function TNewSalesForm.CanChange: Boolean;
begin
  Result := not AddingSale;
end;

procedure TNewSalesForm.FormCreate(Sender: TObject);
begin
  inherited;
  { Для этой формы требуется таблица tblTempItems
    из модуля DDGSalesDataModule. }
  DDGSalesDataModule.OpenTempItems;
  AddingSale := False;  // Исходно операция добавления не выполняется

  { Назначаем обработчик события TempItemsAfterChange
    модуля DDGSalesDataModule. }
```

```
  DDGSalesDataModule.AfterTempItemsChange := TempItemsAfterChange;
  SetSaleMenus;
end;

procedure TNewSalesForm.FormDestroy(Sender: TObject);
begin
  // Закрываем таблицу DDGSalesDataModule.tblTempItems
  inherited;
  DDGSalesDataModule.CloseTempItems;
end;

procedure TNewSalesForm.FormShow(Sender: TObject);
begin
  // Считываем имя текущего покупателя
  inherited;
  lblCustomerName.Caption := DDGSalesDataModule.GetCustomerName;
  { Значение общей стоимости должно быть равно нулю,
    поскольку форма была только что вызвана. }
  lblTotalCost.Caption    := '$   0.00';
end;

procedure TNewSalesForm.TempItemsAfterChange(DataSet: TDataSet);
begin
  { Эти действия необходимо выполнять при обработке события AfterPost
    таблицы tblTempItems, принадлежащей модулю данных DDGSalesDataModule,
    поскольку мы должны пересчитывать это значение каждый раз,
    когда пользователь вносит изменения. }
  lblTotalCost.Caption := FormatFloat('$#,##0.00',
      DDGSalesDataModule.SaleItemsTotalPrice);
end;

procedure TNewSalesForm.sbAddPartClick(Sender: TObject);
begin
  // Добавляем выбранный элемент в сведения о продаже
  inherited;
  DDGSalesDataModule.AddItemToSale;
end;

procedure TNewSalesForm.mmiNewClick(Sender: TObject);
begin
  { Переводим форму в режим добавления данных о продаже. }
  inherited;
  AddingSale := True;
  SetSaleMenus;
end;

procedure TNewSalesForm.mmiCancelClick(Sender: TObject);
begin
  // Отменяем текущую продажу
  inherited;
  AddingSale := False;
```

```
  DDGSalesDataModule.CancelSale;
  SetSaleMenus;
end;

procedure TNewSalesForm.mmiSaveClick(Sender: TObject);
begin
  // Сохраняем сведения о данной продаже
  inherited;
  DDGSalesDataModule.SaveSale;
  AddingSale := False;
  SetSaleMenus;
  { Вызываем обработчик события TempItemsAfterChange, чтобы обеспечить
    соответствующее обновление всех элементов управления формы. }
  TempItemsAfterChange(nil);
end;

procedure TNewSalesForm.SetSaleMenus;
begin
  { Настраиваем элементы меню и кнопки панели инструментов
    на отображение режима формы. }
  mmiNew.Enabled       := not AddingSale;
  mmiCancel.Enabled    := AddingSale;
  mmiSave.Enabled      := AddingSale;
  sbAddPart.Enabled    := AddingSale;
  sbRemovePart.Enabled := AddingSale;

  sbAccept.Enabled     := mmiSave.Enabled;
  sbCancel.Enabled     := mmiCancel.Enabled;
  sbInsert.Enabled     := mmiNew.Enabled;

end;

function TNewSalesForm.GetFormMenu: TMainMenu;
begin
  { Возвращаем главное меню, которое главная форма
    должна использовать при объединении меню. }
  Result := mmFormMenu;
end;

procedure TNewSalesForm.SetToolBarParent(AParent: TWinControl);
begin
  { Поскольку эта форма использует панель инструментов, возвращаем
    ее родительский вариант. Мы вынуждены создать этот метод для данной формы,
    так как она является потомком класса TChildForm, а не класса
    TDBNavStatForm, который уже содержит этот метод. }
  tbSales.Parent := AParent;
end;

end.
```

Диалоговое окно поиска данных о покупателях

Форма TCustomerSearchForm применяется модулем DDGSalesDataModule для получения критериев запроса, необходимых для выполнения поиска в таблице CUSTOMER. На эту форму возлагается ответственность за получение значений полей от пользователя и построение инструкции запроса в SQL-команде. Форма TCustomerSearchForm показана на рис. 33.8.

Форма TCustomerSearchForm, подобно формам, упомянутым выше, не является дочерней. Она не содержит никаких элементов управления для работы с данными. Пользователь должен ввести значения в поля, которые будут участвовать в определении условия поиска, а затем щелкнуть на подписях (элементах управления TLabel), относящихся к полям, используемым в поиске. При этом цвет подписей изменится на красный (значение clRed). В логике формы TCustomerSearchForm при построении SQL-команды запроса используются как значения, вводимые пользователем, так и значения цветов элементов управления TLabel.

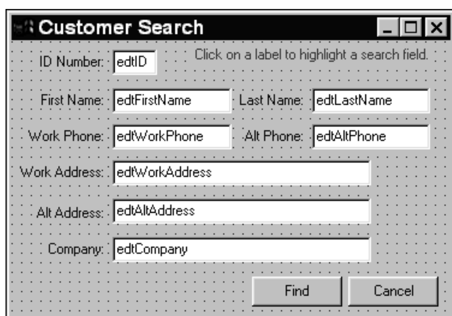


Рис. 33.8. Форма поиска данных о покупателях

Исходный текст формы TCustomerSearchForm представлен в листинге 33.9.

Листинг 33.9. TCustomerSearchForm — форма поиска данных о покупателях

```
unit CustomerSrchFrm;

interface

uses WinTypes, WinProcs, Classes, Graphics, Forms, Controls, Buttons,
  StdCtrls, SysUtils;

type
  TCustomerSearchForm = class(TForm)
    lblIDNumber: TLabel;
    edtIDNumber: TEdit;
    lblFirstName: TLabel;
    lblLastName: TLabel;
    lblAltPhone: TLabel;
    lblWorkPhone: TLabel;
    lblWorkAddress: TLabel;
    lblAltAddress: TLabel;
    lblCompany: TLabel;
    edtFirstName: TEdit;
    edtLastName: TEdit;
    edtWorkPhone: TEdit;
    edtAltPhone: TEdit;
    edtWorkAddress: TEdit;
    edtAltAddress: TEdit;
```

```
    edtCompany: TEdit;
    btnCancel: TButton;
    btnFind: TButton;
    lblInstruction: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure FindCustBtnClick(Sender: TObject);
    procedure CancelBtnClick(Sender: TObject);
    procedure lblIDNumberClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    FindPressed: Boolean;
    procedure ClearEditFields;
    function BuildSQLStatement: string;
  public
    QueryString: String;
  end;

function SearchCustomer: String;

implementation

{$R *.DFM}

uses Dialogs;

function SearchCustomer: String;
var
  CustomerSearchForm: TCustomerSearchForm;
begin
  Result := EmptyStr;
  CustomerSearchForm := TCustomerSearchForm.Create(Application);
  try
    if CustomerSearchForm.ShowModal = mrOk then
      Result := CustomerSearchForm.QueryString;
  finally
    CustomerSearchForm.Free;
  end;

end;


function TCustomerSearchForm.BuildSQLStatement: string;
{ Эта функция строит SQL-инструкцию запроса, основываясь на полях поиска
  в записи покупателя, заданных пользователем. Поля поиска выделяются
  с помощью подписей, окрашенных в красный цвет (значение clRed).
  Для выделения этих подписей пользователю достаточно щелкнуть на них.
  Пользователь должен ввести значения в поля редактирования, к которым
  относятся выделенные подписи. }
var
  Sep: String[3]; // Используется в качестве разделителя
```

```
begin
  Sep := '';
  Result := '';

  if lblIDNumber.Font.Color = clRed then
  begin
    Result := Format('(CUSTOMER_ID = %s)', [edtIDNumber.Text]);
    Sep := 'AND';
  end;

  if lblLastName.Font.Color = clRed then
  begin
    Result := Format('%s %s (UPPER(LNAME) = "%s")',
                [Result, Sep, UpperCase(edtLastName.Text)]);
    Sep := 'AND';
  end;

  if lblFirstName.Font.Color = clRed then
  begin
    Result := Format('%s %s (UPPER(FNAME) = "%s")',
                [Result, Sep, UpperCase(edtFirstName.Text)]);
    Sep := 'AND';
  end;

  if lblWorkPhone.Font.Color = clRed then
  begin
    Result := Format('%s %s (UPPER(WORK_PHONE) = "%s")',
                [Result, Sep, UpperCase(edtWorkPhone.Text)]);
    Sep := 'AND';
  end;

  if lblAltPhone.Font.Color = clRed then
  begin
    Result := Format('%s %s (UPPER(ALT_PHONE) = "%s")',
                [Result, Sep, UpperCase(edtAltPhone.Text)]);
    Sep := 'AND';
  end;

  if lblWorkAddress.Font.Color = clRed then
  begin
    Result := Format('%s %s (UPPER(WORK_ADDRESS) = "%s")',
                [Result, Sep, UpperCase(edtWorkAddress.Text)]);
    Sep := 'AND';
  end;

  if lblAltAddress.Font.Color = clRed then
  begin
    Result := Format('%s %s (UPPER(ALT_ADDRESS) = "%s")',
                [Result, Sep, UpperCase(edtAltAddress.Text)]);
    Sep := 'AND';
```

```
  end;

  if lblCompany.Font.Color = clRed then
  begin
    Result := Format('%s %s (UPPER(COMPANY) = "%s")',
                    [Result, Sep, UpperCase(edtCompany.Text)]);
  end;

  if Length(Result) > 0 then
    Result := Format('SELECT CUSTOMER_ID FROM CUSTOMER WHERE (%s)',
      [Result]);

end;

procedure TCustomerSearchForm.ClearEditFields;
{ Этот метод очищает все поля редактирования и устанавливает
  цвет их подписей равным значению clNavy. }
var
  i: word;
begin
  for i := 0 to ComponentCount - 1 do
  begin
    if Components[i] is TEdit then
      TEdit(Components[i]).Text := '';

    if Components[i] is TLabel then
      TLabel(Components[i]).Font.Color := clNavy;
  end;

end;

procedure TCustomerSearchForm.FormCreate(Sender: TObject);
begin
  FindPressed := False;
  // Очищаем поля редактирования
  ClearEditFields;
end;

procedure TCustomerSearchForm.FindCustBtnClick(Sender: TObject);
begin
  FindPressed := True;
  { Делаем строку запроса QueryString доступной источнику вызова этого
диалогового окна. }
  QueryString := BuildSQLStatement;
end;

procedure TCustomerSearchForm.CancelBtnClick(Sender: TObject);
begin
  ClearEditFields;
end;
```

```
procedure TCustomerSearchForm.lblIDNumberClick(Sender: TObject);
{ Все подписи связаны с этим обработчиком события OnClick, который изменяет
  цвет соответствующих подписей. Значение цвета clRed используется для
  определения подписи, над полем которой нужно будет выполнить операцию поиска. }
begin
  with (Sender as TLabel) do
    if Font.Color = clNavy then
      Font.Color := clRed
    else
      Font.Color := clNavy;
end;

procedure TCustomerSearchForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  { Перед закрытием формы, прежде чем выполнять операцию поиска,
    убедимся, что пользователь указал поля, участвующие в поиске. }
  if (QueryString = '') and FindPressed then
  begin
    MessageDlg('You must highlight a search field by'+
             ' clicking on a label.', mtInformation, [mbOk], 0);
    { Вы должны выделить поле поиска, щелкнув на подписи. }
    Action := caNone;
  end
  else begin
    Action := caHide;
    ClearEditFields;
  end;
end;

end.
```

Основным методом в данном модуле является функция BuildSQLStatement(), которая возвращает строковое представление SQL-команды запроса. Этот метод просматривает все подписи, и если значение цвета какой-нибудь из них окажется равным clRed, то содержимое строки, соответствующей данной подписи, используется для построения инструкции запроса с помощью набора инструкций Format().

Метод ClearEditFields() довольно прост. Его задача — установить значения цветов всех подписей равными константе clNavy и очистить содержимое строк редактирования. Этот метод используется при создании рассматриваемой формы в обработчике события FormCreate().

Обработчик события FormClose() гарантирует, что пользователь задал поля для проведения поиска. Эта гарантия основывается на том, что строка QueryString не пуста. Строка QueryString будет содержать действительную SQL-команду только в том случае, если выбрано хотя бы одно поле. Кроме того, данный метод позволяет закрыть форму независимо от заданных пользователем полей с помощью щелчка на кнопке Cancel. Необходимость выполнения поиска определяется по значению булевой переменной FindPressed, которая устанавливается равной True после щелчка на кнопке Find.

Если пользователь щелкнет на кнопке Find, SQL-команда будет передана вызывающей форме.

Резюме

На этом завершается рассмотрение приложения Inventory Manager. В данной главе продемонстрирован типичный вариант проектирования двухуровневого приложения типа клиент/сервер. Именно на двухуровневую модель и ориентируется большинство разработчиков систем подобного типа. Однако с распространением Internet и сетевых технологий все б*о*льшую популярность приобретает трехуровневая модель, которая подробно обсуждается в следующих главах.

Глава

34

Диспетчер клиента: разработка приложения по технологии MIDAS

Проектирование приложения сервера 829

Проектирование клиентского приложения 832

Резюме 849

В главе 33, “Inventory Manager: пример разработки приложения с архитектурой клиент/сервер”, рассматривались методы разработки двухуровневых приложений, а в этой главе речь пойдет о создании трехуровневого приложения с помощью технологии MIDAS, представленной в главе 32, “Разработка приложений MIDAS”. Хотелось бы обратить внимание на простоту использования компонентов MIDAS, а также на применение модели “портфеля” для автономной работы.

Разрабатываемое приложение будет связано с использованием модели портфеля. Это приложение представляет собой диспетчер клиента. Довольно часто торговые агенты выполняют большой объем работы автономно, причем им приходится трудиться даже во время деловых поездок. Список клиентов, с которыми они работают, важен как для торговых агентов, так и для компании, представителями которой они являются. Поэтому информацию о клиентах, возможно, следует разместить на сервере, принадлежащем компании. Однако сможет ли в таком случае торговый агент использовать эти данные, не подключаясь к сети? И каким образом торговый агент сможет обновить информацию компании теми новыми данными, которые пока хранятся у него?

Благодаря компоненту TClientDataSet можно создавать приложения-портфели с *внутренним кэшированием*. Это позволит торговым агентам загружать данные (или подмножество данных), с которыми они смогут работать автономно. По возвращении из командировки они смогут “выгрузить” из портфеля любые изменения и внести их в базу данных. В этой главе описывается процесс построения простого инструмента управления клиентом.

Проектирование приложения сервера

Приложение сервера разрабатывается с применением той же процедуры, которая рассматривалась в главе 32, “Разработка приложений MIDAS”. Здесь вы снова встретитесь с компонентом TRemoteDataModule, экземпляр которого имеет имя CustomerRemoteDataModule и содержит компоненты TSession, TDatabase, TQuery и TDataSetProvider. Экземпляр ssnCust компонента TSession предназначен для обработки способов создания нескольких экземпляров (его свойство AutoSessionName имеет значение True). Экземпляр DbCust компонента TDatabase используется для подключения клиента к базе данных без отображения диалогового окна регистрации пользователя. Экземпляр qryCust компонента TQuery возвращает результирующий набор в таблицу клиента. Экземпляр prvCust связан с qryCust через свойство DataSet. В разрабатываемом приложении используется таблица Customer, представленная в предыдущей главе.

Исходный код удаленного модуля данных приведен в листинге 34.1.

Листинг 34.1. Исходный код удаленного модуля данных

```
unit CustRDM;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ComServ, ComObj, VCLCom, StdVcl, DataBkr,
  DBClient, CustServ_TLB, Db, DBTables, Provider;

type
  TFilterType = (ftNone, ftCity, ftState);

  TCustomerRemoteDataModule = class(TRemoteDataModule,
```

```
    ICustomerRemoteDataModule)
    ssnCust: TSession;
    dbCust: TDatabase;
    qryCust: TQuery;
    prvCust: TDataSetProvider;
  private
    FFilterStr: String;
    FFilterType: TFilterType;
  public
    { Открытые объявления }
  protected
    procedure FilterByCity(const ACity: WideString; out Data: OleVariant);
      safecall;
    procedure FilterByState(const AStateStr: WideString; out Data: OleVariant);
      safecall;
    procedure NoFilter(out Data: OleVariant); safecall;
  end;

var
  CustomerRemoteDataModule: TCustomerRemoteDataModule;

implementation

{ $R *.DFM}

procedure TCustomerRemoteDataModule.FilterByCity(const ACity: WideString;
  out Data: OleVariant);
begin
  FFilterType  := ftCity;
  FFilterStr   := ACity;
  qryCust.Close;
  qryCust.SQL.Clear;
  qryCust.SQL.Add(Format('select * from CUSTOMER where CITY = "%s"', [ACity]));
  qryCust.Open;
  Data := prvCust.Data;
end;

procedure TCustomerRemoteDataModule.FilterByState(
  const AStateStr: WideString; out Data: OleVariant);
begin
  FFilterType  := ftState;
  FFilterStr   := AStateStr;
  qryCust.Close;
  qryCust.SQL.Clear;
  qryCust.SQL.Add(Format('select * from CUSTOMER where STATE = "%s"',
    [AStateStr]));
  qryCust.Open;
  Data := prvCust.Data;
end;
```

```
procedure TCustomerRemoteDataModule.NoFilter(out Data: OleVariant);
begin
  FFiltertype := ftNone;
  qryCust.Close;
  qryCust.SQL.Clear;
  qryCust.SQL.Add('select * from CUSTOMER');
  qryCust.Open;
  Data := prvCust.Data;
end;

initialization
  TComponentFactory.Create(ComServer, TCustomerRemoteDataModule,
    Class_CustomerRemoteDataModule, ciMultiInstance, tmSingle);
end.
```

В листинге 34.1 представлены три метода, добавленные в класс `TCustomerRemoteDataModule`. Реально эти методы были добавлены в интерфейс `ICustomerRemoteDateModule` с помощью редактора библиотеки типов (Type Library Editor), которым, в свою очередь, созданы методы реализации для класса `TCustomerRemoteDataModule` (рис. 34.1). В редакторе библиотеки типов были определены методы и их параметры, а затем добавлен код в каждый метод реализации, созданный Delphi. Исходный код библиотеки типов можно найти в файле `CustServ_TLV.pas`.

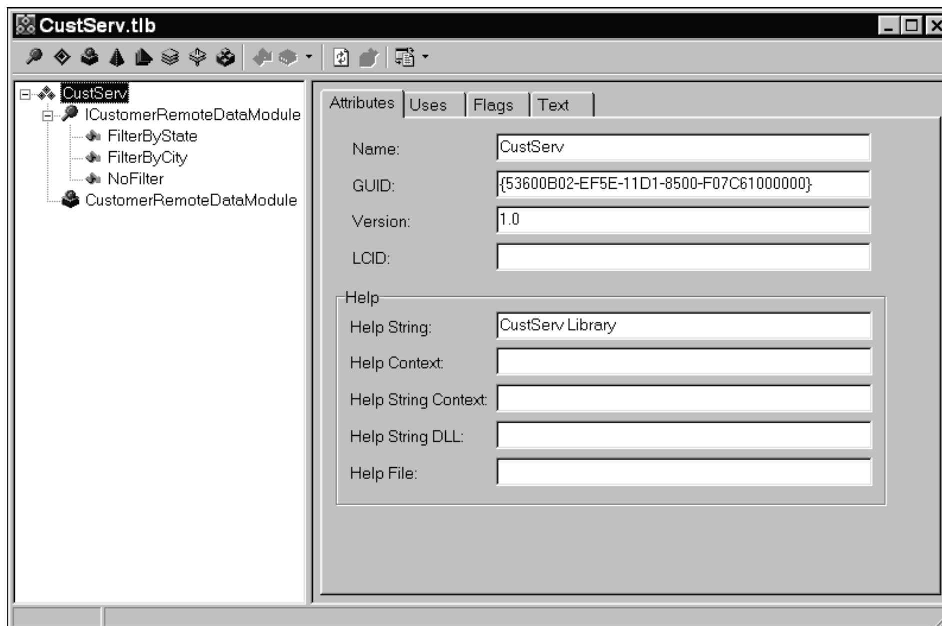


Рис. 34.1. Редактор библиотеки типов

На заметку

Этот пример основывается на демонстрационной программе, написанной для Delphi 4. Чтобы конвертировать приложение сервера, созданное с использованием MIDAS Delphi 4, в версию Delphi 5, необходимо выполнить несколько шагов. Их последовательность подробно описана в интерактивной справочной системе в разделе “Converting MIDAS Applications”.

Методы FilterByCity() и FilterByState() позволяют клиенту загрузить подмножество записей большой таблицы. Такой подход не лишен смысла, поскольку иногда торговому агенту не нужна загрузка целого списка, например, если он собирается посетить только один населенный пункт. Обоим методам передается один строковый параметр, который используется для задания значения фильтра. Функция NoFilter() удаляет любые фильтры, примененные к таблице.

Для того чтобы ограничить диапазон записей, возвращаемых клиенту, эти методы производят фильтрацию со стороны сервера. В качестве альтернативного варианта пользователь может заказать выполнение фильтрации со стороны клиента. Такой подход подразумевает доступ торгового агента к полному результирующему набору, с возможностью "отфильтровать" лишь требуемые ему записи. Далее будут проиллюстрированы оба способа.

Проектирование клиентского приложения

В клиентском приложении содержится модуль данных и главная форма. Сначала рассмотрим модуль данных.

Модуль данных клиента

Модуль данных для приложения Client Tracker иллюстрирует сразу несколько приемов. Во-первых, вы узнаете, как реализовать модель портфеля. Во-вторых, вы поймете, как сделать его режим (работа в системе или автономный (online/offline)) постоянным. Другими словами, после того как пользователь закроет приложение, оно должно "вспомнить" свое состояние при последующем запуске. Благодаря этому при работе в автономном режиме приложение не будет предпринимать попытку подключиться к серверу. Кроме того, будет продемонстрировано выполнение фильтрации на стороне клиента. Когда пользователь подключен к сети, приложение будет выполнять фильтрацию на стороне сервера, а когда он работает в автономном режиме, фильтрация будет выполняться на стороне клиента. Исходный код класса CustomerDataModule представлен в листинге 34.2.

Листинг 34.2. Текст модуля данных клиентского приложения

```
unit CustDM;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, DBClient, MConnect, Db;

const
  cFileName        = 'CustData.cds';

  cRegIniFile      = 'Software\DDG Client App';
  cRegSection      = 'Startup Config';
  cRegOnlineIdent  = 'Run Online';

type
```

```
TFilterType = (ftNone, ftByCity, ftByState);

TAddErrorToClientEvent = procedure(const AFieldName, OldStr, NewStr,
CurStr, ErrMsg: String) of Object;

TCustomerDataModule = class(TDataModule)
  cdsCust: TClientDataSet;
  dcomCust: TDCOMConnection;
  cdsCustCUSTOMER_ID: TIntegerField;
  cdsCustFNAME: TStringField;
  cdsCustLNAME: TStringField;
  cdsCustCREDIT_LINE: TSmallintField;
  cdsCustWORK_ADDRESS: TStringField;
  cdsCustALT_ADDRESS: TStringField;
  cdsCustCITY: TStringField;
  cdsCustSTATE: TStringField;
  cdsCustZIP: TStringField;
  cdsCustWORK_PHONE: TStringField;
  cdsCustALT_PHONE: TStringField;
  cdsCustCOMMENTS: TMemoField;
  cdsCustCOMPANY: TStringField;
  procedure CustomerDataModuleCreate(Sender: TObject);
  procedure cdsCustReconcileError(DataSet: TClientDataSet;
    E: EReconcileError; UpdateKind: TUpdateKind;
    var Action: TReconcileAction);
  procedure CustomerDataModuleDestroy(Sender: TObject);
  procedure cdsCustFilterRecord(DataSet: TDataSet; var Accept: Boolean);
private
  FFilterType: TFilterType;
  FFilterStr: String;
  FOnAddErrorToClient: TAddErrorToClientEvent;

  function GetOnline: Boolean;
  procedure SetOnline(const Value: Boolean);
  { Закрытые объявления }
protected
  function GetChangeCount: Integer;

public
  procedure EditClient;
  procedure AddClient;
  procedure SaveClient;
  procedure CancelClient;
  procedure DeleteClient;
  procedure ApplyUpdates;
  procedure CancelUpdates;
  procedure First;
  procedure Previous;
  procedure Next;
  procedure Last;
```

```
    function IsBOF: Boolean;
    function IsEOF: Boolean;

    procedure FilterByState;
    procedure FilterByCity;
    procedure NoFilter;

    property ChangeCount: Integer read GetChangeCount;
    property Online: Boolean read GetOnline write SetOnline;

    property OnAddErrorToClient: TAddErrorToClientEvent
      read FOnAddErrorToClient
      write FOnAddErrorToClient;

  end;

var
  CustomerDataModule: TCustomerDataModule;

implementation
uses MainCustFrm, Registry;

{ $R *.DFM}

procedure TCustomerDataModule.AddClient;
begin
  cdsCust.Insert;
end;

procedure TCustomerDataModule.ApplyUpdates;
begin
  cdsCust.ApplyUpdates(-1);
end;

procedure TCustomerDataModule.CancelClient;
begin
  cdsCust.Cancel;
end;

procedure TCustomerDataModule.CancelUpdates;
begin
  cdsCust.CancelUpdates;
end;

procedure TCustomerDataModule.DeleteClient;
begin
  if MessageDlg('Are you sure you want to delete the current record?',
    mtConfirmation, [mbYes, mbNo], 0) = mrYes then cdsCust.Delete;
end;
```

```
procedure TCustomerDataModule.EditClient;
begin
  cdsCust.Edit;
end;

function TCustomerDataModule.IsBOF: Boolean;
begin
  Result := cdsCust.Bof;
end;

function TCustomerDataModule.IsEOF: Boolean;
begin
  Result := cdsCust.Eof;
end;

procedure TCustomerDataModule.First;
begin
  cdsCust.First;
end;

procedure TCustomerDataModule.Last;
begin
  cdsCust.Last;
end;

procedure TCustomerDataModule.Next;
begin
  cdsCust.Next;
end;

procedure TCustomerDataModule.Previous;
begin
  cdsCust.Prior;
end;

procedure TCustomerDataModule.SaveClient;
begin
  cdsCust.Post;
end;

procedure TCustomerDataModule.cdsCustReconcileError(
  DataSet: TClientDataSet; E: EReconcileError; UpdateKind: TUpdateKind;
   var Action: TReconcileAction);
{ Если возникает ошибка, обновляем соответствующий список Tlistview
  в главной форме с использованием ошибочных данных.  }
var
  CurStr, NewStr, OldStr: String;
  i: integer;
  V: Variant;
```

```
procedure SetString(V: Variant; var S: String);
{ Необходимо проверить значение параметра V на равенство null, которое
  возвращается в том случае, если поле таблицы имело значение null.
  Это нужно сделать обязательно, поскольку операция приведения значения
  null к строковому типу не может быть выполнена. }
begin

  if VarIsNull(V) then
    S := EmptyStr
  else
    S := String(V);
end;

begin
  for i := 0 to DataSet.FieldCount - 1 do
  begin

    V := DataSet.Fields[i].NewValue;
    SetString(V, NewStr);

    V := DataSet.Fields[i].CurValue;
    SetString(V, CurStr);

    V := DataSet.Fields[i].OldValue;
    SetString(V, OldStr);

    if NewStr <> CurStr then
      if Assigned(FOnAddErrorToClient) then
        FOnAddErrorToClient(DataSet.Fields[i].FieldName, OldStr, NewStr,
          CurStr, E.Message)
  end;
  //Обновление записи и удаление изменения из журнала изменений.
  Action := raRefresh;
end;

function TCustomerDataModule.GetChangeCount: Integer;
begin
  Result := cdsCust.ChangeCount;
end;

function TCustomerDataModule.GetOnline: Boolean;
begin
  Result := dcomCust.Connected;
end;

procedure TCustomerDataModule.SetOnline(const Value: Boolean);
begin

  if Value = True then
  begin
```

```
    dcomCust.Connected := True;

    if cdsCust.ChangeCount > 0 then begin
      ShowMessage('Your changes must be applied before going online');
        cdsCust.ApplyUpdates(-1);
    end;
    cdsCust.Refresh;

  end
  else begin
    cdsCust.FileName := cFileName;
    dcomCust.Connected := False;
  end;
end;

procedure TCustomerDataModule.CustomerDataModuleCreate(Sender: TObject);
{ Определяем, в каком режиме - online или offline - пользователь оставил
  приложение в последнем сеансе работы, и перезапускаем приложение
  в том же режиме. }
var
  RegIniFile: TRegIniFile;
  IsOnline: Boolean;
begin
  RegIniFile := TRegIniFile.Create(cRegIniFile);
  try
    IsOnline := RegIniFile.ReadBool(cRegSection, cRegOnlineIdent, True);
  finally
    RegIniFile.Free;
  end;

  if IsOnline then
  begin
    dcomCust.Connected := True;
    cdsCust.Open;
  end
  else begin
    cdsCust.FileName := cFileName;
    cdsCust.Open;
  end;

end;

procedure TCustomerDataModule.CustomerDataModuleDestroy(Sender: TObject);
{ Сохраняем в реестре состояние приложения: online/offline.
  Когда пользователь снова запустит приложение, оно "вспомнит"
  состояние, в котором его оставили. }
var
  RegIniFile: TRegIniFile;
begin
  RegIniFile := TRegIniFile.Create(cRegIniFile);
```

```
  try
    RegIniFile.WriteBool(cRegSection, cRegOnlineIdent, Online);
  finally
    RegIniFile.Free;
  end;
end;

procedure TCustomerDataModule.FilterByCity;
{ Если есть подключение к сети, разрешаем серверу применять фильтр, чтобы
  выполнять выборку только нужных нам записей. В противном случае применяем
  фильтр к результирующему набору данных клиента cdsCust в памяти. }
var
  CityStr: String;
  Data: OleVariant;
begin
  InputQuery('Filter on City', 'Enter City: ', CityStr);
  FFilterStr := CityStr;

  if Online then
  begin
    dcomCust.AppServer.FilterByCity(CityStr, Data);
    cdsCust.Refresh;
  end
  else begin
    FFilterType      := ftByCity;
    cdsCust.Filtered := True;
    cdsCust.Refresh;
  end;
end;

procedure TCustomerDataModule.FilterByState;
{ Если есть подключение к сети, разрешаем серверу применять фильтр,
  чтобы выполнять выборку только нужных нам записей. В противном случае
  применяем фильтр к результирующему набору данных клиента cdsCust в памяти }

var
  StateStr: String;
  Data: OleVariant;
begin
  InputQuery('Filter on State', 'Enter State: ', StateStr);
  FFilterStr := StateStr;

  if Online then
  begin
    dcomCust.AppServer.FilterByState(StateStr, Data);
    cdsCust.Refresh;
  end
  else begin
    FFilterType      := ftByState;
    cdsCust.Filtered := True;
```

```
    cdsCust.Refresh;
  end;
end;

procedure TCustomerDataModule.NoFilter;
{ Если есть подключение к сети, разрешаем серверу применять фильтр,
  чтобы выполнять выборку только нужных нам записей. В противном случае
  применяем фильтр к результирующему набору данных клиента cdsCust в памяти. }

var
  Data: OleVariant;
begin

  if Online then
  begin
    dcomCust.AppServer.NoFilter(Data);
    cdsCust.Refresh;
  end
  else begin
    FFilterType      := ftNone;
    cdsCust.Filtered := False;
    cdsCust.Refresh;
  end;
end;

procedure TCustomerDataModule.cdsCustFilterRecord(DataSet: TDataSet; var Accept:
Boolean);
begin
  case FFilterType of
    ftByCity:  Accept := DataSet.FieldByName('CITY').AsString = FFilterStr;
    ftByState: Accept := DataSet.FieldByName('STATE').AsString = FFilterStr;
    ftNone:  Accept := True;
  end;
end;

end.
```

Начальное формирование соединения

Большинство методов, содержащихся в листинге 34.2, довольно просты и предназначены для выполнения задач навигации или манипуляций набором данных клиента cdsCust. Обратите внимание на то, что все операции с объектом cdsCust выполняются в модуле данных, а не путем непосредственного обращения со стороны каких-либо форм. Таким образом обеспечивается строгое следование методологии объектно-ориентированного программирования. И хотя это условие не является обязательным для работы в среде Delphi, мы поступаем так ради согласованности и централизации логики работы с базой данных.

Класс CustomerDataModule содержит экземпляр dcomCust компонента TDCOMConnection и экземпляр cdsCust компонента TClientDataSet. Экземпляр dcomCust подключается к приложению сервера с помощью свойства ServerName, в котором содержится значение CustServ.CustomerRemoteDataModule.

Экземпляр cdsCust связывается с запросом qryCust модуля удаленных данных с помощью свойства ProviderName. Благодаря этому устанавливается соединение, необходимое для того, чтобы сервер и клиент приложения MIDAS начали работать и осуществлять между собой взаимодействие. Однако для того чтобы воспользоваться преимуществами этой технологии, нужно написать некоторый дополнительный код.

Согласование разногласий

После передачи на сервер изменений из клиентского приложения могут возникнуть ошибки (особенно в модели портфеля,) связанные с тем, что некоторую запись уже модифицировал другой пользователь. Такие ошибки могут быть обработаны либо сервером, либо клиентом. Если ошибка обрабатывается клиентом, сервер передает информацию об ошибке обратно клиенту через обработчик события OnReconcileError компонента TClientDataSet. В этом обработчике события доступно несколько параметров, которые рассматриваются чуть ниже. Свойство OnReconcileError ссылается на метод TReconcileErrorEvent, определяемый следующим образом:

```
TReconcileErrorEvent = procedure(DataSet: TClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind;
  var Action: TReconcileAction) of object;
```

Параметр DataSet указывает на набор данных, в котором встретилась ошибка. EReconcileError — это класс исключительных ситуаций для ошибок в наборах данных клиента. Этот класс можно использовать подобно любому другому классу исключительных ситуаций. Параметр UpdateKind может принимать любое из значений, перечисленных в табл. 34.1 (эту информацию можно получить в справочной системе Delphi).

Таблица 34.1. Допустимые значения параметра TUpdateKind

Значение TUpdateKind	Описание
ukModify	Кэшированное обновление является модификацией содержимого записи
ukInsert	Кэшированное обновление является вставкой новой записи
ukDelete	Кэшированное обновление является удалением записи

Параметр Action имеет тип TReconcileAction. Установив значение параметра Action равным raRefresh, клиентское приложение отменяет любые изменения, внесенные пользователем, и обновляет свою копию результата таким образом, чтобы она совпадала с копией сервера. Именно этот вариант и реализован в данном примере. Другие возможные значения свойства Action приведены в табл. 34.2, которая также взята из справочной системы Delphi (в ней вы можете также получить дополнительную информацию о согласовании разногласий).

Таблица 34.2. Допустимые значения параметра TReconcileAction

Значение TReconcileAction	Описание
raSkip	Пропускает запись, которая вызвала появление ошибки, и не использует изменения из журнала изменений
raAbort	Прерывает операцию согласования разногласий
raMerge	Объединяет обновленную запись с записью на сервере

Значение `TReconcileAction`	**Описание**
`raCorrect`	Заменяет текущую обновленную запись значением записи в обработчике события
`raCancel`	Отменяет все изменения для этой записи, возвращаясь к исходным значениям полей
`raRefresh`	Отменяет все изменения для этой записи, заменяя ее текущими значениями с сервера

Внутри обработчика события `OnReconcileError` можно обращаться к свойствам `OldValue`, `NewValue` и `CurValue` каждого поля набора данных клиента. Эти свойства рассмотрены в главе 32, "Разработка приложений MIDAS".

Обработчик события `OnReconcileError` для объекта `cdsCust` — `cdsCustReconcileError()` — считывает новое, старое и текущее значения любого поля, для которого во время операции обновления клиенту была обнаружена ошибка согласования. Затем вызывается метод, на который указывает объект `FOnAddErrorToClient`. Это указатель типа `TAddErrorToClientEvent`, определение которого можно найти в начале листинга 34.2. Позже, при рассмотрении главной формы приложения `MainCustForm`, будет продемонстрирована реализация метода `TAddErrorToClientEvent` и его назначение объекту `FOnAddErrorToClient`. Это еще один пример хранения модуля данных независимо от элементов пользовательского интерфейса.

Установка и определение режима работы клиента

Клиентское приложение переводится в состояние соединения с сервером (online) или в автономное состояние (offline) при выполнении метода записи `SetOnline()` свойства `Online`, имеющего тип `Boolean`.

Метод `SetOnline()` устанавливает значение свойства `dcomCust.Connected` равным `True`, если пользователь переходит в режим соединения с сервером. Если предыдущее состояние клиента было автономным, то в базу данных сервера будут внесены все текущие изменения. Если при этом возникнут ошибки, то будет вызван обработчик события `cdsCust.OnReconcileError`. Если же пользователь собирается перейти в автономный режим работы, значение свойства `dcomCust.Connected` устанавливается равным `False`. Набор данных клиента (объект `cdsCust`) по-прежнему будет работать с копией данных, размещенной в памяти. И, на самом деле, поскольку имя файла задано в свойстве `cdsCust.FileName`, данные можно сохранить локально, записав их в обычный файл.

Метод чтения состояния клиента `GetOnline()` просто возвращает значение `True`, если пользователь работает в режиме подключения к серверу.

На заметку

Свойство `TClientDataSet.FileName` можно использовать лишь в версиях Delphi 4 и 5. Если вы работаете в среде Delphi 3, аналогичные действия можно выполнить с помощью методов `SaveToFile()` и `LoadFromFile()` объекта `TClientDataSet`.

Сохранение режима работы клиента

Обработчики событий `OnCreate` и `OnDestroy` класса `CustomerDataModule` гарантируют, что клиентское приложение будет запущено в том же режиме (в режиме подключения к серверу или автономном), в котором оно находилось в момент завершения предыдущего сеанса работы. Это

достигается за счет сохранения информации о состоянии в системном реестре, соответствующее значение которого опрашивается при каждом запуске приложения. В самом начале листинга 34.2 определены константы, указывающие раздел и ключи системного реестра.

Фильтрация записей

Класс CustomerDataModule разрешает пользователю отфильтровывать записи по названию города или штата, в котором живут его заказчики. Фильтрация на стороне клиента выполняется в том случае, когда приложение работает в автономном режиме. Если же клиент находится в состоянии подключения к серверу, то выполнение функции фильтрации возлагается на сервер. Здесь следует обратить внимание на то, что, когда клиент, находясь в режиме подключения, устанавливает фильтр, при переходе в автономное состояние на компьютере клиента локально будут сохранены только те записи, которые были выбраны этим фильтром.

Методы FilterByCity() и FilterByState(), в свою очередь, вызывают методы приложения сервера, рассмотренные выше. Причем эти методы вызываются только в том случае, если пользователь находится в состоянии подключения к серверу. Если же он работает в автономном режиме, фильтрация выполняется посредством свойства Filter и обработчика события OnFilterRecord объекта TClientDataSet.

Главная форма клиентского приложения

Главная форма клиентского приложения очень проста. Ее исходный текст приводится в листинге 34.3.

Листинг 34.3. Главная форма клиентского приложения — модуль MainCustFrm.pas

```
unit MainCustFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, DBNAVSTATFRM, Db, StdCtrls, DBCtrls, Mask, ComCtrls, Menus,
  ImgList, ToolWin, DBMODEFRM, Grids, DBGrids;

type

  TMainCustForm = class(TDBNavStatForm)
    pcClients: TPageControl;
    dsClientDetail: TTabSheet;
    lblFirstName: TLabel;
    lblLastName: TLabel;
    lblCreditLine: TLabel;
    lblWorkAddress: TLabel;
    lblAltAddress: TLabel;
    lblCity: TLabel;
    lblState: TLabel;
    lblZipCode: TLabel;
    lblWorkPhone: TLabel;
    lblAltPhone: TLabel;
```

```
lblCompany: TLabel;
dbeFirstName: TDBEdit;
dbeLastName: TDBEdit;
dbeCreditLine: TDBEdit;
dbeWorkAddress: TDBEdit;
dbeAltAddress: TDBEdit;
dbeCity: TDBEdit;
dbeState: TDBEdit;
dbeZipCode: TDBEdit;
dbeWorkPhone: TDBEdit;
dbeAltPhone: TDBEdit;
dbeCompany: TDBEdit;
tsComments: TTabSheet;
dbmComments: TDBMemo;
dsClients: TDataSource;
SaveDialog1: TSaveDialog;
OpenDialog1: TOpenDialog;
mmiSave: TMenuItem;
N3: TMenuItem;
mmiApplyUpdates: TMenuItem;
mmiCancelUpdates: TMenuItem;
mmiMode: TMenuItem;
mmiOffline: TMenuItem;
mmiOnline: TMenuItem;
tsErrors: TTabSheet;
lvClient: TListView;
mmiExit: TMenuItem;
mmiFilter: TMenuItem;
mmiByState: TMenuItem;
mmiByCity: TMenuItem;
mmiNoFilter: TMenuItem;
tsClientList: TTabSheet;
DBGrid1: TDBGrid;
procedure sbAcceptClick(Sender: TObject);
procedure sbCancelClick(Sender: TObject);
procedure sbInsertClick(Sender: TObject);
procedure sbEditClick(Sender: TObject);
procedure sbDeleteClick(Sender: TObject);
procedure sbFirstClick(Sender: TObject);
procedure sbPrevClick(Sender: TObject);
procedure sbNextClick(Sender: TObject);
procedure sbLastClick(Sender: TObject);
procedure FormCreate(Sender: TObject);
procedure mmiOnlineClick(Sender: TObject);
procedure mmiApplyUpdatesClick(Sender: TObject);
procedure mmiCancelUpdatesClick(Sender: TObject);
procedure dsClientsDataChange(Sender: TObject; Field: TField);
procedure Exit1Click(Sender: TObject);
procedure mmiExitClick(Sender: TObject);
```

```
    procedure mmiByStateClick(Sender: TObject);
    procedure mmiByCityClick(Sender: TObject);
    procedure mmiNoFilterClick(Sender: TObject);
  private
    procedure SetControls;
    procedure GoToOnlineMode;
    procedure GoToOfflineMode;

  public
    procedure AddErrorToClient(const aFieldName, aOldValue, aNewValue,
      aCurValue, aErrorStr: String);
  end;

var
  MainCustForm: TMainCustForm;

implementation

uses CustDM;

{ $R *.DFM}

procedure TMainCustForm.AddErrorToClient(const aFieldName, aOldValue,
  aNewValue, aCurValue, aErrorStr: String);
{ Этот метод используется для добавления элемента TListItem в список aLV
  типа TListView. Добавляемые сюда элементы индицируют ошибки,
  возникающие при выполнении операции обновления данных сервера. }
var
  NewItem: TListItem;
begin
  NewItem := lvClient.Items.Add;
  NewItem.Caption := aFieldName;
  NewItem.SubItems.Add(aOldValue);
  NewItem.SubItems.Add(aNewValue);
  NewItem.SubItems.Add(aCurValue);
  NewItem.SubItems.Add(aErrorStr);
end;

procedure TMainCustForm.SetControls;
begin
  { Настройка кнопок навигации в соответствии с режимом формы. }
  sbFirst.Enabled := not CustomerDataModule.IsBof;
  sbLast.Enabled  := not CustomerDataModule.IsEof;
  sbPrev.Enabled  := not CustomerDataModule.IsBof;
  sbNext.Enabled  := not CustomerDataModule.IsEof;

  { Синхронизация элементов меню навигации с кнопками панели инструментов. }
  mmiFirst.Enabled    := sbFirst.Enabled;
  mmiLast.Enabled     := sbLast.Enabled;
```

```
  mmiPrevious.Enabled := sbPrev.Enabled;
  mmiNext.Enabled     := sbNext.Enabled;
  { Установка других меню. }

  mmiApplyUpdates.Enabled  := mmiOnline.Checked and (FormMode = fmBrowse) and
    (CustomerDataModule.ChangeCount > 0);
  mmiCancelUpdates.Enabled := mmiOnline.Checked and (FormMode = fmBrowse) and
    (CustomerDataModule.ChangeCount > 0);

  mmiOnline.Checked  := CustomerDataModule.Online;
  mmiOffline.Checked := not mmiOnline.Checked;

  stbStatusBar.Panels[0].Text := Format('Changed Records: %d',
[CustomerDataModule.ChangeCount]);

  if CustomerDataModule.Online then
    stbStatusBar.Panels[2].Text := 'Working Online'
  else
    stbStatusBar.Panels[2].Text := 'Working Offline'

end;

procedure TMainCustForm.sbAcceptClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.SaveClient;
  SetControls;
end;

procedure TMainCustForm.sbCancelClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.CancelClient;
  SetControls;
end;

procedure TMainCustForm.sbInsertClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.AddClient;
  SetControls;
end;

procedure TMainCustForm.sbEditClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.EditClient;
  SetControls;
end;
```

```
procedure TMainCustForm.sbDeleteClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.DeleteClient;
  SetControls;
end;

procedure TMainCustForm.sbFirstClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.First;
  SetControls;
end;

procedure TMainCustForm.sbPrevClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.Previous;
  SetControls;
end;

procedure TMainCustForm.sbNextClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.Next;
  SetControls;
end;

procedure TMainCustForm.sbLastClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.Last;
  SetControls;
end;

procedure TMainCustForm.FormCreate(Sender: TObject);
begin
  inherited;
  CustomerDataModule.OnAddErrorToClient := AddErrorToClient;
  SetControls;

  { Заставляем эти свойства обратиться друг к другу. }
  mmiOnline.Tag := Longint(mmiOffline);
  mmiOffline.Tag := Longint(mmiOnline);
end;

procedure TMainCustForm.GoToOnlineMode;
begin
  CustomerDataModule.Online := True;
```

```
  SetControls;
end;

procedure TMainCustForm.GoToOfflineMode;
begin
  CustomerDataModule.Online := False;
  SetControls;
end;

procedure TMainCustForm.mmiOnlineClick(Sender: TObject);
var
  mi: TMenuItem;
begin
  inherited;
  mi := Sender as TMenuItem;

  if not mi.Checked then
  begin

    mi.Checked := not mi.Checked;
    TMenuItem(mi.Tag).Checked := not mi.Checked;

    if mi = mmiOnline then
    begin
      if mi.Checked then
        GoToOnlineMode
      else
        GoToOffLineMode
    end

    else begin
      if mi.Checked then
        GoToOfflineMode
      else
        GoToOnlineMode
    end;
  end;
end;

procedure TMainCustForm.mmiApplyUpdatesClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.ApplyUpdates;
  SetControls;
end;

procedure TMainCustForm.mmiCancelUpdatesClick(Sender: TObject);
begin
  inherited;
```

```
  CustomerDataModule.CancelUpdates;
  SetControls;
end;

procedure TMainCustForm.dsClientsDataChange(Sender: TObject;
  Field: TField);
begin
  inherited;
  SetControls;
end;

procedure TMainCustForm.Exit1Click(Sender: TObject);
begin
  inherited;
  Close;
end;

procedure TMainCustForm.mmiExitClick(Sender: TObject);
begin
  inherited;
  Close;
end;

procedure TMainCustForm.mmiByStateClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.FilterByState;
end;

procedure TMainCustForm.mmiByCityClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.FilterByCity;
end;

procedure TMainCustForm.mmiNoFilterClick(Sender: TObject);
begin
  inherited;
  CustomerDataModule.NoFilter;
end;

end.
```

Большинство методов класса TMainCustForm вызывает методы класса CustomerDataModule.

Обратите внимание на метод AddErrorToClient(). Он используется в качестве свойства OnAddErrorToClient класса CustomerDataModule. Обработчик события OnCreate класса TMainCustForm присваивает этот метод упомянутому свойству модуля данных. Метод AddErrorToClient() добавляет любые события в список TListView, расположенный в глав-

ной форме, чтобы пользователь мог их просмотреть. В списке TListView для элемента ошибки отображается имя поля, а также его старое, новое и текущее значения. Здесь также указывается строковое значение ошибки.

Простой метод SetControls() выполняет установку различных элементов управления в форме и обеспечивает их соответствующее состояние (доступное или недоступное). Назначение остальных методов разъясняется в комментариях, приведенных в листинге исходного кода.

Резюме

Несмотря на простоту приложения Client Tracker большинство связующих звеньев, необходимых для создания трехуровневых приложений, все же нашло отражение в этом примере. При необходимости вы могли бы также найти применение и таким специфическим средствам, как обратные вызовы и буферизация соединений, рассмотренным в главе 32, “Разработка приложений MIDAS”. Важно понять одно: разработка трехуровневого приложения с использованием технологии MIDAS не сложнее разработки двухуровневого или даже настольного приложения базы данных.

Глава

35 Разработка настольного приложения: сбор сведений об ошибках

Общие требования к приложению 851

Модель данных 852

Разработка модуля данных 852

Разработка интерфейса пользователя 869

Подготовка приложения к работе в Web 876

Резюме 876

В этой главе рассматриваются способы разработки настольного приложения для работы с базой данных. В предлагаемом вашему вниманию приложении, предназначенном для сбора сведений об ошибках, иллюстрируются методы, которые следует принять во внимание, особенно в том случае, если приложение будет использоваться в Internet. В данном приложении демонстрируются также некоторые приемы, позволяющие справиться с проблемами, возникающими при отделении пользовательского интерфейса от процедур обработки данных.

Благодаря простоте работы в среде Delphi разработка приложений баз данных не составляет особого труда. Однако при этом можно упустить из виду весьма существенные моменты, которые позднее, на этапе расширения базовых функций приложения, могут стать причиной головной боли разработчика. В этой главе будет показано, как учесть подобные аспекты при создании приложений, работающих с базами данных.

Общие требования к приложению

В этом разделе рассматриваются общие требования к приложению накопления сведений об ошибках в программном обеспечении. В данном случае не ставилась цель разработать законченный коммерческий инструмент отслеживания ошибок, просто была использована одна из практических задач, и проиллюстрированы возможные методы ее решения. Поэтому не следует ожидать от данного приложения большой практической пользы, поскольку основной акцент в нем сделан на демонстрации отдельных приемов, а не на обеспечении всей необходимой функциональности.

Возможность размещения в World Wide Web

Приложение сбора сведений об ошибках должно быть спроектировано таким образом, чтобы минимизировать затраты на разработку функций, отвечающих за его доступность в World Wide Web. Это значит, что пользовательский интерфейс должен быть полностью (а не почти полностью) отделен от логики базы данных. По сути, требуется обеспечить средства подключения к логике базы данных самых различных пользовательских интерфейсов. О том, как это делается, речь пойдет в главе 36, “Приложение с использованием компонентов WebBroker: сбор сведений об ошибках”, где мы рассмотрим создание Web-версии данного приложения.

Ввод данных о пользователе и регистрация

Приложение сбора сведений об ошибках включает таблицу с именами пользователей, которые имеют право зарегистрироваться в нем и начать работу. С помощью этого приложения пользователь может ввести сообщение о наличии программной ошибки. Кроме того, пользователь может пополнить таблицу имен сведениями о других пользователях и тем самым позволить им работать с этим приложением. В данной версии приложения не требуется, чтобы пользователи имели право редактировать или удалять информацию о пользователях.

При регистрации в данном приложении пользователи сообщают свое имя, причем список допустимых имен хранится в таблице `Users.db`. Процедура регистрации используется лишь для получения идентификатора `UserID`, который требуется для работы с зафиксированными сведениями об ошибках, и не является средством обеспечения безопасности.

Обработка, просмотр и фильтрация записей об ошибках

Пользователи могут добавлять, редактировать и удалять информацию об ошибках. Они также могут указывать интересующие их поля каждой из ошибок. Например, можно ввести дату регистрации ошибки и адресовать ее некоторому другому пользователю, определить статус ошибки, дать ее краткое описание, указать подробные сведения и источник. При этом имя пользователя, зарегистрировавшего ошибку, добавляется автоматически.

Способы работы с записями об ошибках

Для зарегистрированных ошибок можно добавлять описания действий по их устранению (примечания). Пользователи могут также просматривать подобные примечания, введенные ими самими или другими пользователями. Для заинтересованных сторон это очень удобный способ отслеживания прогресса в коррекции ошибок.

Другие функции пользовательского интерфейса

В приложении должны использоваться методы, обеспечивающие простоту и легкость понимания пользовательского интерфейса. Там, где это необходимо, можно использовать поля подстановки и всплывающие подсказки.

Модель данных

Модель данных для приложения отчета об ошибках показана на рис. 35.1. В этой модели содержатся следующие таблицы.

- `IDs` — представляет собой таблицу генерации ключей и отслеживает следующий доступный ключ для таблиц `Users`, `Bugs` и `Actions`.
- `Users` — содержит имена пользователей, имеющих право добавлять в базу данных сведения о регистрации ошибок.
- `Bugs` — хранит общую информацию об ошибках.
- `Actions` — содержит замечания о коррекции ошибок. Каждая ошибка может иметь несколько примечаний.
- `Status` — представляет собой справочную таблицу, используемую для назначения каждой ошибке определенного статуса.

Разработка модуля данных

Модуль данных является центральным звеном структуры приложения регистрации ошибок. Именно в модуле данных сосредоточен весь спектр действий, выполняемых с базой данных. Пользовательский интерфейс использует функциональные возможности модуля данных, обращаясь к его открытым методам и свойствам. В пользовательском интерфейсе нет ни одной прямой ссылки на компоненты доступа к данным, за исключением обеспечения доступа из инспектора объектов. Примером прямого обращения к компоненту доступа к данным мо-

жет служить свойство `DataSet` компонента `TDataSource`, который располагается в формах пользовательского интерфейса. Аналогичным образом, модуль данных никогда не должен обращаться к элементам пользовательского интерфейса.

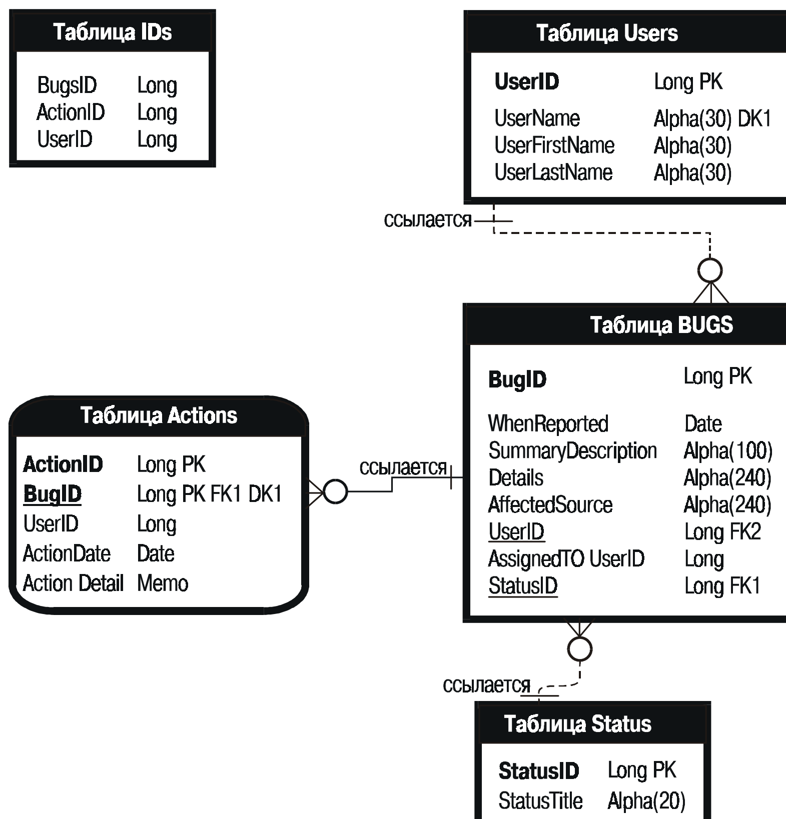


Рис. 35.1. Модель данных приложения регистрации ошибок

На заметку

При разработке приложений, в которых логику обработки данных требуется отделить от пользовательского интерфейса, местоположение компонента `TDataSource` не имеет принципиального значения. В данном приложении регистрации ошибок он помещен не в модуль данных, а в формы пользовательского интерфейса, поскольку он больше связан с функционированием пользовательского интерфейса, чем с доступом к данным. Однако при желании можно поступить и по-другому.

В листинге 35.1 содержится исходный текст модуля данных приложения регистрации ошибок.

Листинг 35.1. Модуль данных для приложения регистрации ошибок

```
unit DDGBugsDM;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
```

```
  Forms, Dialogs, Db, DBTables, HTTPApp, DBWeb;

type

  EUnableToObtainID = class(Exception);

  TDDGBugsDataModule = class(TDataModule)
    dbDDGBugs: TDatabase;
    tblBugs: TTable;
    tblUsers: TTable;
    tblStatus: TTable;
    tblActions: TTable;
    tblBugsBugID: TIntegerField;
    tblBugsWhenReported: TDateField;
    tblBugsSummaryDescription: TStringField;
    tblBugsDetails: TStringField;
    tblBugsAffectedSource: TStringField;
    tblBugsUserID: TIntegerField;
    tblBugsStatusID: TIntegerField;
    dsUsers: TDataSource;
    dsStatus: TDataSource;
    tblIDs: TTable;
    tblBugsUserNameLookup: TStringField;
    tblBugsAssignedToLookup: TStringField;
    tblUsersUserID: TIntegerField;
    tblUsersUserName: TStringField;
    tblUsersUserFirstName: TStringField;
    tblUsersUserLastName: TStringField;
    tblBugsAssignedToUserID: TIntegerField;
    dsBugs: TDataSource;
    wbdpBugs: TWebDispatcher;
    dstpBugs: TDataSetTableProducer;
    procedure DDGBugsDataModuleCreate(Sender: TObject);
    procedure tblBugsBeforePost(DataSet: TDataSet);
    procedure tblBugsFilterRecord(DataSet: TDataSet; var Accept: Boolean);
    procedure tblUsersBeforePost(DataSet: TDataSet);
    procedure tblBugsAfterInsert(DataSet: TDataSet);
    procedure wbdpBugswaShowAllBugsAction(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
    procedure wbdpBugswaIntroAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure wbdpBugswaUserNameAction(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
    procedure wbdpBugswaVerifyUserNameAction(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
  private
    FLoginUserID: Integer;
    FLoginUserName: String;

    function GetFilterOnUser: Boolean;
    procedure SetFilterOnUser(const Value: Boolean);
```

```
    function GetNumBugs: Integer;
  protected
    procedure PostAction(Sender: TObject; Action: TStrings);
  public

    // Методы работы с данными об ошибках
    procedure FirstBug;
    procedure LastBug;
    procedure NextBug;
    procedure PreviousBug;
    function IsLastBug: Boolean;
    function IsFirstBug: Boolean;
    function IsBugsTblEmpty: Boolean;
    procedure InsertBug;
    procedure DeleteBug;
    procedure EditBug;
    procedure SaveBug;
    procedure CancelBug;
    procedure SearchForBug;

    // Пользовательские функции
   { $IFNDEF DDGWEBBUGS}
    procedure AddUser;
   { $ENDIF}

    procedure PostUser(Sender: TObject);
    function GetUserFLName(AUserID: Integer): String;

    // Методы коррекции ошибок

    { $IFNDEF DDGWEBBUGS}
    procedure AddAction;
    { $ENDIF}

    procedure GetActions(AActions: TStrings);

    // Генерация идентификатора (ID)
    function GetDataSetID(const AFieldName: String): Integer;
    function GetNewBugID: Integer;
    function GetNewUserID: Integer;
    function GetNewActionID: Integer;

    // Функция регистрации
    function Login: Boolean;

    // Открытые свойства

    property LoginUserID: Integer read FLoginUserID;
    property FilterOnUser: Boolean read GetFilterOnUser write
      SetFilterOnUser;
    property NumBugs: Integer read GetNumBugs;
```

```
  end;

var
  DDGBugsDataModule: TDDGBugsDataModule;

implementation

{ $IFNDEF DDGWEBBUGS}
uses UserFrm, ActionFrm;
{ $ENDIF}

{ $R *.DFM}

// Вспомогательные функции

function IsInteger(IntVal: String): Boolean;
var
  v, code: Integer;
begin
  val(IntVal, v, code);
  Result := code = 0;
end;

procedure MemoFromStrings(AMemoField: TMemoField; AStrings: TStrings);
var
  Stream: TMemoryStream;
begin
  Stream := TMemoryStream.Create;
  try
    AStrings.SaveToStream(Stream);
    Stream.Seek(0, soFromBeginning);
    AMemoField.LoadFromStream(Stream);
  finally
    Stream.Free;
  end;
end;

procedure StringsFromMemo(AStrings: TStrings; AMemoField: TMemoField);
var
  Stream: TMemoryStream;
begin
  Stream := TMemoryStream.Create;
  try
    AMemoField.SaveToStream(Stream);
    Stream.Seek(0, soFromBeginning);
    AStrings.LoadFromStream(Stream);
  finally
    Stream.Free;
  end;
end;
```

```
// Внутренние методы

function TDDGBugsDataModule.GetFilterOnUser: Boolean;
begin
  Result := tblBugs.Filtered;
end;

procedure TDDGBugsDataModule.SetFilterOnUser(const Value: Boolean);
begin
 tblBugs.Filtered := Value;
end;

function TDDGBugsDataModule.GetNumBugs: Integer;
begin
  Result := tblBugs.RecordCount;
end;

// Методы идентификации

function TDDGBugsDataModule.GetDataSetID(const AFieldName: String): Integer;
const
  MaxAttempts = 50;
var
  Attempts: Integer;
  NextID: Integer;
begin
  tblIDs.Active := True;
  { Производится не более 50 попыток, после чего
    генерируется исключительная ситуация }
  Attempts := 0;
  while Attempts <= MaxAttempts do
  begin
    try
      Inc(Attempts);
      { Если другой пользователь редактирует таблицу, возникает ошибка. }
      tblIDs.Edit;
      { Если достигнут оператор Break, то попытка оказалась
        успешной. Выход из цикла. }
      Break;
    except
      on EDBEngineError do
      begin
        // Некоторая задержка
        Continue;
      end;
    end;
  end;

  if tblIDs.State = dsEdit then
  begin
    { Увеличиваем значение, полученное из таблицы, и восстанавливаем
```

```
      новое значение в таблице для следующей записи. }
    NextID := tblIDs.FieldByName(AFieldName).AsInteger;
    tblIDs.FieldByName(AFieldName).AsInteger := NextID + 1;
    TblIDs.Post;
    Result := NextID;
  end
  else
    Raise EUnableToObtainID.Create('Cannot create unique ID');
end;

function TDDGBugsDataModule.GetNewActionID: Integer;
begin
  Result := GetDataSetID('ActionsID');
end;

function TDDGBugsDataModule.GetNewBugID: Integer;
begin
  Result := GetDataSetID('BugsID');
end;

function TDDGBugsDataModule.GetNewUserID: Integer;
begin
  Result := GetDataSetID('UsersID');
end;

// Методы инициализации/регистрации

procedure TDDGBugsDataModule.DDGBugsDataModuleCreate(Sender: TObject);
begin
  { Эти таблицы открыты в требуемом порядке, так что связи
    "главная-детальная" (master-detail) не разрушаются. }
  dbDDGBugs.Connected := True;
  tblUsers.Active  := True;
  tblStatus.Active := True;
  tblBugs.Active   := True;
  tblActions.Active := True;
end;

function TDDGBugsDataModule.Login: Boolean;
var
  UserName: String;
begin
  InputQuery('Login', 'Enter User Name: ', UserName);
  Result := tblUsers.Locate('UserName', UserName, []);
  if Result then
  begin
    FLoginUserID   := tblUsers.FieldByName('UserID').AsInteger;
    FLoginUserName := tblUsers.FieldByName('UserName').AsString;
  end;
end;
```

```
// Методы работы с данными об ошибках

procedure TDDGBugsDataModule.FirstBug;
begin
  tblBugs.First;
end;

procedure TDDGBugsDataModule.LastBug;
begin
  tblBugs.Last;
end;

procedure TDDGBugsDataModule.NextBug;
begin
  tblBugs.Next;
end;

procedure TDDGBugsDataModule.PreviousBug;
begin
  tblBugs.Prior;
end;

function TDDGBugsDataModule.IsLastBug: Boolean;
begin
  Result := tblBugs.Eof;
end;

function TDDGBugsDataModule.IsFirstBug: Boolean;
begin
  Result := tblBugs.Bof;
end;

function TDDGBugsDataModule.IsBugsTblEmpty: Boolean;
begin
  { Если переменная RecordCount равна нулю, в таблице нет
    сведений ни об одной ошибке. }
  Result := tblBugs.RecordCount = 0;
end;

procedure TDDGBugsDataModule.InsertBug;
begin
  tblBugs.Insert;
end;

procedure TDDGBugsDataModule.DeleteBug;
var
  Qry: TQuery;
  BugID: Integer;
begin
  if MessageDlg('Delete Action?', mtConfirmation,
    [mbYes, mbNo], 0) = mrYes then
```

```
  begin
    BugID := tblBugs.FieldByName('BugID').AsInteger;
    { Для выполнения этих действий используется динамически
      создаваемый компонент TQuery. }
    Qry := TQuery.Create(self);
    try
      dbDDGBugs.StartTransaction;
      try
        { Сначала удаляются любые примечания, относящиеся к данной ошибке. }
        Qry.DatabaseName := dbDDGBugs.DataBaseName;
        Qry.SQL.Add(Format('DELETE FROM ACTIONS WHERE BugID = %d', [BugID]));
        Qry.ExecSQL;

        { Теперь ошибка удаляется из таблицы. }
        Qry.SQL.Clear;
        Qry.SQL.Add(Format('DELETE FROM BUGS WHERE BugID = %d', [BugID]));
        Qry.ExecSQL;

        tblBugs.Refresh;
        tblActions.Refresh;

        dbDDGBugs.Commit;
      except
        dbDDGBugs.Rollback;
        raise;
      end;
    finally
      Qry.Free;
    end;
  end;
end;

procedure TDDGBugsDataModule.EditBug;
begin
  tblBugs.Edit;
end;

procedure TDDGBugsDataModule.SaveBug;
begin
  tblBugs.Post;
end;

procedure TDDGBugsDataModule.CancelBug;
begin
  tblBugs.Cancel;
end;

procedure TDDGBugsDataModule.SearchForBug;
var
  BugStr: String;
begin
```

```
  InputQuery('Search for bug', 'Enter bug ID: ', BugStr);
  { Поиск ошибки. Введите ее идентификатор. }
  if IsInteger(BugStr) then
    if not tblBugs.Locate('BugID', StrToInt(BugStr), []) then
      MessageDlg('Bug not found.', mtInformation, [mbOK], 0);
end;

// Пользовательские методы

{ $IFNDEF DDGWEBBUGS}
procedure TDDGBugsDataModule.AddUser;
begin
  tblUsers.Insert;
  try
    if NewUserForm(PostUser) = mrCancel then
      tblUsers.Cancel;
  except
    { Возникло рассогласование. Таблица переводится в режим
      просмотра и снова генерируется исключительная ситуация }
    tblUsers.Cancel;
    raise;
  end;
end;
{ $ENDIF}

procedure TDDGBugsDataModule.PostUser(Sender: TObject);
begin
  if tblUsers.State = dsInsert then
    tblUsers.FieldByName('UserID').AsInteger := GetNewUserID;
  tblUsers.Post;
end;

function TDDGBugsDataModule.GetUserFLName(AUserID: Integer): String;
begin
  // Возвращает объединенные имя и фамилию
  if tblUsers.Locate('UserID', AUserID, []) then
    Result := Format('%s %s', [tblUsers.FieldByName('UserFirstName').AsString,
      tblUsers.FieldByName('UserLastName').AsString])
  else
    Result := EmptyStr;
end;

{ $IFNDEF DDGWEBBUGS}
procedure TDDGBugsDataModule.AddAction;
begin
  NewActionForm(PostAction);
end;
{ $ENDIF}

procedure TDDGBugsDataModule.GetActions(AActions: TStrings);
var
```

```
  Action: TStringList;
  ActionUserId: Integer;

begin
  Action := TStringList.Create;
  try
    with tblActions do
    begin
      tblActions.First;
      while not Eof do
      begin
        Action.Clear;
        ActionUserID := FieldByName('UserID').AsInteger;
        StringsFromMemo(Action, TMemoField(FieldByName('ActionDetail')));
        AActions.Add(Format('Action Added on: %s',
          [FormatDateTime('mmm dd, yyyy',
          FieldByName('ActionDate').AsDateTime)]));
        AActions.Add(Format('Action Added by: %s',
          [GetUserFLName(ActionUserID)]));
        AActions.Add(EmptyStr);
        AActions.AddStrings(Action);
        AActions.Add('=============================');
        AActions.Add(EmptyStr);
        tblActions.Next;
      end;          // Конец цикла while
    end;            // конец инструкции with
  finally
    Action.Free;
  end;
end;

procedure TDDGBugsDataModule.PostAction(Sender: TObject; Action: TStrings);
var
  BugID: Integer;
begin
  tblActions.Insert;
  try
    BugID := tblBugs.FieldByName('BugID').AsInteger;
    tblActions.FieldByName('ActionID').AsInteger      := GetNewActionID;
    tblActions.FieldByName('BugID').AsInteger         := BugID;
    tblActions.FieldByName('UserID').AsInteger        := LoginUserID;
    tblActions.FieldByName('ActionDate').AsDateTime   := Date;
    MemoFromStrings(TMemoField(tblActions.FieldByName('ActionDetail')),
      Action);
    tblActions.Post;
  except
    tblActions.Cancel;
    raise;
  end;
end;
```

```
// Обработчики событий

procedure TDDGBugsDataModule.tblBugsBeforePost(DataSet: TDataSet);
begin
  if tblBugs.State = dsInsert then
    tblBugs.FieldByName('BugID').AsInteger := GetNewBugID;
end;

procedure TDDGBugsDataModule.tblBugsFilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
begin
  Accept := tblBugs.FieldByName('UserID').AsInteger = FLoginUserID;
end;

procedure TDDGBugsDataModule.tblUsersBeforePost(DataSet: TDataSet);
begin
  if tblUsers.State = dsInsert then
    tblUsers.FieldByName('UserID').AsInteger := GetNewUserID;
end;

procedure TDDGBugsDataModule.tblBugsAfterInsert(DataSet: TDataSet);
begin
  tblBugs.FieldByName('UserID').AsInteger := FLoginUserID;
  tblBugs.FieldByName('UserNameLookup').AsString := FLoginUserName;
end;

end.
```

Инициализация приложения и регистрация

Как видно из листинга 35.2, класс `TDDGBugsDataModule` помещен в проект таким образом, чтобы он создавался в первую очередь. Затем вызывается его метод `Login()`, который определяет, продолжать ли выполнение данного приложения. Значение, возвращаемое этим методом, зависит от того, существует ли в таблице `Users.db` введенное имя пользователя (см. код метода `TDDGBugsDataModule.Login()` в листинге 35.1).

Для поддержки регистрации пользователя требуется модифицировать файл проекта приложения так, как показано в листинге 35.2.

Листинг 35.2. Файл проекта для приложения регистрации ошибок

```
program DDGBugs;

uses
  Forms,
  Dialogs,
  ChildFrm in '..\ObjRepos\CHILDFRM.pas' { ChildForm},
  DBModeFrm in '..\ObjRepos\DBMODEFRM.pas' { DBModeForm},
  DBNavStatFrm in '..\ObjRepos\DBNAVSTATFRM.pas' { DBNavStatForm},
  MainFrm in 'MainFrm.pas' { MainForm} ,
  UserFrm in 'UserFrm.pas' { UserForm} ,
```

```
  ActionFrm in 'ActionFrm.pas' { ActionForm} ,
  DDGBugsDM in '..\ Shared\DDGBugsDM.pas' { DDGBugsDataModule: TDataModule};

{ $R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TDDGBugsDataModule, DDGBugsDataModule);
  if DDGBugsDataModule.Login then
  begin
    Application.CreateForm(TMainForm, MainForm);
    Application.Run;
  end
  else
    MessageDlg('Invalid Login', mtError, [mbOk], 0);
    // Некорректная регистрация пользователя
end.
```

Генерирование ключей для таблиц Paradox

В приложении регистрации ошибок в качестве внутреннего интерфейса используется база данных Paradox, что привело к возникновению незначительной проблемы, которую необходимо обойти. Проблема заключается в автоинкрементных полях Paradox. Хотя, теоретически, автоинкрементные поля Paradox могут использоваться в качестве ключевых полей, на практике они оказываются весьма ненадежными. Как показывает опыт, их синхронизация с внешними ключами может легко нарушиться. Для того чтобы избежать их использования, можно создать собственные ключи на основе значений, содержащихся в таблице `IDs.db`.

В таблице `IDs.db` хранятся следующие доступные целые значения для ключей таблиц `Bugs`, `Users` и `Action`. Метод `TDDGBugsDataModule.GetDataSetID()` гарантирует, что специфическое ключевое поле таблицы `tblIDs` в режим редактирования может переключить только один пользователь. Тем самым гарантируется, что два пользователя при вставке записи никогда не получат идентичных значений ключа. Метод `GetDataSetID()` применяется для трех типов ключей. Это достигается за счет передачи имени поля для желаемого значения ключа. Следовательно, этот метод можно использовать для получения ключей таблиц `Bugs`, `Users` и `Action`. И в самом деле, этот метод вызывается методами `GetNewActionID()`, `GetNewBugID()` и `GetNewUserID()`. Этих три метода могут быть вызваны при отправке записи в одну из этих таблиц. Сами вызовы выполняются в обработчиках события `BeforePost` для таблиц `tblBugs` и `tblUsers`, а также в методе `PostAction()` для таблицы `tblActions`.

Процедуры работы с данными об ошибках

Под процедурами работы с данными об ошибках подразумеваются методы, отмеченные комментарием `// Методы работы с данными об ошибках`. Большинство этих функций (особенно методы навигации) не нуждается в дополнительных разъяснениях. Пожалуй, самым большим по объему является метод `DeleteBug()`. Он гарантирует, что любые примечания, относящиеся к конкретной ошибке, будут удалены до удаления соответствующей записи ошибки. Благодаря использованию механизма транзакций компонента `TDatabase` при воз-

никновении сбоя не происходит потери данных. Для того чтобы выполнить обработку транзакций с локальной базой данных Paradox, свойству TransIsolation компонента TDatabase необходимо задать значение tiDirtyRead, что и было сделано.

Просмотр и фильтрация ошибок

Каждый пользователь должен иметь возможность просматривать информацию обо всех ошибках или только о тех из них, которые принадлежат лично ему. Это реализуется с помощью свойства Filtered компонента tblBugs. Когда свойство tblBugs.Filtered принимает значение True, для каждой записи вызывается обработчик события tblBugs.OnFilterRecord. В рассматриваемом примере запись отображается только в том случае, если ее поле UserID совпадает со значением в глобальной переменной FLoginUserID. Обратите внимание на то, как для пользовательского интерфейса делается доступным свойство Filtered таблицы tblBugs. Чтобы избежать обращения к свойству tblBugs.Filtered из пользовательского интерфейса напрямую, доступ к нему обеспечивается через свойство TDDGBugsDataModule.FilterOnUser. Метод записи этого свойства, SetFilterOnUser(), выполняет присваивание значения свойству tblBugs.Filtered. Теперь уже нельзя утверждать наверняка, что формы не имеют прямого доступа к свойствам компонентов, которые размещаются в классе TDatamodules, поскольку библиотека VCL не придерживается строгих правил ограничения области видимости, характерных для объектно-ориентированного программирования.

Добавление пользователей

Добавление пользователей выполняется с помощью методов TDDGBugsDataModule.AddUser() и TDDGBugsDataModule.PostUser(). Для добавления данных о новом пользователе метод AddUser() активизирует простое диалоговое окно. Обратите внимание на то, как метод PostUser() передается функции NewUserForm(), которая вызывает пользовательскую форму. Тем самым иллюстрируется один из возможных способов избежать повторного обращения формы к активизировавшему ее модулю данных. Возникновение этой проблемы связано со стремлением защитить компоненты модуля данных от внешнего доступа. Для достижения этой цели существует множество путей, в том числе следующий. Метод NewUserForm() вызывает форму, определенную в модуле UserFrm.pas, исходный код которого представлен в листинге 35.3.

Листинг 35.3. Модуль UserFrm.pas: форма добавления пользователя

```
unit UserFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Mask, DBCtrls;

type

  TUserForm = class(TForm)
    lblUserName: TLabel;
```

```
    dbeUserName: TDBEdit;
    lblFirstName: TLabel;
    dbeFirstName: TDBEdit;
    lblLastName: TLabel;
    dbeLastName: TDBEdit;
    btnOK: TButton;
    btnCancel: TButton;
    procedure btnOKClick(Sender: TObject);
  private
    FPostUser: TNotifyEvent;
  public
    { Объявление открытых данных }
  end;

function NewUserForm(APostUser: TNotifyEvent): Word;

implementation
uses dbTables;
{ $R *.DFM}

function NewUserForm(APostUser: TNotifyEvent): Word;
var
  UserForm: TUserForm;
begin
  UserForm := TUserForm.Create(Application);
  try
    UserForm.FPostUser := APostUser;
    Result := UserForm.ShowModal;
  finally
    UserForm.Free;
  end;
end;

procedure TUserForm.btnOKClick(Sender: TObject);
begin
  if dbeUserName.Text = EmptyStr then begin
    MessageDlg('A user name is required.', mtWarning, [mbOK], 0);
    // Требуется указать имя пользователя
    dbeUserName.SetFocus;
    ModalResult := mrNone;
  end
  else begin
    try
      FPostUser(self);
    except
      on EDBEngineError do
      begin
        MessageDlg('User name already exists.', mtWarning, [mbOK], 0);
        // Имя пользователя уже существует
```

```
        dbeUserName.SetFocus;
        ModalResult := mrNone;
      end;
    end;
  end;
end;

end.
```

Как видно из листинга 35.3, метод NewUserForm() создает экземпляр компонента TUserForm и отображает созданную форму. Обратите внимание на то, что параметр APostUser присваивается полю FPostUser, имеющему тип TNotifyEvent. Объявив объект FPostUse указателем на метод TNofityEvent, указателю FPostUser можно поставить в соответствие метод PostUser() класса TDDGBugDataModule, поскольку тип метода PostUser() совпадает с определением типа TNotifyEvent. Эта концепция описана в главах 20, "Ключевые элементы VCL и информация о типах времени выполнения", и 21, "Создание пользовательских компонентов в Delphi".

По щелчку на кнопке OK вызывается метод btnOkClick(). После проверки того факта, что имя пользователя действительно введено, вызывается метод TDDGBugDataModule.PostUser() (на который ссылается указатель FPostUser), предназначенный для сохранения записи с именем пользователя (см. метод PostUser() в листинге 35.1). Если в методе PostUser() возникает ошибка, значит, введенное имя пользователя уже существует в базе данных. Это служит иллюстрацией еще одного преимущества передачи метода PostUser() форме TUserForm. Компонент TUserForm может обработать ошибку, возникшую в модуле данных. Эта концепция не идет вразрез с концепцией разработки компонентов. Модуль данных разрабатывается таким образом, чтобы он был полностью самодостаточным, однако при этом пользователям разрешается обрабатывать любые ошибки, возникающие внутри модуля данных.

Добавление сведений о действиях (примечаний)

Сведения о действиях в данном контексте оформляются как примечания, которые при необходимости присоединяются к каждой записи об ошибке. Любой пользователь может добавить примечание к любой ошибке. Метод TDDGBugsDataModule.AddAction() вызывает процедуру NewActionForm(), которая получает данные примечания от пользователя и добавляет их в базу данных. Метод NewActionForm() определен в модуле ActionFrm.pas, исходный код которого содержится в листинге 35.4.

Листинг 35.4. Модуль ActionFrm.pas: форма ввода примечаний

```
unit ActionFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;
```

```
type

  TPostActionEvent = procedure (Sender: TObject; Action: TStrings) of Object;

  TActionForm = class(TForm)
    memAction: TMemo;
    lblAction: TLabel;
    btnOK: TButton;
    btnCancel: TButton;
    procedure btnOKClick(Sender: TObject);
  private
    FPostAction: TPostActionEvent;
  public
    { Открытые объявления }
  end;


procedure NewActionForm(APostAction: TPostActionEvent);

implementation
{ $R *.DFM}

procedure NewActionForm(APostAction: TPostActionEvent);
var
  ActionForm: TActionForm;
begin
  ActionForm := TActionForm.Create(Application);
  try
    ActionForm.FPostAction := APostAction;
    ActionForm.ShowModal;
  finally
    ActionForm.Free;
  end;
end;

procedure TActionForm.btnOKClick(Sender: TObject);
begin
  if Assigned(FPostAction) then
    FPostAction(Self, memAction.Lines);
end;

end.
```

Подобно методу `NewUserForm()`, методу `NewActionForm()` в качестве параметра также передается указатель на метод. Но на этот раз определен собственный тип метода `TPostActionEvent`, которому передаются объект `TObject` и объект `TStrings`, содержащий

текст примечания. По щелчку на кнопке OK вызывается обработчик события btnOKClick(), который, в свою очередь, вызывает метод TDDGBugsDataSource.PostAction(), чтобы добавить примечание в базу данных (указатель FPostAction ссылается на метод PostAction()).

Для получения дополнительной информации о модуле данных обращайтесь к комментариям, содержащимся в его исходном коде. Ниже будет показано, как в этот модуль данных добавить код, для того чтобы его можно было использовать совместно с другим приложением — сервером ISAPI, который сделает программу регистрации ошибок доступной для использования в Web.

Разработка интерфейса пользователя

В этом разделе рассматривается разработка пользовательского интерфейса создаваемого приложения. Кроме того, мы обсудим, что нужно предусмотреть, чтобы это приложение могло успешно работать в Web.

Главная форма

Пользовательский интерфейс в основном обращается к методам модуля данных. В этом случае вполне подойдет интерфейс, состоящий из одной формы с тремя вкладками. В первой вкладке можно добавлять, редактировать и просматривать информацию о зарегистрированных ошибках. Вторая вкладка предназначена для просмотра примечаний. Третья вкладка позволяет просматривать либо все содержимое базы данных, либо только те ошибки, которые введены данным (зарегистрированным) пользователем. На рис. 35.2–35.4 показаны все три вкладки главной формы.

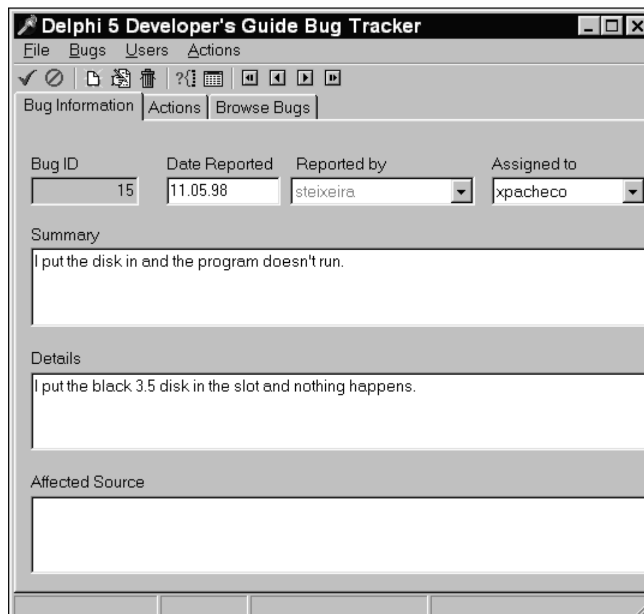


Рис. 35.2. Вкладка Bug Information *(Сведения об ошибках)*

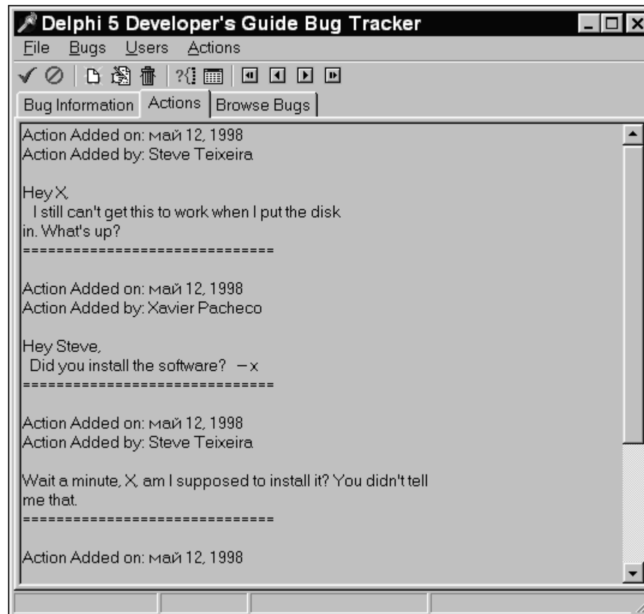


Рис. 35.3. Вкладка Actions *(Примечания)*

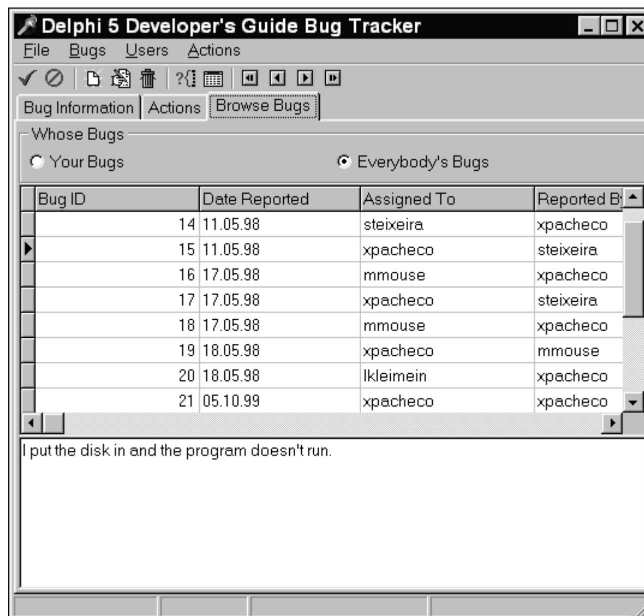


Рис. 35.4. Вкладка Browse Bugs *(Просмотр сведений об ошибках)*

Главная форма приложения `TMainForm` определена в модуле `MainFrm.pas`, исходный код которого содержится в листинге 35.5.

Листинг 35.5. Главная форма приложения регистрации ошибок

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, DBNAVSTATFRM, Menus, ImgList, ComCtrls, ToolWin, StdCtrls,
  DBCtrls, Db, Mask, dbModeFrm, ActnList, Grids, DBGrids, ExtCtrls;

type

  TMainForm = class(TDBNavStatForm)
    pcMain: TPageControl;
    tsBugInformation: TTabSheet;
    tsActions: TTabSheet;
    lblBugID: TLabel;
    dbeBugID: TDBEdit;
    dsBugs: TDataSource;
    lblDateReported: TLabel;
    lblSummary: TLabel;
    lblDetails: TLabel;
    lblAffectedSource: TLabel;
    lblReportedBy: TLabel;
    lblAssignedTo: TLabel;
    lblStatus: TLabel;
    dbmSummary: TDBMemo;
    dbmDetails: TDBMemo;
    dbmAffectedSource: TDBMemo;
    tsBrowseBugs: TTabSheet;
    rgWhoseBugs: TRadioGroup;
    dbgBugs: TDBGrid;
    dbmSummary2: TDBMemo;
    memAction: TMemo;
    dblcAssignedTo: TDBLookupComboBox;
    dblcStatus: TDBLookupComboBox;
    dbeDateReported: TDBEdit;
    mmiFile: TMenuItem;
    mmiExit: TMenuItem;
    mmiUsers: TMenuItem;
    mmiAddUser: TMenuItem;
    mmiActions: TMenuItem;
    mmiAddActionToBug: TMenuItem;
    dblcReportedBy: TDBLookupComboBox;
    procedure FormCreate(Sender: TObject);
    procedure sbFirstClick(Sender: TObject);
    procedure sbPreviousClick(Sender: TObject);
    procedure sbNextClick(Sender: TObject);
    procedure sbLastClick(Sender: TObject);
    procedure sbSearchClick(Sender: TObject);
```

```
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure sbAcceptClick(Sender: TObject);
    procedure sbCancelClick(Sender: TObject);
    procedure sbInsertClick(Sender: TObject);
    procedure sbEditClick(Sender: TObject);
    procedure sbDeleteClick(Sender: TObject);
    procedure sbBrowseClick(Sender: TObject);
    procedure rgWhoseBugsClick(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
    procedure mmiAddUserClick(Sender: TObject);
    procedure mmiAddActionToBugClick(Sender: TObject);
    procedure dsBugsDataChange(Sender: TObject; Field: TField);
  private
    procedure SetActionStatus;

  protected
  public
    { Объявления открытых данных }
  end;

var
  MainForm: TMainForm;

implementation

uses DDGBugsDM;

{ $R *.DFM}

{  TMainForm }

procedure TMainForm.SetActionStatus;
begin
  mmiFirst.Enabled := not DDGBugsDataModule.IsFirstBug;
  mmiLast.Enabled  := not DDGBugsDataModule.IsLastBug;
  mmiNext.Enabled  := not DDGBugsDataModule.IsLastBug;
  mmiPrevious.Enabled := not DDGBugsDataModule.IsFirstBug;
  mmiDelete.Enabled := not DDGBugsDataModule.IsBugsTblEmpty;

  sbFirst.Enabled := mmiFirst.Enabled;
  sbLast.Enabled  := mmiLast.Enabled;
  sbNext.Enabled  := mmiNext.Enabled;
  sbPrev.Enabled  := mmiPrevious.Enabled;
  sbDelete.Enabled := mmiDelete.Enabled;

  { В режиме добавления или редактирования ошибки нельзя
    добавлять других пользователей или примечания. }
  mmiUsers.Enabled := FormMode = fmBrowse;
  mmiActions.Enabled := (FormMode = fmBrowse) and
    (DDGBugsDataModule.NumBugs <> 0);
```

```
  { При редактировании или добавлении новой ошибки просмотр
    записей об ошибках запрещается. }
  dbgBugs.Enabled     := FormMode = fmBrowse;
  rgWhoseBugs.Enabled := FormMOde = fmBrowse;

end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  inherited;
  SetActionStatus;
end;

procedure TMainForm.sbFirstClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.FirstBug;
  SetActionStatus;
end;

procedure TMainForm.sbPreviousClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.PreviousBug;
  SetActionStatus;
end;

procedure TMainForm.sbNextClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.NextBug;
  SetActionStatus;
end;

procedure TMainForm.sbLastClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.LastBug;
  SetActionStatus;
end;

procedure TMainForm.sbSearchClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.SearchForBug;
end;

procedure TMainForm.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
var
  Rslt: word;
begin
```

```
  inherited;
  if not (FormMode = fmBrowse) then
  begin
    rslt := MessageDlg('Save changes?', mtConfirmation, mbYesNoCancel, 0);
    { Сохранить изменения? }
    case rslt of
      mrYes:
        begin
          DDGBugsDataModule.SaveBug;
          FormMode := fmBrowse;
          CanClose := True;
        end;
      mrNo:
        begin
          DDGBugsDataModule.CancelBug;
          FormMode := fmBrowse;
          CanClose := True;
        end;
      mrCancel:
        CanClose := False;
    end;
  end;
end;

procedure TMainForm.sbAcceptClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.SaveBug;
  SetActionStatus;
end;

procedure TMainForm.sbCancelClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.CancelBug;
  SetActionStatus;
end;

procedure TMainForm.sbInsertClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.InsertBug;
  SetActionStatus;
end;

procedure TMainForm.sbEditClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.EditBug;
  SetActionStatus;
end;
```

```
procedure TMainForm.sbDeleteClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.DeleteBug;
  SetActionStatus;
end;

procedure TMainForm.sbBrowseClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.CancelBug;
  SetActionStatus;
end;

procedure TMainForm.rgWhoseBugsClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.FilterOnUser := rgWhoseBugs.ItemIndex = 0;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  inherited;
  Close;
end;

procedure TMainForm.mmiAddUserClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.AddUser;
end;

procedure TMainForm.mmiAddActionToBugClick(Sender: TObject);
begin
  inherited;
  DDGBugsDataModule.AddAction;
  dsBugsDataChange(nil, nil);
end;

procedure TMainForm.dsBugsDataChange(Sender: TObject; Field: TField);
begin
  inherited;
  { Отображается новая ошибка, поэтому список примечаний очищается
    и считываются примечания для заново отображаемой ошибки. }
  memAction.Lines.Clear;
  DDGBugsDataModule.GetActions(memAction.Lines);
end;

end.
```

Класс формы TMainForm является производным от класса TDBNavStatForm, создание которого описано в главе 4, “Строение приложения и концепции конструирования”. Он содержится в хранилище объектов (Object Repository). Класс TDBNavStatForm включает функции, предназначенные для обновления кнопок панели инструментов и строки состояния в зависимости от режима формы (добавление, редактирование, просмотр). Задача большинства методов состоит лишь в вызове соответствующих методов модуля данных.

Обратите внимание на то, что в свойстве dsBugs.AutoEdit содержится значение False, благодаря чему предотвращается случайное переключение таблицы Bugs в режим редактирования. Перевод таблицы Bugs в режим редактирования или вставки должен происходить явным образом после щелчка на соответствующей кнопке панели инструментов или при выборе соответствующей команды меню.

Эта форма довольна проста. В зависимости от различных условий процедура SetActionStatus() просто разрешает или запрещает использование кнопок и команд меню. Метод FormCloseQuery() гарантирует, что любая незавершенная операция редактирования или вставки будет сохранена или отменена.

Другие функции пользовательского интерфейса

Управление отображением меток полей осуществлялось за счет добавления полей в объект TTable модуля данных и задания более понятных названий в окне инспектора объектов. То же самое можно сделать и для объекта TDBGrid, если модифицировать свойство Title свойства TDBGrid.Columns. Для управления отображением меток использовались оба метода.

Подготовка приложения к работе в Web

Ранее уже упоминалось, что для разрабатываемого приложения потребуется Web-версия. Для этого из класса TDDGBugsDataModule нужно удалить ссылки на любые формы. Это можно реализовать с помощью директив условной компиляции, как это сделано в модуле DDGBugsDM.pas. Использование директив условной компиляции рассмотрим на примере следующего кода:

```
{$IFNDEF DDGWEBBUGS}
  procedure AddUser;
{$ENDIF}
```

Директива {$IFNDEF} гарантирует, что метод AddUser() компилируется только в том случае, если условие DDGWEBBUGS не определено.

Резюме

В этой главе рассматривались методы разработки настольных приложений для работы с базами данных. Особое внимание было уделено вопросам отделения пользовательского интерфейса от процедур обработки данных. Благодаря этому существенно облегчается процесс создания Web-версии приложения. Как это осуществить, вы узнаете в следующей главе, “Приложение с использованием компонентов WebBroker: сбор сведений об ошибках”.

Глава 36

Приложение с использованием компонентов WebBroker: сбор сведений об ошибках

Макеты страниц приложения	**878**
Внесение изменений в модуль данных	**879**
Настройка компонента TDataSetTableProducer**:** **объект** dstpBugs	**879**
Настройка компонента TWebDispatcher**:** **объект** wbdpBugs	**880**
Настройка компонента TPageProducer**:** **объект** pprdBugs	**880**
Кодирование функций ISAPI-сервера: добавление экземпляров объекта TActionItem	**881**
Просмотр сведений об ошибках	**887**
Добавление сведений о новой ошибке	**893**
Резюме	**899**

В предыдущей главе, “Разработка настольного приложения: сбор сведений об ошибках”, были продемонстрированы различные аспекты проектирования настольных приложений баз данных. При этом в качестве одного из важнейших условий выдвигалось требование спроектировать эти приложения так, чтобы максимально упростить их перенос в среду World Wide Web. В данной главе реальное приложение, процесс создания которого был описан в предыдущей главе, будет преобразовано нами для работы в Web в качестве ISAPI-сервера. Как и предполагалось, в уже написанный код потребуется внести минимальные изменения. При этом будут использоваться приемы, описанные в главе 31, “Компоненты WebBroker открывают двери в Internet”, поэтому здесь мы не станем вдаваться в подробности, о которых уже шла речь. Если “в присутствии” компонентов WebBroker вы чувствуете себя не очень уверенно и ощущаете необходимость освежить в памяти основные темы, рассмотренные в главе 31, обязательно сделайте это, прежде чем продолжить чтение.

Макеты страниц приложения

Общая структура создаваемого Web-ориентированного приложения для регистрации ошибок показана на рис. 36.1.

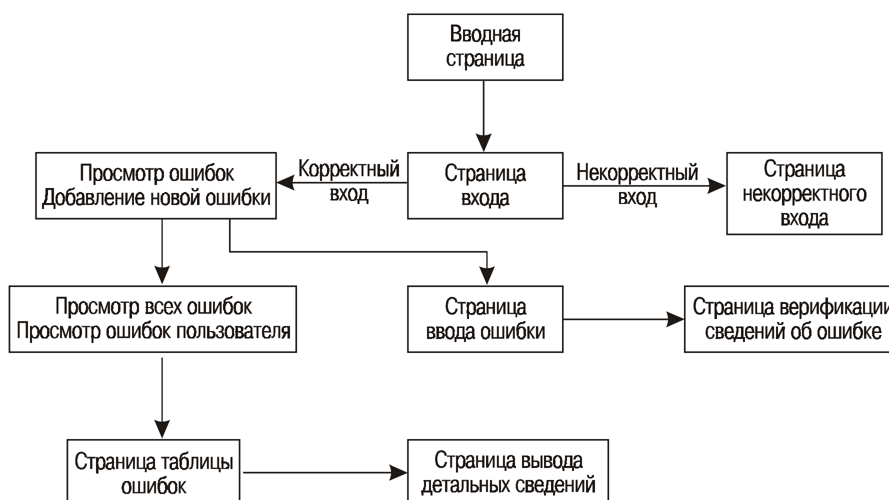


Рис. 36.1. Схема построения Web-ориентированного приложения для регистрации ошибок

Как видно из рис. 36.1, данное приложение на самом деле реализует лишь ограниченное подмножество функций, поддерживаемых приложением, описанным в главе 35. Добавление к нему всей функциональности исходного приложения мы оставляем читателям (в качестве практического упражнения).

В последующих разделах разъясняются особенности программного текста, созданного при реализации всех указанных на схеме Web-страниц. Обратите внимание на то, что все страницы создаются динамически во время выполнения, т.е. без загрузки заранее созданных HTML-документов. Это не значит, что существуют какие-то веские причины, которые вынудили нас выбрать именно этот вариант вместо написания статических HTML-документов, загружаемых с помощью компонентов WebBroker. При создании собственных приложений вы можете поступить по-другому.

Внесение изменений в модуль данных

Основная цель — это применить как можно больше функций и компонентов, которые использовались при проектировании класса TDDGBugsDataModule в предыдущей главе. Постараемся добавить в этот модуль данных некоторые функции и минимизировать любые изменения, которые могли бы потенциально помешать использованию этого приложения в его исходной версии, не предназначенной для Web. Решение данной задачи состоит в исключении внесения каких-либо изменений в уже существующие методы. Кроме того, чтобы удостовериться, что предыдущее приложение осталось целым и невредимым, исходное приложение должно быть перекомпилировано и протестировано.

Обратите внимание: создавать отдельный Web-модуль не потребовалось — просто к существующему классу TDataModule был добавлен компонент TWebDispatcher. Благодаря этому стало возможным воспользоваться ранее созданным классом TDataModule.

Для создания Web-ориентированной версии приложения регистрации ошибок к классу TDDGBugsDataModule было добавлено еще четыре компонента: TWebDispatcher, TDataSetTableProducer, TPageProducer и TSession.

Следует подробнее пояснить назначение компонента TSession. Потенциально к библиотеке DLL ISAPI-сервера могут получать доступ несколько клиентов, т.е. с помощью единственного экземпляра динамической библиотеки одновременно несколько человек могут попытаться обратиться к базе данных. Такая библиотека будет функционировать в пространстве одного процесса, и, следовательно, для каждого клиента, который предпринимает попытку обращения к серверу, потребуется отдельный, специально выделенный ему Web-модуль. Эти отдельные Web-модули создаются во время выполнения и функционируют в собственном уникальном потоке. А это, в свою очередь, требует, чтобы каждое подключение к базе данных осуществлялось с помощью собственного компонента TSession, что предотвратит появление конфликтов между несколькими клиентами, одновременно получающими доступ к базе данных. Задав свойству TSession.AutoSessionName компонента TSession значение True, мы гарантируем, что каждому экземпляру компонента TSession будет присвоено собственное уникальное имя. Поэтому каждый создаваемый поток потребует организации собственного сеанса работы с BDE.

Обратите внимание на то, что при написании серверных приложений WinCGI или CGI добавлять компонент TSession к Web-модулю или в компонент TDataModule не требуется, поскольку они компилируются в отдельные приложения, функционирующие в своем собственном пространстве процесса.

Настройка компонента TDataSetTableProducer: объект dstpBugs

Компонент TDataSetTableProducer с именем dstpBugs подключается к компоненту TTable с именем tblBugs. Как и при настройке компонента TDBGrid, для задания собственных заголовков необходимо модифицировать свойство dstpBugs.Columns — как показано на рис. 36.2. Эти заголовки будут отображаться в таблице Web-страницы. Для создания рамки шириной в один пиксель следует модифицировать свойство dstpBugs.TableAttributes. Это придаст таблице визуальную объемность, отображение которой поддерживает большинство Web-броузеров.

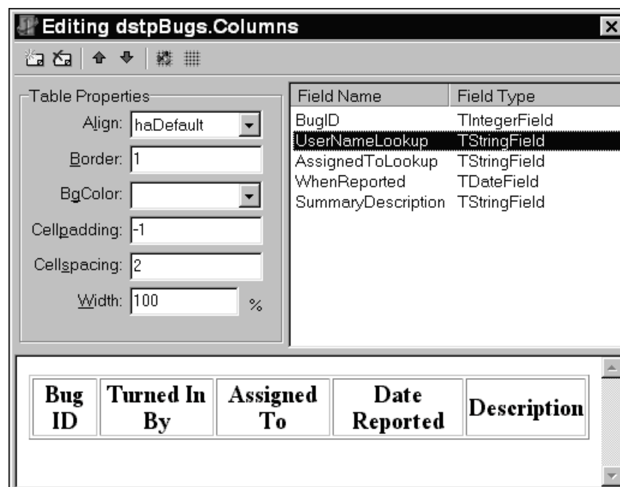


Рис. 36.2. Редактирование свойства `Columns` объекта `dstpBugs`

Настройка компонента `TWebDispatcher`: объект `wbdpBugs`

На рис. 36.3 показан редактор свойства `Actions`, используемый для добавления в объект `wbdpBugs` нескольких экземпляров объектов `TWebActionItem`. Каждое из определяемых этими объектами действий, а также способы предоставления ими пользователю доступа к приложению регистрации ошибок через Web будут подробно рассмотрены ниже.

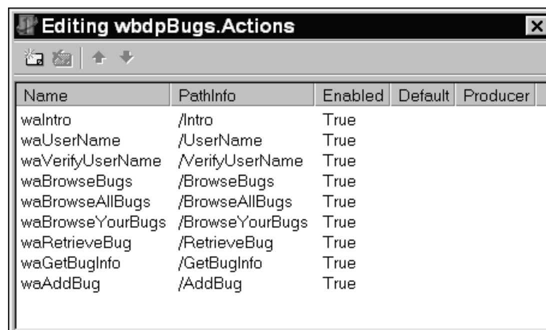


Рис. 36.3. Редактирование свойства `Actions` для экземпляра `dstpBugs` компонента `TDataSetTableProducer`

Настройка компонента `TPageProducer`: объект `pprdBugs`

Если проверить значение свойства `pprdBugs.HTMLDoc`, то окажется, что оно пусто. Дело в том, что все манипуляции значением этого свойства выполняются во время работы приложения программно. Как мы увидим чуть позже, один и тот же экземпляр компонента `TPageProducer` будет использоваться в двух различных ситуациях.

Кодирование функций ISAPI-сервера: добавление экземпляров объекта TActionItem

Функционирование Web-ориентированного приложения регистрации ошибок обеспечивается благодаря экземплярам объекта TActionItem, создаваемым внутри компонента TWebDispatcher. Назначение каждого создаваемого экземпляра объекта TActionItem описано в табл. 36.1. Все они будут рассмотрены отдельно.

Таблица 36.1. Назначение экземпляров компонента TActionItem

TActionItem	Назначение
waIntro	Отображает на экране начальную страницу приложения
waUserName	Предлагает пользователю ввести свое имя
waVerifyUserName	Активизируется в обработчике события waUserName.OnAction. Проверяет имя, введенное пользователем
waBrowseBugs	Предоставляет пользователю два варианта просмотра: просмотр всех ошибок или просмотр ошибок, введенных данным пользователем
waBrowseAllBugs	Отображает таблицу, содержащую все ошибки в базе данных
waBrowseYourBugs	Отображает таблицу, содержащую ошибки, "принадлежащие" данному пользователю
waRetrieveBug	Отображает подробную информацию об ошибках
waGetBugInfo	Предоставляет страницу для ввода новой информации об ошибках
waAddBug	Добавляет в таблицу информацию о новой ошибке и отображает экран верификации

Вместо полного листинга модуля DDBBugsDM.pas в последующих разделах приводятся отдельные листинги каждого из методов, добавленных в этот модуль.

Вспомогательные процедуры

Процедура AddHeader(), текст которой приведен в листинге 36.1, используется для добавления на Web-страницы приложения регистрации ошибок стандартного верхнего колонтитула, состоящего из названия страницы и самого колонтитула. Кроме того, здесь также задается и фоновое изображение. Обратите внимание на то, что расположение этого фонового изображения зависит от Web-сервера. По всей вероятности, для того чтобы файл изображения был найден, вам придется модифицировать этот код в соответ-

ствии с вашей системой. Метод `AddFooter()` (его текст показан в листинге 36.2) используется для добавления стандартного нижнего колонтитула, включающего информацию об авторских правах.

Листинг 36.1. Метод `TDDGBugsDataModule.AddHeader()` используется для создания стандартного верхнего колонтитула

```
procedure AddHeader(AWebPage: TStringList);
// Добавление стандартного верхнего колонтитула на каждую Web-страницу
begin
  with AWebPage do
  begin
    Add('<HTML>');
    Add('<HEAD>');
    Add('<BODY BACKGROUND=''/samples/images/backgrnd.gif''">');
    Add('<TITLE>Delphi 5 Developer''s Guide Bug Demo</Title>');
    Add('<CENTER>');
    Add('<P>');
    Add('<FONT SIZE=6>Delphi 5 Developer''s Guide Bug Demo</font>');
    Add('</CENTER>');
    Add('</HEAD>');
  end;
end;
```

Листинг 36.2. Метод `TDDGBugsDataModule.AddFooter()` используется для добавления стандартного нижнего колонтитула

```
procedure AddFooter(AWebPage: TStringList);
// Добавление на каждую Web-страницу стандартного нижнего колонтитула
begin
  with AWebPage do
  begin
    Add('<BR><BR>Copyright (c) 1998, Delphi 5 Developer''s Guide.');
    Add('</BODY>');
    Add('</HTML>');
  end;
end;
```

Вводная страница

Вводная страница показана на рис. 36.4. Она создается обработчиком события `waIntro.OnAction` в методе `wbdpBugswaIntroAction()`, код которого приведен в листинге 36.3.

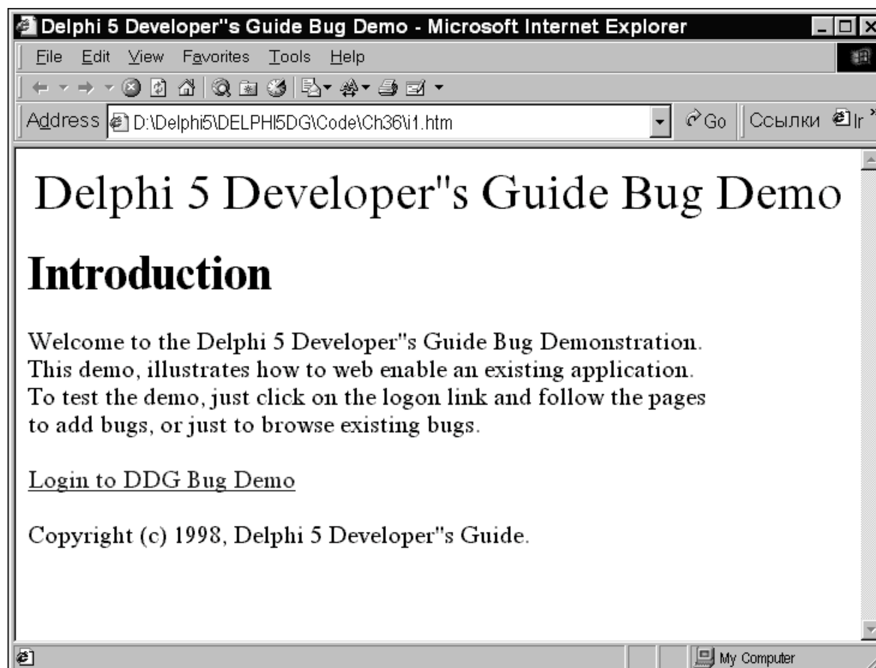


Рис. 36.4. Начальная страница приложения

Листинг 36.3. Метод `TDDGBugsDataModule.wbdpBugswaIntroAction()` отображает начальную страницу

```
procedure TDDGBugsDataModule.wbdpBugswaIntroAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
// Начальная страница демонстрационной версии Web-приложения
var
  WebPage: TStringList;
begin
  WebPage := TStringList.Create;
  try
    AddHeader(WebPage);
    with WebPage do
    begin
      Add('<BODY>');
      Add('<H1>Introduction</H1>');
      Add('<P>Welcome to the Delphi 5 Developer''s Guide Bug Demonstration.');
      Add('<BR>This demo, illustrates how to web enable an existing
      ➥ application.');
      Add('<BR>To test the demo, just click on the logon link and follow
      ➥ the pages');
       Add('<BR>to add bugs, or just to browse existing bugs.');
      Add('</P>');
      Add('<A href="../DDGWebBugs.dll/UserName">Login to DDG Bug Demo</A>');
```

```
      AddFooter(WebPage);
      Response.Content :=  WebPage.Text;
      Handled := True;
    end;
  finally
    WebPage.Free;
  end;
end;
```

Обратите внимание на то, как в каждом экземпляре, в котором генерируется Web-страница, процедурам `AddHeader()` и `AddFooter()` передается объект `WebPage`. Вводная страница довольно проста. Помимо текста, на ней содержится лишь одна гиперссылка на экземпляр `waUserName` компонента `TWebAction`. Описание работы компонента `TWebAction` можно найти в главе 31, “Компоненты WebBroker открывают двери в Internet”.

Получение и проверка регистрационного имени пользователя

На рис. 36.5 показана страница, сгенерированная в методе `TDDGBugsDataModule.wbdpBugswaUserNameAction()`; текст этого метода представлен в листинге 36.4. По сути, эта страница представляет собой HTML-форму, предназначенную для получения имени пользователя, а результатом ее работы является вызов обработчика события `TDDGBugsDataModule.wbdpBugswaVerifyUserNameAction()`. Текст этого метода показан в листинге 36.5.

Рис. 36.5. Получение имени пользователя

Листинг 36.4. Метод `TDDGBugsDataModule.wbdpBugswaUserNameAction()` — отображает страницу, на которой пользователь может ввести свое имя

```
procedure TDDGBugsDataModule.wbdpBugswaUserNameAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
// На этой странице пользователю предлагается ввести свое имя
var
  WebPage: TStringList;
begin
  WebPage := TStringList.Create;
  try
    AddHeader(WebPage);
    with WebPage do
    begin
      Add('<BODY>');
      Add('<H1>Enter your user name</H1>'); // Введите свое имя
      Add('<FORM action="../DDGWebBugs.dll/VerifyUserName" method="GET">');
      Add('<p>UserName: <INPUT type="text" name="UserName" maxlength="30"
      size="50"></P>');
      Add('<p><INPUT type="SUBMIT"><INPUT type="RESET"></p>');
      Add('</FORM>');
      AddFooter(WebPage);
      Response.Content :=  WebPage.Text;
      Handled := True;
    end;
  finally
    WebPage.Free;
  end;
end;
```

Листинг 36.5. В методе `TDDGBugsDataModule.wbdpBugswaVerifyUserNameAction()` выполняется проверка имени пользователя

```
procedure TDDGBugsDataModule.wbdpBugswaVerifyUserNameAction(Sender: TObject;
Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
{ На данной странице считывается имя, введенное пользователем.
  Эта информация сохраняется и передается назад клиенту в виде cookie.
  Кроме того, в тексте cookie передается также и дополнительная информация,
  которая позже будет использоваться для добавления данных об ошибках в Web. }

var
  WebPage: TStringList;
  CookieList: TStringList;
  UserName: String;
  UserFName,
  UserLName: String;
  UserID: Integer;
  ValidLogin: Boolean;

procedure BuildValidLoginPage;
```

```
begin
  AddHeader(WebPage);
  with WebPage do
  begin
    Add('<BODY>');
    Add(Format('<H1>User name, %s verified. User ID is: %d</H1>',
      [Request.QueryFields.Values['UserName'], UserID]));
    Add('<BR><BR><A href="../DDGWebBugs.dll/BrowseBugs">Browse Bug List</A>');
    Add('<BR><A href="../DDGWebBugs.dll/GetBugInfo">Add a New Bug</A>');
    AddFooter(WebPage);
  end;
end;

procedure BuildInValidLoginPage;
begin
  AddHeader(WebPage);
  with WebPage do
  begin
    Add('<BODY>');
    Add(Format('<H1>User name, %s is not a valid user.</H1>',
      [Request.QueryFields.Values['UserName']]));
    AddFooter(WebPage);
  end;
end;

begin

  UserName := Request.QueryFields.Values['UserName'];

  { Регистрация пользователя успешна, если его имя имеется
    в таблице Users.db. }
  ValidLogin := tblUsers.Locate('UserName', UserName, []);

  WebPage := TStringList.Create;
  try

    if ValidLogin then
    begin

      { Получение идентификатора пользователя (UserID), его имени и фамилии }
      UserID := tblUsers.FieldByName('UserID').AsInteger;
      UserFName := tblUsers.FieldByName('UserFirstName').AsString;
      UserLName := tblUsers.FieldByName('UserLastName').AsString;

      CookieList := TSTringList.Create;
      try

        // Сохранение информации о пользователе в виде cookies
        CookieList.Add('UserID='+IntToStr(UserID));
        CookieList.Add('UserName='+UserName);
```

```
        CookieList.Add('UserFirstName='+UserFName);
        CookieList.Add('UserLastName='+UserLName);

        Response.SetCookieField(CookieList, '', '', Now + 1, False);
      finally
        CookieList.Free;
      end;
      BuildValidLoginPage;
    end
    else begin
      UserID := -1;
      BuildInvalidLoginPage;
    end;

    Response.Content :=  WebPage.Text;
    Handled := True;

  finally
    WebPage.Free;
  end;

end;
```

В методе wbdpBugswaVerifyUserNameAction() выполняется несколько действий. Во-первых, проверяется наличие введенного имени пользователя в таблице tblUsers. Если указанное имя в таблице имеется, вызывается процедура BuildValidLoginPage(), в противном случае — метод BuildInvalidLoginPage().

Если имя пользователя оказалось допустимым, то из таблицы tblUsers считывается идентификатор (ID) пользователя, его имя и фамилия. Затем эти элементы в виде файла cookies возвращаются клиенту. В дальнейшем при формировании запросов эти значения будут передаваться Web-серверу, который использует их при генерации других страниц приложения. Наконец, вызывается метод BuildValidLoginPage(). В нем создается страница, содержащая гиперссылки для просмотра ошибок и добавления сведений о новых ошибках. Если регистрация пользователя прошла неудачно, вызывается метод BuildInvalidLoginPage(), в котором просто пользователю сообщается о том, что введенное им имя является некорректным.

После успешного прохождения этапа регистрации пользователю будет предложено выбрать один из двух возможных режимов работы: просмотр данных о существующих ошибках либо ввод информации о новой ошибке.

Просмотр сведений об ошибках

При выборе режима просмотра ошибок будет сгенерирована страница, на которой пользователю предлагается выбрать просмотр сведений либо обо всех ошибках, занесенных в базу данных, либо сведений об ошибках, введенных данным пользователем. Эта страница создается в методе TDDGBugsDataModule.wbdpBugswaBrowseBugsAction(), исходный код которого приведен в листинге 36.6.

Листинг 36.6. В методе `TDDGBugsDataModule.wbdpBugswaBrowseBugsAction()` создается страница, на которой можно выбрать один из двух возможных режимов просмотра

```
procedure TDDGBugsDataModule.wbdpBugswaBrowseBugsAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
{ На этой странице пользователь может выбрать просмотр либо
  всех ошибок, либо ошибок, введенных им самим. }
var
  WebPage: TStringList;
begin
  WebPage := TStringList.Create;
  try
    AddHeader(WebPage);
    with WebPage do
    begin
      Add('<BODY>');
      Add('<H1>Browse Option</H1>');
      Add('<BR><BR><A href="../DDGWebBugs.dll/BrowseAllBugs">
      ↳Browse All Bugs</A>');
      Add('<BR><A href="../DDGWebBugs.dll/BrowseYourBugs">
      ↳Browse Your Bugs</A>');
      AddFooter(WebPage);
      Response.Content :=  WebPage.Text;
      Handled := True;
    end;
  finally
    WebPage.Free;
  end;

end;
```

Просмотр всех ошибок

При выборе режима просмотра всех ошибок вызывается обработчик события `TDDGBugsDataModule.wbdpBugswaBrowseAllBugsAction()`, исходный код которого содержится в листинге 36.7.

Листинг 36.7. В методе `TDDGBugsDataModule.wbdpBugswaBrowseBugsAction()` отображаются все ошибки, зарегистрированные в приложении

```
procedure TDDGBugsDataModule.wbdpBugswaBrowseAllBugsAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
{ На этой странице для просмотра всех ошибок используется компонент
  TPageProducer. На нее помещаются стандартные верхний и нижний колонтитулы,
  а для добавления таблицы используется специализированный дескриптор. }
var
  WebPage: TStringList;
begin
```

```
  WebPage := TStringList.Create;
  try
    AddHeader(WebPage);
    WebPage.Add('<BODY>');
    WebPage.Add('<H1>Browsing all Bugs</H1>');
    WebPage.Add('<#TABLE>');
    AddFooter(WebPage);

    pprdBugs.HTMLDoc.Clear;
    pprdBugs.HTMLDoc.AddStrings(WebPage);

    { В результате выполнения содержащейся в следующей строке инструкции
      для объекта pprdBugs будет вызван обработчик события OnHTMLTag. }
    Response.Content := pprdBugs.Content;

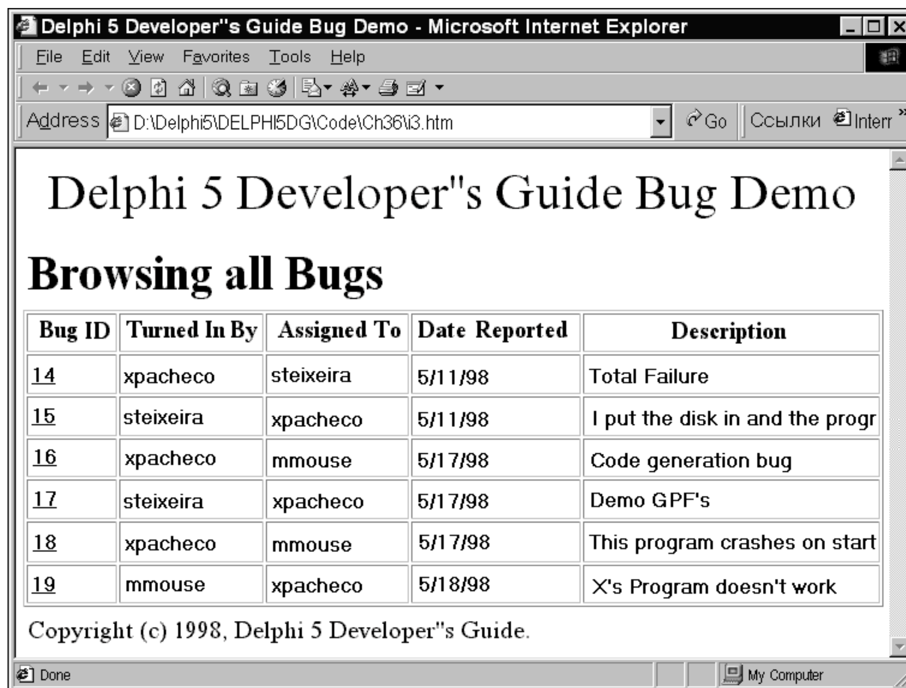
    Handled := True;
  finally
    WebPage.Free;
  end;
end;
```

В этом обработчике события используется экземпляр pprdBugs компонента TPageProducer. Для нас важна его способность обрабатывать специализированные дескрипторы внутри HTML-кода страницы. В данном случае используется дескриптор #TABLE. На эту Web-страницу также были добавлены стандартные верхний и нижний колонтитулы. Однако вместо присвоения объекта WebPage свойству Response.Content это объект был присвоен свойству pprdBugs.HTMLDoc, и лишь затем значение свойства pprdBugs.Content было присвоено свойству Response.Content. Целью всех этих манипуляций является организация генерации события pprdBugs.OnHTMLTag. Обработка этого события производится в методе TDDGBugsDataModule.pprdBugsHTMLTag(), исходный текст которого представлен в листинге 36.8.

Листинг 36.8. Метод `TDDGBugsDataModule.pprdBugsHTMLTag()`

```
procedure TDDGBugsDataModule.pprdBugsHTMLTag(Sender: TObject; Tag: TTag;
 const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if Tag = tgTable then begin
    with dstpBugs do
    begin
      DataSet.Close;
      DataSet.Open;
      ReplaceText := dstpBugs.Content;
    end;
  end;
end;
```

Этот простой обработчик события присваивает свойство dstpBugs.Content, ссылающееся на таблицу, свойству pprdBugs.ReplaceText. При этом дескриптор #TABLE будет заменен содержимым таблицы. Сгенерированная в результате описанных действий страница показана на рис. 36.6. На ней отображается информация об ошибках, внесенных всеми пользователями.



Bug ID	Turned In By	Assigned To	Date Reported	Description
14	xpacheco	steixeira	5/11/98	Total Failure
15	steixeira	xpacheco	5/11/98	I put the disk in and the progr
16	xpacheco	mmouse	5/17/98	Code generation bug
17	steixeira	xpacheco	5/17/98	Demo GPF's
18	xpacheco	mmouse	5/17/98	This program crashes on start
19	mmouse	xpacheco	5/18/98	X's Program doesn't work

Рис. 36.6. Список ошибок, внесенных всеми пользователями

Просмотр данных об ошибках, внесенных конкретным пользователем

Если пользователь предпочел просмотреть только свои ошибки, создается страница, содержащая таблицу с ошибками, внесенными им самим. Эта страница создается обработчиком события `TDDGBugsDataModule.wbdpBugswaBrowseYourBugsAction()` (листинг 36.9).

Листинг 36.9. Метод `TDDGBugsDataModule.wbdpBugswaBrowseYourBugsAction()` — отображает сведения об ошибках только данного пользователя

```
procedure TDDGBugsDataModule.wbdpBugswaBrowseYourBugsAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
{ На этой странице для просмотра ошибок, "принадлежащих" данному пользователю,
  подготавливается компонент TPageProducer. На нее помещаются стандартные
  верхний и нижний колонтитулы, однако для добавления таблицы используется
  специализированный дескриптор. }
var
  WebPage: TStringList;
  UserID: Integer;
  UserFName,
  UserLName: String;
begin
```

```
  WebPage := TStringList.Create;
  try
    AddHeader(WebPage);
    WebPage.Add('<BODY>');

    // Считывание идентификатора пользователя, сохраняемого в файле cookie
    UserID    := StrToInt(Request.CookieFields.Values['UserID']);
    UserFName := Request.CookieFields.Values['UserFirstName'];
    UserLName := Request.CookieFields.Values['UserLastName'];

    WebPage.Add(Format('<H1>Browsing Bugs Entered by %s %s</H1>',
      [UserFName, UserLName]));
    WebPage.Add('<#TABLE>');
    pprdBugs.HTMLDoc.Clear;
    pprdBugs.HTMLDoc.AddStrings(WebPage);

    AddFooter(WebPage);

    // Фильтрация таблицы по значению UserID
    FLoginUserID := UserID;
    FilterOnUser := True;

    Response.Content := pprdBugs.Content;

    Handled := True;
  finally
    WebPage.Free;
  end;

end;
```

Как и при просмотре всех ошибок, на данную страницу добавляются стандартные верхний и нижний колонтитулы. Кроме того, из свойства Request.CookieFields считываются значения полей UserID, UserFirstName и UserLastName. Значения UserFirstName и UserLastName отображаются на Web-странице. Значение переменной UserID присваивается свойству FLoginUserID, а затем значение свойства FilterOnUser устанавливается равным True. Как упоминалось в предыдущей главе, при установке значения свойства FilterOnUser равным True вызывается его метод записи SetFilterOnUser(), который, в свою очередь, помещает значение True в свойство tblBugs.Filtered. Это приводит к вызову обработчика события OnFilterRecord экземпляра tblBugs — tblBugsFilterRecord() — для каждой записи в наборе данных. В этом обработчике содержится следующая строка кода:

```
Accept := tblBugs.FieldByName('UserID').AsInteger = FLoginUserID;
```

Обратите внимание на то, что применяемый фильтр зависит от значения, содержащегося в поле FLoginUserID. Это объясняет, почему значение переменной UserID поля cookie необходимо присвоить полю FLoginUserID.

Наконец, значение свойства pprdBugs.Content присваивается свойству Response.Content. И это снова приведет к возникновению события pprdBugs.OnHTMLTag.

Форматирование ячеек таблицы и отображение подробной информации об ошибках

В экземпляре объекта dstpBugs имеется метод TDDGBugsDataModule.dstpBugsFormatCell(), представляющий собой обработчик события OnFormatCell. Этот обработчик события преобразует отображаемое значение BugID в HTML-ссылку, которая предоставляет подробную информацию об этой ошибке. На самом деле отображение подробных сведений об ошибке обеспечивает обработчик события TDDGBugsDataModule.wbdpBugswaRetrieveBugAction(). Исходный текст обоих обработчиков событий содержится в листинге 36.10.

Листинг 36.10. Обработчики событий, предназначенные для отображения подробной информации об ошибках

```
procedure TDDGBugsDataModule.dstpBugsFormatCell(Sender: TObject; CellRow,
  CellColumn: Integer; var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
{ Преобразование ячейки BugID таблицы в гиперссылку, которая
  приводит к отображению подробных сведений об ошибке. }
begin
  if (CellColumn = 0) and not (CellRow = 0) then
    CellData := Format('<A href="../DDGWebBugs.dll/RetrieveBug?
↳BugID=%s">%s</A>',
      [CellData, CellData]);
end;

procedure TDDGBugsDataModule.wbdpBugswaRetrieveBugAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
{ Просмотр подробной информации об ошибке. }
var
  BugID: Integer;
  WebPage: TStringList;

procedure GetBug;
begin
  if tblBugs.Locate('BugID', BugID, []) then
    with tblBugs do
    begin
      WebPage.Add(Format('Bug ID:        %d', [BugID]));
      WebPage.Add(Format('<BR>Reported By:   %s',
        [FieldByName('UserNameLookup').AsString]));
      WebPage.Add(FormatDateTime('"<BR>Reported On:"   mmm dd, yyyy',
        FieldByName('WhenReported').AsDateTime));
      WebPage.Add(Format('<BR>Assigned To:   %s',
        [FieldByName('AssignedToLookup').AsString]));
      WebPage.Add(Format('<BR>Status:        %s',
        [FieldByName('StatusTitle').AsString]));
      WebPage.Add(Format('<BR>Summary:       %s',
```

```
        [FieldByName('SummaryDescription').AsString]));
      WebPage.Add(Format('<BR>Details:      %s',
        [FieldByName('Details').AsString]));
      WebPage.Add('<BR>');
      WebPage.Add('<BR>');

      GetActions(WebPage);
    end;
end;

begin
  BugID := StrToInt(Request.QueryFields.Values['BugID']);

  WebPage := TStringList.Create;
  try
    AddHeader(WebPage);
    with WebPage do
    begin
      Add('<BODY>');
      Add('<H1>Bug Detail</H1>');
      GetBug;
      AddFooter(WebPage);
      Response.Content :=  WebPage.Text;
      Handled := True;
    end;
  finally
    WebPage.Free;
  end;

end;
```

Добавление сведений о новой ошибке

В рассматриваемом приложении пользователь имеет возможность добавить в базу данных информацию о новой ошибке. В следующих разделах будут рассмотрены страницы, которые предназначены для ввода данных пользователем и отображения информации о новой ошибке после завершения ввода.

Ввод сведений о новой ошибке

В обработчике события `TDDGBugsDataModule.wbdpBugswaGetBugInfoAction()`, исходный код которого содержится в листинге 36.11, генерируется страница, предназначенная для получения от пользователя информации о новой ошибке. По существу, на этой странице создается HTML-форма, содержащая соответствующие элементы управления, позволяющие ввести нужные данные. Вид страницы, созданной в этом обработчике события, показан на рис. 36.7.

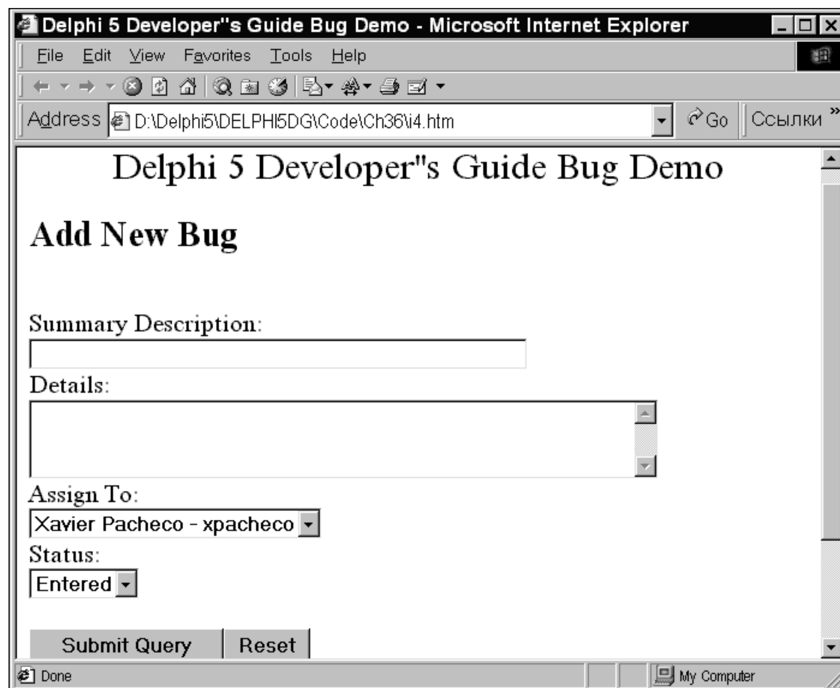


Рис. 36.7. Страница ввода данных об ошибке

Листинг 36.11. Метод `TDDGBugsDataModule.wbdpBugswaGetBugInfoAction()`, предназначенный для генерации страницы, используемой для ввода сведений о новой ошибке

```
procedure TDDGBugsDataModule.wbdpBugswaGetBugInfoAction(Sender: TObject; Request:
TWebRequest; Response: TWebResponse; var Handled: Boolean);
{ Подготовка страницы к приему от пользователя информации о новой ошибке. }
var
  WebPage: TStringList;

procedure AddAssignToNames;
{ Добавление на HTML-страницу раскрывающегося списка Assign To,
  связанного с именами пользователей. }
begin

  WebPage.Add('<BR>Assign To:');
  WebPage.Add('<BR><SELECT name="AssignTo"><BR>');

  with tblUsers do
  begin
    First;
    while not Eof do
    begin
```

```
      WebPage.Add(Format('<OPTION>%s %s - %s',
        [FieldByName('UserFirstName').AsString,
        FieldByName('UserLastName').AsString,
        FieldByName('UserName').AsString]));
      tblUsers.Next;
    end;
    WebPage.Add('</SELECT>');
  end;
end;

procedure AddStatusTitles;
{ Добавление на HTML-страницу раскрывающегося списка элементов
  состояния ошибок. }
begin
  WebPage.Add('<BR>Status:');
  WebPage.Add('<BR><SELECT name="Status"><BR>');

  with tblStatus do
  begin
    First;
    while not Eof do
    begin
      WebPage.Add(Format('<OPTION>%s', [FieldByName('StatusTitle').AsString]));
      tblStatus.Next;
    end;
    WebPage.Add('</SELECT>');
  end;
end;

begin
  WebPage := TStringList.Create;
  try
    AddHeader(WebPage);
    with WebPage do
    begin
      Add('<BODY>');
      Add('<H1>Add New Bug</H1>');
      Add('<FORM action="../DDGWebBugs.dll/AddBug"method="GET">');
      Add('<BR>Summary Description:<BR><INPUT type="text"name="Summary"
      maxlength="100" size="50">');
      Add('<BR>Details:<BR><TEXTAREA name="Details"rows=5 cols=50> </TEXTAREA>');

      AddAssignToNames;
      AddStatusTitles;

      Add('<p><INPUT type="SUBMIT"><INPUT type="RESET"></p>');
      Add('</FORM>');
      AddFooter(WebPage);
```

```
      Response.Content :=  WebPage.Text;
      Handled := True;
    end;
  finally
    WebPage.Free;
  end;
end;
```

С помощью двух вспомогательных функций — `AddAssignToNames()` и `AddStatusTitle()` — создаются комбинированные элементы управления списками, из которых при вводе сведений об ошибках пользователь может выбрать требуемые значения. В отличие от Delphi, где имеются компоненты, предназначенные для управления работой с данными и способные автоматически присваивать новой записи выбранные значения подстановки, в данном случае таких компонентов нет и поэтому присвоение должно быть выполнено вручную. Именно эти действия и реализованы в обработчике события, предназначенном для добавления в базу данных сведений о новой ошибке.

Верификация введенной информации об ошибке

В листинге 36.12 представлен исходный текст обработчика события `TDDGBugsDataModule.wbdpBugswaAddBugAction()`.

Листинг 36.12. Метод `TDDGBugsDataModule.wbdpBugswaAddBugAction()`, предназначенный для добавления в таблицу данных о новой ошибке

```
procedure TDDGBugsDataModule.wbdpBugswaAddBugAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
{ Добавление сведений об ошибке в базу данных.
  Для отображения информации о пользователе используется файл cookies. }
var
  SummaryStr,
  DetailsStr,
  AssignToStr,
  StatusStr: String;
  WebPage: TStringList;
  UserID: Integer;
  UserName: String;
  UserFName,
  UserLName: String;
  AssignedToUserName: String;
  PostSucceeded: boolean;

function GetAssignedToID: Integer;
var
  PosIdx: Integer;
begin
  PosIdx := Pos('-', AssignToStr);
```

```
  AssignedToUserName := Copy(AssignToStr, PosIdx+2, 100);
  tblUsers.Locate('UserName', AssignedToUserName, []);
  Result := tblUsers.FieldByName('UserID').AsInteger;
end;

function GetStatusID: Integer;
begin
  tblStatus.Locate('StatusTitle', StatusStr, []);
  Result := tblStatus.FieldByName('StatusID').AsInteger;
end;

procedure DoPostSuccessPage;
begin
  with WebPage do
  begin
    Add(Format('<H1>Thank you %s %s, your bug has been added.</H1>',
      [UserFName, UserLName]));
    Add(FormatDateTime('"<BR><BR>Bug Entered on:"  mmm dd, yyyy', Date));
    Add(Format('<BR>Bug Assigned to: %s', [AssignedToUserName]));
    Add(Format('<BR>Details: %s', [DetailsStr]));
    Add(Format('<BR>Status: %s', [StatusStr]));
  end;
end;

procedure DoPostFailPage;
begin
  WebPage.Add('<BR>Bug Entry failed.');
end;

begin

  // Считывание вставляемых полей
  SummaryStr  := Request.QueryFields.Values['Summary'];
  DetailsStr  := Request.QueryFields.Values['Details'];
  AssignToStr := Request.QueryFields.Values['AssignTo'];
  StatusStr   := Request.QueryFields.Values['Status'];


  // Считывание полей файла cookie
  UserID   := StrToInt(Request.CookieFields.Values['UserID']);
  UserName := Request.CookieFields.Values['UserName'];
  UserFName := Request.CookieFields.Values['UserFirstName'];
  UserLName := Request.CookieFields.Values['UserLastName'];

  // Данные, необходимые обработчику события AfterInsert
  FLoginUserID   := UserID;
  FLoginUserName := UserName;

  InsertBug;
  try
```

```
    tblBugs.FieldByName('SummaryDescription').AsString := SummaryStr;
    tblBugs.FieldByName('WhenReported').AsDateTime := Date;
    tblBugs.FieldByName('Details').AsString := DetailsStr;
    tblBugs.FieldByName('AssignedToUserID').AsInteger := GetAssignedToID;
    tblBugs.FieldByName('StatusID').AsInteger := GetStatusID;
    tblBugs.Post;
    PostSucceeded := True;

  except
    tblBugs.Cancel;
    PostSucceeded := False;
  end;

  WebPage := TStringList.Create;
  try
    AddHeader(WebPage);
    with WebPage do
    begin
      Add('<BODY>');

      if PostSucceeded then
        DoPostSuccessPage
      else
        DoPostFailPage;

      AddFooter(WebPage);
      Response.Content :=  WebPage.Text;
      Handled := True;
    end;
  finally
    WebPage.Free;
  end;

end;
```

Сначала в этом обработчике события считываются все введенные пользователем значения со страницы ввода данных об ошибке, показанной на рис. 36.7. Кроме того, считываются также и значения полей из файла cookie, созданного ранее. Обратите внимание на следующие три строки кода:

```
// Данные, необходимые для обработчика события AfterInsert
FLoginUserID   := UserID;
FLoginUserName := UserName;
```

Указанные значения требуются для обработчика события `AfterInsert` компонента `tblBugs`, который обрабатывает их следующим образом:

```
tblBugs.FieldByName('UserID').AsInteger := FLoginUserID;
tblBugs.FieldByName('UserNameLookup').AsString := FLoginUserName;
```

И, наконец, запись со сведениями о новой ошибке вставляется в таблицу tblBugs. При успешной вставке с помощью вызова метода DoPostSuccessPage() создается Web-страница. В противном случае вызывается метод DoPostFailPage(). Метод DoPostSuccessPage() просто выводит на экран данные об ошибке, в то время как DoPostFailPage() уведомляет о проблемах, возникших при добавлении сведений о новой ошибке.

Вспомните, что для получения действительных полей AssignToUserID и StatusID экземпляра tblBugs элементы управления работы с данными не используются. Страница ввода данных об ошибке предоставляет пользователю строки, которые содержат эти элементы в раскрывающихся комбинированных списках. Чтобы добавить соответствующие значения индексов подстановки в таблицу tblBugs, поиск выполняется по строкам, выбранным пользователем как в таблице tblUsers, так и в таблице tblStatus. Обратите внимание на то, что для выделения строки, с помощью которой осуществляется поиск по полю AssignToUserID, требуется выполнение незначительных преобразований (см. метод GetAssignToID()).

Резюме

В этой главе описан процесс создания Web-версии приложения, работающего с базой данных. Особое внимание обращалось на то, что при надлежащем проектировании для достижения поставленной цели требуются лишь небольшие модификации существующего кода (за исключением добавления кода, связанного со спецификой работы приложения в среде Web). И в самом деле, бо́льшая часть представленного здесь кода связана с построением HTML-документов, а не с манипуляциями базой данных. Попробуйте расширить функции этого демонстрационного приложения, а также перенести текст построения HTML-страниц в обычные HTML-файлы.

Приложения

ЧАСТЬ

VI

СООБЩЕНИЯ ОБ ОШИБКАХ И ИСКЛЮЧЕНИЯ 902

КОДЫ ОШИБОК BORLAND DATABASE ENGINE 935

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА 963

ОПИСАНИЕ СОДЕРЖИМОГО
ПРИЛАГАЕМОГО КОМПАКТ-ДИСКА 966

Приложение

A

Сообщения об ошибках и исключения

Уровни обработки 903

Ошибки времени выполнения 904

Разница между хорошей и отличной программой состоит в том, что первая хорошо работает, а вторая еще и хорошо *ошибается*. В среде Delphi ошибки, выявленные во время выполнения программ, фиксируются и обрабатываются как исключительные ситуации (ниже — просто исключения). Это позволяет программе выявить причину ошибки и попытаться восстановить нормальную работу (вернувшись в исходное состояние и применив другой подход) или, по крайней мере, корректно завершить свое выполнение (освободив занятые ресурсы, закрыв файлы и выведя на экран сообщение об ошибке), вместо грубого прерывания вычислительного процесса и внесения беспорядка в систему. Большинство исключений в среде Delphi генерируется и обрабатывается внутри самого приложения, и лишь некоторые из возникших во время выполнения ошибок действительно способны привести к аварийному останову программы.

В данном приложении перечислены наиболее типичные сообщения об ошибках, которые могут быть получены из приложения Delphi. Приведенное ниже описание этих сообщений поможет вам определить причины и условия возникновения ошибки. Обычно каждый добавляемый в среду разработки Delphi компонент имеет свой собственный набор сообщений об ошибках. Этот список постоянно пополняется. Поэтому в данном приложении содержатся лишь чаще всего встречающиеся сообщения, с которыми разработчик наверняка столкнется при создании и отладке приложений Delphi.

Уровни обработки

В каждой программе Delphi (по умолчанию) имеется два обработчика исключений разного уровня. Библиотека VCL предоставляет свой обработчик исключений, с которым программисту приходится иметь дело чаще всего. По существу, этот обработчик "отслеживает" точку входа оконной процедуры каждого объекта VCL. Если исключение было сгенерировано в тот момент, когда программа обрабатывала сообщение Windows (а этим она занимается 99% всего времени ее выполнения) и оно оказалось необработанным в коде приложения или используемого компонента VCL, то в конечном счете его обработка будет выполнена в оконной процедуре с помощью обработчика исключений библиотеки VCL, используемого по умолчанию. В этом обработчике вызывается метод `Application.HandleException`, который выводит диалоговое окно с текстом сообщения. После этого выполнение программы продолжается и на экране возможно появление окон с дополнительными сообщениями.

Обработчик самого низкого уровня находится в библиотеке времени выполнения Delphi (Runtime Library — RTL) и обрабатывает исключения на более низком уровне, по сравнению с обработчиком исключений библиотеки VCL, принимаемым по умолчанию. Если исключение возникло за пределами контекста обработки сообщений (например, во время запуска/завершения программы или во время выполнения кода обработчика VCL, используемого по умолчанию) и осталось необработанным, то его обработка осуществляется обработчиком RTL. На этом уровне нормальное выполнение программы обычно не возобновляется, поскольку не существует цикла обработки сообщений, способного обеспечить дальнейшее функционирование программы. При активизации используемого по умолчанию обработчика исключений библиотеки RTL на экран выводится подробное сообщение об ошибке, после чего выполнение приложения завершается.

Кроме текста сообщения об ошибке стандартный обработчик исключений библиотеки RTL сообщает также шестнадцатеричный адрес инструкции, при выполнении которой было сгенерировано исключение. Для того чтобы перейти к исходному коду программы, соответствующему этому адресу, выберите в интегрированной среде Delphi команду Search⇨Find Error, а затем в появившемся диалоговом окне введите адрес. Редактор кода Delphi переместит курсор в ту строку исходного текста программы, которая соответствует указанному адресу (если это окажется возможным).

Если в ответ на выбор указанной команды появится сообщение “Address not Found”, то это может означать, что ошибка произошла в другом модуле (например, если наличие в программе “дикого” указателя вызвало перезапись области памяти, используемой некоторым другим приложением). Однако, как правило, это сообщение означает, что в модуле отключен режим генерации отладочной информации о номерах строк ({$D-}) или отсутствует исходный код модуля, сгенерировавшего исключение. Перед компиляцией проекта выберите команду Project⇨Options. В появившемся диалоговом окне Project Options перейдите во вкладку Compiler и удостоверьтесь, что в секции Debugging установлен флажок опции, соответствующей режиму генерации отладочной информации. В том же диалоговом окне, проверьте, указаны ли в поле Search Path вкладки Directories/Conditionals все те каталоги, в которых находятся исходные тексты используемых модулей. Если файл с исходным кодом некоторого модуля не будет найден интегрированной средой Delphi, то номер строки исходного кода, соответствующей адресу возникшей ошибки, определить не удастся. Установив новые параметры, с помощью команды Project⇨Build All перекомпилируйте все приложение заново.

Ошибки времени выполнения

В этом разделе содержатся некоторые рекомендации, которые окажутся полезными при анализе ошибок, генерируемых либо в форме исключений, либо в форме отказов вызываемых функций программного интерфейса Win32.

Исключения

В этом разделе описаны исключения, которые могут быть сгенерированы компонентами библиотеки VCL. Помните, что в пользовательских компонентах и в вашем собственном программном тексте могут (а зачастую и должны) определяться дополнительные классы исключений, специфические для конкретных решаемых задач.

Некоторые из приведенных классов исключений описывают связанные с ошибками условия и представляют собой целые семейства ошибок. Взаимосвязь классов исключений организуется посредством создания класса исключения общего назначения, представляющего все семейство, а также специфических классов исключений, производных от основного класса. Таким образом, при обработке всех ошибок одного семейства в блоке except нужно использовать класс исключений общего назначения. Если же требуется обработать лишь определенные ошибки семейства, то в блоке обработки except необходимо указать соответствующие производные классы.

В приведенном ниже перечне семейства родственных классов исключений сгруппированы (с помощью отступов) под теми общими классами, от которых они произведены.

- Класс Exception. Это предок всех классов исключений. Использование этого класса формально не является ошибкой и вполне допустимо для быстрого достижения сиюминутных целей, однако в коммерческих программах, как правило, требуется различать множество отдельных семейств ошибок. В этом случае более правильным подходом будет создание для каждого семейства собственных специфических классов исключений, производных от класса Exception.
- Класс EAbort. Это “молчаливое” исключение, которое перехватывается стандартным обработчиком исключений библиотеки VCL. При этом пользователь о возникшей ошибке не информируется. Используйте класс EAbort в тех случаях, когда требуется прервать некоторый сложный процесс, не уведомляя об этом пользователя. Не забы-

вайте, что термины *исключение* и *ошибка* не являются эквивалентными. Исключение представляет собой способ изменения хода выполнения программы с целью проведения обработки ошибки (или для каких-нибудь иных целей).

- Класс EAccessViolation. Данное исключение фиксирует нарушение доступа и генерируется операционной системой. Обычно оно происходит при обращении к памяти по указателю со значением Nil или по указателю, в котором содержится некорректное значение.
- Класс EAssertionFailed. Это исключение генерируется в том случае, если переданные в процедуру Assert() операторы, привели к получению значения False.
- Класс EBitsError. Генерируется, если значения свойств Bits или Size объекта TBits вышли за допустимые границы.
- Класс EComponentError. Это исключение генерируется в двух ситуациях. Во-первых, оно происходит при попытке зарегистрировать компонент с помощью RegisterClasses() вне процедуры Register(). Во-вторых, оно имеет место, когда имя компонента некорректно или не является уникальным.
- Класс EControlC. Эта исключительная ситуация генерируется лишь для консольных приложений — в том случае, если пользователь прерывает их выполнение с помощью комбинации клавиш <Ctrl+C>.
- Класс EDbEditError. Генерируется в том случае, если пользователь ввел в компонент TMaskEdit или TDbEdit текст, не совместимый с текущей маской редактирования.
- Класс EDdeError. Эта ошибка происходит во время операции динамического обмена данными (DDE) при использовании компонента TDdeClientConv, TDdeClientItem, TDdeServerConv или TDdeServerItem.
- Класс EExternalException. Возникает в том случае, если операционной системой сгенерировано нераспознанное исключение.
- Класс EInOutError. Сообщает о том, что в программе произошла ошибка ввода/вывода. Это исключение генерируется лишь при включении в код директивы компилятора {$I+} или при установке в интегрированной среде флажка опции I/O Checking, расположенной во вкладке Compiler окна параметров проекта (Options⇨Project).
- Класс EIntError. Этот класс является предком всех классов математических исключений. Производными от него являются следующие классы.
  - Класс EDivByZero. Это исключение генерируется при делении на 0 целого числа и является результатом возникновения ошибки времени выполнения 200. Вот пример кода, который приведет к генерации этого исключения:

    ```
    var
      I: integer;
    begin
      I := 0;
      I := 10 div I;   { Здесь генерируется исключение }
    end;
    ```

  - Класс EIntOverflow. Это исключение генерируется при выполнении операции, вызывающей переполнение переменной целого типа, и представляет собой результат ошибки времени выполнения 215. Это исключение генерируется лишь при включении в код директивы компилятора {$Q+} или при установке в интегрированной сре-

де флажка опции Overflow Checking во вкладке Compiler окна параметров проекта (Options⇨Project). Вот пример кода, который приведет к генерации исключения EIntOverflow:

```
var
  l: longint;
begin
  l := MaxLongint;
  l := l * l; { Здесь генерируется исключение }
end;
```

- Класс ERangeError. Это исключение генерируется при попытке обращения к элементу массива за пределами его объявленных границ или при попытке сохранения в целочисленной переменной слишком большого значения. Исключение ERangeError представляет собой результат ошибки времени выполнения 201. Проверка допустимого диапазона будет выполняться лишь при включении в код директивы {$R+} или при установке флажка опции Range Checking во вкладке Compiler диалогового окна Options Project. Вот пример кода, в котором будет сгенерировано это исключение:

```
var
  a: array[1..16] of integer;
  i: integer;
begin
  i := 17;
  a[i] := 1; { Здесь генерируется исключение }
end;
```

- Класс EIntfCastError. Генерируется при попытке преобразования объекта или интерфейса в неподдерживаемый интерфейс.
- Класс EInvalidCast. Это исключение генерируется при попытке использования оператора as для преобразования в несовместимый класс и представляет собой результат ошибки времени выполнения 219. К генерации этого исключения приведет следующий код:

```
var
  B: TObject;
begin
  B := TButton.Create(nil);
{ Здесь генерируется исключение -
  класс TMemo не является предком класса TButton }
  with B as TMemo do
  ...
end;
```

- Класс EInvalidGraphic. Это исключение генерируется при попытке использования метода LoadFromFile() для загрузки файла в несовместимом графическом формате.
- Класс EInvalidGraphicOperation. Это исключение генерируется при попытке выполнения некорректной операции с графическим объектом (например, при попытке изменить размер объекта класса Ticon).
- Класс EInvalidOperation. Это исключение генерируется при попытке отображения или выполнения другой операции, требующей указания дескриптора окна, с управляющим элементом, не имеющим родителя. Например:

```
var
  b: TBitBtn;
begin
  b := TBitBtn.Create(Self);
  b.SetFocus;  { Здесь генерируется исключение }
end;
```

- Класс `EInvalidPointer`. Обычно это исключение генерируется при попытке освобождения некорректного или ранее освобожденного указателя с помощью вызова функций `Dispose()`, `FreeMem()` или деструктора класса. К генерации исключения `EInvalidPointer` приведет следующий код:

```
var
  p: pointer;
begin
  GetMem(p, 8);
  FreeMem(p, 8);
  FreeMem(p, 8); { Здесь генерируется исключение }
end;
```

- Класс `EListError`. Это исключение генерируется при попытке обращения к элементу с номером индекса, выходящим за пределы, установленные для потомка класса `TList`. Например:

```
var
  S: TStringList;
  Strng: String;
begin
  S := TStringList.Create;
  S.Add('One String');
  Strng := S.Strings[2]; { Здесь генерируется исключение }
end;
```

- Класс `EMathError`. Это базовый класс для следующих исключений, возникающих при выполнении операций с плавающей точкой.
  - Класс `EInvalidOp`. Это исключение генерируется при передаче неверной инструкции математическому сопроцессору. Обычно оно возникает лишь при работе с сопроцессором напрямую из кода на языке ассемблера.
  - Класс `EOverflow`. Это исключение генерируется в результате переполнения числа с плавающей точкой (т.е. в том случае, если значение, содержащееся в переменной с плавающей точкой, становится слишком большим). Это исключение соответствует ошибке времени выполнения `205`.
  - Класс `EUnderflow`. Генерируется в случае, если число слишком мал*о* для его размещения в переменной с плавающей точкой. Это исключение соответствует ошибке времени выполнения `206`.
  - Класс `EZeroDivide`. Это исключение генерируется при делении на 0 числа с плавающей точкой.
- Класс `EMCIDeviceError`. Это исключение информирует о том, что ошибка произошла в компоненте `TMediaPlayer`. Чаще всего оно генерируется при попытке проиграть некоторый мультимедиа-файл, формат которого несовместим с аппаратным обеспечением.

- Класс EMenuError. Обобщенное исключение, которое генерируется при возникновении ошибок в компоненте TMenu, TMenuItem или TPopupMenu.
- Класс EOleCtrlError. Это исключение зарезервировано для ошибок, происходящих в оболочках управляющих элементов ActiveX, однако в настоящее время в библиотеке VCL оно не используется.
- Класс EOleError. Это исключение генерируется при возникновении ошибок автоматизации OLE.
    - Класс EOleSysError. Генерируется процедурами OleCheck() и OleError() при возникновении ошибок во время вызовов функций интерфейса OLE API.
    - Класс EOleException. Возникает при появлении ошибок внутри функции или процедуры с типом вызова safecall.
- Класс EOutlineError. Это обобщенное исключение, генерирующееся при возникновении ошибок в компоненте TOutline.
- Класс EOutOfMemory. Это исключение генерируется при вызове функций New(), GetMem() или конструктора класса в том случае, если объем доступной памяти меньше, чем требуется для распределения. Это исключение соответствует ошибке времени выполнения 203.
    - Класс EOutOfResources. Это исключение генерируется в том случае, если Windows не в состоянии выполнить запрос на получение системного ресурса (например, на получение нового дескриптора). Появление исключения EOutOfResources зачастую свидетельствует об ошибке в драйвере видеоплаты, в особенности при работе в режимах с большим количеством цветов (32 Кбайт или 64 Кбайт). Если при переходе к стандартному VGA- или другому драйверу Windows с более низким разрешением это исключение исчезает, значит, скорее всего, используется некачественный видеодрайвер. Для получения обновленной версии драйвера обращайтесь к производителю видеоплаты.
- Класс EPackageError. Этот тип исключения генерируется в том случае, если ошибка произошла при загрузке, инициализации или завершении обработки пакета.
- Класс EParserError. Это исключение генерируется в том случае, если текстовый файл формы не удается проанализировать и преобразовать в двоичный .dfm-файл. Как правило, такое исключение является следствием синтаксической ошибки, возникшей при редактировании формы в интегрированной среде.
- Класс EPrinter. Это исключение генерируется при возникновении ошибок во время работы с объектом TPrinter.
- Класс EPrivilege. Это исключение указывает, что программа сделала попытку выполнить привилегированную инструкцию процессора.
- Класс EPropertyError. Это исключение генерируется при возникновении ошибок в редакторе свойств компонентов.
- Класс ERegistryException. Это исключение генерируется объектами TRegistry и TRegIniFile в случае ошибок, возникающих в процессе чтения или записи в системный реестр Windows.
- Класс EStackOverflow. Это исключение извещает о том, что при управлении стеком на уровне операционной системы возникла серьезная ошибка. Такая ошибка может произойти, если стек приложения был динамически увеличен операционной системой, однако в этот момент оказалось мало доступной памяти.

- Класс EReportError. Это обобщенное исключение генерируется при возникновении ошибок во время работы с компонентом Report.
- Класс EResNotFound. Данное исключение генерируется в том случае, если возникли проблемы при загрузке формы из .dfm-файла. Обычно это происходит при повреждении файла во время его редактирования, повреждении .dfm- или .exe-файла либо в случае, если .dfm-файл при компиляции не был связан с выполняемым файлом. Убедитесь, что в модуле формы не удалена и не изменена директива компилятора {$R *.DFM}.
- Класс EStreamError. Это исключение является базовым классом для всех исключений, связанных с потоками. Обычно оно свидетельствует о том, что при загрузке компонента TStrings из потока возникли проблемы. Более точно возможные ошибки характеризуются следующими производными классами.
  - Класс EFCreateError. Это исключение генерируется при ошибке во время создания файла потока. Часто оно указывает на то, что файл невозможно создать из-за неверного задания имени файла или из-за его использования другим процессом.
  - Класс EFilerError. Это исключение возникает при повторной попытке зарегистрировать класс с помощью процедуры RegisterClasses(). Кроме того, этот класс является базовым для других классов исключений.
    - Класс EClassNotFound. Это исключение генерируется, если имя класса компонента было считано из потока, но в соответствующем модуле отсутствует его объявление. Помните, что код и объявления, не используемые в программе, в результирующий EXE-файл компоновщиком Delphi внесены не будут.
    - Класс EInvalidImage. Это исключение возникает при попытке считывания компонентов из некорректного файла ресурсов.
    - Класс EMethodNotFound. Это исключение генерируется, если метод, определенный в .dfm-файле или файле ресурсов, отсутствует в соответствующем модуле. Это может произойти в том случае, если из модуля был удален код, а затем при компиляции выполняемого файла были проигнорированы предупреждения о ссылках на удаленный код в .dfm-файле.
    - Класс EReadError. Это исключение генерируется, если приложению не удалось считать из потока требуемое количество байтов (например, при неожиданном достижении конца файла) или если подпрограммам Delphi не удалось прочитать некоторое свойство.
  - Класс EFOPenError. Это исключение генерируется в том случае, когда заданный файл потока не может быть открыт. Такое обычно происходит, если файл не существует.
- Класс EStringListError. Это обобщенное исключение генерируется при возникновении ошибок во время работы с объектом TStringList.
- Класс EThread. Это исключение связано с компонентом TThread. В настоящее время исключение EThread генерируется лишь при попытке вызова метода Synchronize() для приостановленного потока.
- Класс ETreeViewError. Это исключение генерируется при передаче неверного индекса в метод или свойство компонента TTreeView.
- Класс EWin32Error. Это исключение генерируется при ошибках вызова функций программного интерфейса Win32 API. В сообщении, связанном с этим исключением, содержится код ошибки и ее описание.

Системные ошибки программного интерфейса Win32

Если ошибка возникла при вызове функции программного интерфейса Win32, то ее код обычно можно получить с помощью функции GetLastError(). Поскольку возвращаемое функцией GetLastError() значение имеет тип DWORD, иногда могут возникать затруднения с получением развернутого описания возникшей проблемы. В табл. А.1 приведен перечень констант и значений ошибок, а также краткое описание каждой из них.

Таблица А.1. Коды ошибок программного интерфейса Win32

Константа	Значение	Описание
ERROR_SUCCESS	0	Операция выполнена успешно
ERROR_INVALID_FUNCTION	1	Неверная функция
ERROR_FILE_NOT_FOUND	2	Системе не удается найти указанный файл
ERROR_PATH_NOT_FOUND	3	Системе не удается найти указанный путь
ERROR_TOO_MANY_OPEN_FILES	4	Системе не удается открыть файл
ERROR_ACCESS_DENIED	5	Попытка доступа отклонена
ERROR_INVALID_HANDLE	6	Неверный дескриптор
ERROR_ARENA_TRASHED	7	Разрушены управляющие блоки памяти
ERROR_NOT_ENOUGH_MEMORY	8	Недостаточно памяти для выполнения команды
ERROR_INVALID_BLOCK	9	Неверный адрес управляющего блока памяти
ERROR_BAD_ENVIRONMENT	10	Ошибка в среде
ERROR_BAD_FORMAT	11	Была сделана попытка загрузить программу в некорректном формате
ERROR_INVALID_ACCESS	12	Неверный код доступа
ERROR_INVALID_DATA	13	Ошибка в данных
ERROR_OUTOFMEMORY	14	Недостаточно памяти для завершения операции
ERROR_INVALID_DRIVE	15	Системе не удается найти указанный диск
ERROR_CURRENT_DIRECTORY	16	Не удается удалить каталог
ERROR_NOT_SAME_DEVICE	17	Системе не удается переместить файл на другой диск
ERROR_NO_MORE_FILES	18	Больше файлов не осталось
ERROR_WRITE_PROTECT	19	Носитель защищен от записи
ERROR_BAD_UNIT	20	Системе не удается найти указанное устройство
ERROR_NOT_READY	21	Устройство не готово
ERROR_BAD_COMMAND	22	Устройство не распознает команду
ERROR_CRC	23	Ошибка в данных (CRC)
ERROR_BAD_LENGTH	24	Некорректная длина команды, выданной программой

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_SEEK	25	Не удается найти заданную область или дорожку на диске
ERROR_NOT_DOS_DISK	26	Нет доступа к заданному диску или дискете
ERROR_SECTOR_NOT_FOUND	27	Не удается найти заданный сектор на диске
ERROR_OUT_OF_PAPER	28	Нет бумаги в принтере
ERROR_WRITE_FAULT	29	Системе не удается произвести запись на заданное устройство
ERROR_READ_FAULT	30	Системе не удается выполнить считывание с заданного устройства
ERROR_GEN_FAILURE	31	Присоединенное к системе устройство не работает
ERROR_SHARING_VIOLATION	32	Процесс не может получить доступ к файлу, поскольку последний используется другим процессом
ERROR_LOCK_VIOLATION	33	Процесс не может получить доступ к файлу, поскольку часть этого файла заблокирована другим процессом
ERROR_WRONG_DISK	34	В дисковод вставлена неверная дискета. Вставьте %2 (серийный номер тома: %3) в устройство %1
ERROR_SHARING_BUFFER_EXCEEDED	36	Открыто слишком много файлов для совместного использования
ERROR_HANDLE_EOF	38	Достигнут конец файла
ERROR_HANDLE_DISK_FULL	39	Недостаточно места на диске
ERROR_NOT_SUPPORTED	50	Сетевой запрос не поддерживается
ERROR_REM_NOT_LIST	51	Удаленный компьютер недоступен
ERROR_DUP_NAME	52	В сети существуют повторяющиеся имена
ERROR_BAD_NETPATH	53	Не найден сетевой путь
ERROR_NETWORK_BUSY	54	Сеть занята
ERROR_DEV_NOT_EXIST	55	Сетевой ресурс или устройство недоступно
ERROR_TOO_MANY_CMDS	56	Достигнуто предельное число команд NetBIOS
ERROR_ADAP_HDW_ERR	57	Аппаратная ошибка сетевого адаптера
ERROR_BAD_NET_RESP	58	Указанный сервер не может выполнить требуемую операцию
ERROR_UNEXP_NET_ERR	59	Неожиданная ошибка в сети
ERROR_BAD_REM_ADAP	60	Удаленный адаптер имеет несовместимый тип
ERROR_PRINTQ_FULL	61	Очередь печати переполнена
ERROR_NO_SPOOL_SPACE	62	На сервере отсутствует место для записи файла, выводимого на печать

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_PRINT_CANCELLED	63	Файл, ожидающий вывода на печать, был удален
ERROR_NETNAME_DELETED	64	Указанное сетевое имя недоступно
ERROR_NETWORK_ACCESS_DENIED	65	Отсутствует доступ к сети
ERROR_BAD_DEV_TYPE	66	Неверно указан тип сетевого ресурса
ERROR_BAD_NET_NAME	67	Не найдено сетевое имя
ERROR_TOO_MANY_NAMES	68	Превышен предел количества имен для локального сетевого адаптера компьютера
ERROR_TOO_MANY_SESS	69	Превышено предельное число сеансов NetBIOS
ERROR_SHARING_PAUSED	70	Удаленный сервер был приостановлен либо находится в процессе запуска
ERROR_REQ_NOT_ACCEP	71	Дополнительные подключения к этому удаленному компьютеру в настоящее время невозможны, поскольку достигнут предел подключений к компьютеру
ERROR_REDIR_PAUSED	72	Работа указанного принтера или дискового накопителя приостановлена
ERROR_FILE_EXISTS	80	Файл существует
ERROR_CANNOT_MAKE	82	Не удается создать файл или каталог
ERROR_FAIL_I24	83	Ошибка при обращении к прерыванию INT 24
ERROR_OUT_OF_STRUCTURES	84	Недостаточно памяти для обработки запроса
ERROR_ALREADY_ASSIGNED	85	Имя локального устройства уже используется
ERROR_INVALID_PASSWORD	86	Указан неверный сетевой пароль
ERROR_INVALID_PARAMETER	87	Неверный параметр
ERROR_NET_WRITE_FAULT	88	В сети произошла ошибка записи
ERROR_NO_PROC_SLOTS	89	В настоящее время система не может запустить другой процесс
ERROR_TOO_MANY_SEMAPHORES	100	Не удается создать еще один системный семафор
ERROR_EXCL_SEM_ALREADY_OWNED	101	Семафор монопольного доступа занят другим процессом
ERROR_SEM_IS_SET	102	Семафор установлен и не может быть закрыт
ERROR_TOO_MANY_SEM_REQUESTS	103	Семафор не может быть установлен повторно
ERROR_INVALID_AT_INTERRUPT_TIME	104	Запросы к семафорам монопольного доступа во время обработки прерываний запрещены
ERROR_SEM_OWNER_DIED	105	Предыдущий сеанс владения этим семафором завершен
ERROR_SEM_USER_LIMIT	106	Вставьте диск в устройство %1

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_DISK_CHANGE	107	Программа была остановлена, поскольку не был вставлен нужный диск
ERROR_DRIVE_LOCKED	108	Диск занят или заблокирован другим процессом
ERROR_BROKEN_PIPE	109	Канал был закрыт
ERROR_OPEN_FAILED	110	Системе не удается открыть указанное устройство или файл
ERROR_BUFFER_OVERFLOW	111	Указано слишком длинное имя файла
ERROR_DISK_FULL	112	Недостаточно места на диске
ERROR_NO_MORE_SEARCH_HANDLES	113	Исчерпаны внутренние идентификаторы файлов
ERROR_INVALID_TARGET_HANDLE	114	Некорректный результирующий внутренний идентификатор файла
ERROR_INVALID_CATEGORY	117	Вызов IOCTL произведен приложением неверно
ERROR_INVALID_VERIFY_SWITCH	118	Параметр проверки записи данных имеет неверное значение
ERROR_BAD_DRIVER_LEVEL	119	Требуемая команда системой не поддерживается
ERROR_CALL_NOT_IMPLEMENTED	120	Эта функция допустима только при работе в Windows NT
ERROR_SEM_TIMEOUT	121	Истек интервал ожидания семафора
ERROR_INSUFFICIENT_BUFFER	122	Область данных, переданная по системному вызову, слишком мала
ERROR_INVALID_NAME	123	Синтаксическая ошибка в имени файла, имени каталога или метке тома
ERROR_INVALID_LEVEL	124	Неверный уровень системного вызова
ERROR_NO_VOLUME_LABEL	125	У диска отсутствует метка тома
ERROR_MOD_NOT_FOUND	126	Не найден указанный модуль
ERROR_PROC_NOT_FOUND	127	Не найдена указанная процедура
ERROR_WAIT_NO_CHILDREN	128	Дочерние процессы, окончания которых требуется ожидать, отсутствуют
ERROR_CHILD_NOT_COMPLETE	129	Приложение %1 нельзя запустить в режиме Windows NT
ERROR_DIRECT_ACCESS_HANDLE	130	Попытка использовать дескриптор файла для открытия раздела диска и выполнения операции, отличающейся от ввода/вывода нижнего уровня
ERROR_NEGATIVE_SEEK	131	Попытка переместить указатель файла до его начала
ERROR_SEEK_ON_DEVICE	132	Указатель файла не может быть связан с заданным устройством или файлом

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_IS_JOIN_TARGET	133	Команды JOIN и SUBST нельзя использовать для дисков, уже содержащих объединенные диски
ERROR_IS_JOINED	134	Попытка применить команду JOIN или SUBST к диску, уже включенному в набор объединенных дисков
ERROR_IS_SUBSTED	135	Попытка применить команду JOIN или SUBST к диску, для которого подстановка пути уже была выполнена ранее
ERROR_NOT_JOINED	136	Попытка снять признак объединения с диска, для которого не выполнялась команда JOIN
ERROR_NOT_SUBSTED	137	Попытка удалить виртуальный диск, для которого не выполнялась команда SUBST
ERROR_JOIN_TO_JOIN	138	Попытка объединить диск с каталогом на объединенном диске
ERROR_SUBST_TO_SUBST	139	Попытка сопоставить диск каталогу на виртуальном диске
ERROR_JOIN_TO_SUBST	140	Попытка объединить диск с каталогом на виртуальном диске
ERROR_SUBST_TO_JOIN	141	Попытка сопоставить диск с каталогом, находящимся на объединенном диске
ERROR_BUSY_DRIVE	142	В настоящее время команду JOIN или SUBST выполнить невозможно
ERROR_SAME_DRIVE	143	Невозможно объединить (или сопоставить) диск с каталогом этого же диска
ERROR_DIR_NOT_ROOT	144	Этот каталог не является подкаталогом корневого каталога
ERROR_DIR_NOT_EMPTY	145	Каталог не пуст
ERROR_IS_SUBST_PATH	146	Указанный путь используется для виртуального диска
ERROR_IS_JOIN_PATH	147	Для обработки команды недостаточно ресурсов
ERROR_PATH_BUSY	148	Указанный путь в настоящее время использовать нельзя
ERROR_IS_SUBST_TARGET	149	Попытка объединить или сопоставить диск, каталог которого уже используется для сопоставления
ERROR_SYSTEM_TRACE	150	Сведения о трассировке в файле CONFIG.SYS не найдены, либо трассировка запрещена
ERROR_INVALID_EVENT_COUNT	151	Число событий семафора для DosMuxSemWait задано неверно

Константа	Значение	Описание
ERROR_TOO_MANY_MUXWAITERS	152	Не выполнен вызов DosMuxSemWait. Установлено слишком много семафоров
ERROR_INVALID_LIST_FORMAT	153	Некорректный вызов DosMuxSemWait
ERROR_LABEL_TOO_LONG	154	Длина введенной метки тома превышает допустимую длину в 11 символов. Первых 11 символов были записаны на диск, а все остальные автоматически удалены
ERROR_TOO_MANY_TCBS	155	Не удается создать еще один поток
ERROR_SIGNAL_REFUSED	156	Сигнал был отклонен процессом-получателем
ERROR_DISCARDED	157	Сегмент уже освобожден и не может быть заблокирован
ERROR_NOT_LOCKED	158	Сегмент уже был разблокирован ранее
ERROR_BAD_THREADID_ADDR	159	Некорректный адрес идентификатора потока
ERROR_BAD_ARGUMENTS	160	В DosExecPgm передана некорректная строка параметров
ERROR_BAD_PATHNAME	161	Указан неверный путь
ERROR_SIGNAL_PENDING	162	Сигнал ранее уже был отложен
ERROR_MAX_THRDS_REACHED	164	Создание дополнительных потоков невозможно
ERROR_LOCK_FAILED	167	Нельзя заблокировать область файла
ERROR_BUSY	170	Требуемый ресурс занят
ERROR_CANCEL_VIOLATION	173	Запрос на блокировку не соответствует указанной области снятия
ERROR_ATOMIC_LOCKS_NOT_SUPPORTED	174	Файловая система не поддерживает атомарные изменения типа блокировки
ERROR_INVALID_SEGMENT_NUMBER	180	Системой обнаружен неверный номер сегмента
ERROR_INVALID_ORDINAL	182	Операционная система не может запустить %1
ERROR_ALREADY_EXISTS	183	Невозможно создать файл, поскольку он уже существует
ERROR_INVALID_FLAG_NUMBER	186	Передан неверный флаг
ERROR_SEM_NOT_FOUND	187	Не найдено указанное имя системного семафора
ERROR_INVALID_STARTING_CODESEG	188	Операционная система не может запустить %1
ERROR_INVALID_STACKSEG	189	Операционная система не может запустить %1
ERROR_INVALID_MODULETYPE	190	Операционная система не может запустить %1
ERROR_INVALID_EXE_SIGNATURE	191	Не удается запустить %1 в Windows NT
ERROR_EXE_MARKED_INVALID	192	Операционная система не может запустить %1
ERROR_BAD_EXE_FORMAT	193	%1 не является приложением Windows NT

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_ITERATED_DATA_EXCEEDS_64k	194	Операционная система не может запустить %1
ERROR_INVALID_MINALLOCSIZE	195	Операционная система не может запустить %1
ERROR_DYNLINK_FROM_INVALID_RING	196	Операционная система не может запустить это приложение
ERROR_IOPL_NOT_ENABLED	197	Конфигурация операционной системы не предназначена для запуска этого приложения
ERROR_INVALID_SEGDPL	198	Операционная система не может запустить %1
ERROR_AUTODATASEG_EXCEEDS_64k	199	Операционная система не может запустить это приложение
ERROR_RING2SEG_MUST_BE_MOVABLE	200	Сегмент кода не может превышать 64 Кбайт
ERROR_RELOC_CHAIN_XEEDS_SEGLIM	201	Операционная система не может запустить %1
ERROR_INFLOOP_IN_RELOC_CHAIN	202	Операционная система не может запустить %1
ERROR_ENVVAR_NOT_FOUND	203	Системе не удается найти указанный параметр среды
ERROR_NO_SIGNAL_SENT	205	Ни один из процессов в дереве команды не имеет обработчика сигналов
ERROR_FILENAME_EXCED_RANGE	206	Имя файла или его расширение имеет слишком большую длину
ERROR_RING2_STACK_IN_USE	207	Стек 2-го кольца занят
ERROR_META_EXPANSION_TOO_LONG	208	Символы подстановки шаблонов (* и/или ?) заданы неверно либо определяют слишком много имен файлов
ERROR_INVALID_SIGNAL_NUMBER	209	Некорректный отправляемый сигнал
ERROR_THREAD_1_INACTIVE	210	Не удается установить обработчик сигналов
ERROR_LOCKED	212	Сегмент заблокирован и не может быть перемещен
ERROR_TOO_MANY_MODULES	214	К этой программе или модулю присоединено слишком много динамически подключаемых модулей
ERROR_NESTING_NOT_ALLOWED	215	Вызовы LoadModule не могут быть вложенными
ERROR_BAD_PIPE	230	Некорректное состояние канала
ERROR_PIPE_BUSY	231	Все экземпляры канала заняты
ERROR_NO_DATA	232	Канал закрывается
ERROR_PIPE_NOT_CONNECTED	233	Процессы отсутствуют с обоих концов канала
ERROR_MORE_DATA	234	Имеются дополнительные данные
ERROR_VC_DISCONNECTED	240	Сеанс был прекращен
ERROR_INVALID_EA_NAME	254	Имя расширенного атрибута было задано неверно

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_EA_LIST_INCONSISTENT	255	Расширенные атрибуты несовместимы
ERROR_NO_MORE_ITEMS	259	Дополнительные данные отсутствуют
ERROR_CANNOT_COPY	266	Не удается использовать программный интерфейс Copy
ERROR_DIRECTORY	267	Неверное имя каталога
ERROR_EAS_DIDNT_FIT	275	Расширенные атрибуты не помещаются в буфере
ERROR_EA_FILE_CORRUPT	276	Файл расширенных атрибутов в монтированной файловой системе поврежден
ERROR_EA_TABLE_FULL	277	Файл таблицы расширенных атрибутов переполнен
ERROR_INVALID_EA_HANDLE	278	Неверно указан дескриптор расширенного атрибута
ERROR_EAS_NOT_SUPPORTED	282	Монтированная файловая система не поддерживает расширенных атрибутов
ERROR_NOT_OWNER	288	Попытка освободить мьютекс, не принадлежащий процессу
ERROR_TOO_MANY_POSTS	298	Слишком много попыток занесения события для семафора
ERROR_PARTIAL_COPY	299	Запрос `Read/Write ProcessMemory` был выполнен только частично
ERROR_MR_MID_NOT_FOUND	317	В файле %2 не удается найти сообщение с номером $%1
ERROR_INVALID_ADDRESS	487	Попытка обращения к неверному адресу
ERROR_ARITHMETIC_OVERFLOW	534	Длина результата арифметической операции превысила 32 разряда
ERROR_PIPE_CONNECTED	535	С другой стороны канала присутствует процесс
ERROR_PIPE_LISTENING	536	Ожидается открытие процессом другой стороны канала
ERROR_EA_ACCESS_DENIED	994	Нет доступа к расширенному атрибуту
ERROR_OPERATION_ABORTED	995	Операция ввода/вывода была прервана из-за завершения потока или по запросу приложения
ERROR_IO_INCOMPLETE	996	Перекрытое событие ввода/вывода не находится в указанном состоянии
ERROR_IO_PENDING	997	Выполняется перекрытая операция ввода/вывода
ERROR_NOACCESS	998	Некорректный доступ по адресу памяти
ERROR_SWAPERROR	999	Ошибка при выполнении операции со страницей памяти

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_STACK_OVERFLOW	1001	Слишком глубокий уровень рекурсии. Стек переполнен
ERROR_INVALID_MESSAGE	1002	Окно не может взаимодействовать с отправленным сообщением
ERROR_CAN_NOT_COMPLETE	1003	Не удается завершить выполнение функции
ERROR_INVALID_FLAGS	1004	Некорректные флаги
ERROR_UNRECOGNIZED_VOLUME	1005	Том не содержит указанную файловую систему. Убедитесь, что загружены все требуемые драйверы файловой системы, а также в том, что не поврежден сам том
ERROR_FILE_INVALID	1006	Том для открытого файла был изменен извне, так что работа с файлом невозможна
ERROR_FULLSCREEN_MODE	1007	Требуемая операция не может быть выполнена в полноэкранном режиме
ERROR_NO_TOKEN	1008	Попытка ссылки на несуществующий элемент
ERROR_BADDB	1009	Повреждена база данных системного реестра
ERROR_BADKEY	1010	Неверное значение ключа системного реестра
ERROR_CANTOPEN	1011	Не удается открыть ключ системного реестра
ERROR_CANTREAD	1012	Не удается прочитать параметр системного реестра
ERROR_CANTWRITE	1013	Не удается записать параметр реестра
ERROR_REGISTRY_RECOVERED	1014	Один из файлов базы данных системного реестра потребовалось восстановить с помощью журнала или резервной копии. Восстановление прошло успешно
ERROR_REGISTRY_CORRUPT	1015	Системный реестр поврежден. Повреждена структура одного из его файлов. Возможно, поврежден образ файла в памяти или файл нельзя восстановить из-за отсутствия резервной копии/журнала
ERROR_REGISTRY_IO_FAILED	1016	Неустранимый сбой операции ввода/вывода, инициированной для системного реестра. Не удалось выполнить чтение, запись или запись буфера для одного из файлов, содержащего образ системного реестра
ERROR_NOT_REGISTRY_FILE	1017	При попытке загрузить или восстановить файл реестра выяснилось, что этот файл имеет неверный формат
ERROR_KEY_DELETED	1018	Попытка выполнить недопустимую операцию с параметром реестра, помеченным для удаления
ERROR_NO_LOG_SPACE	1019	Не удалось выделить требуемое место в журнале системного реестра

Константа	Значение	Описание
ERROR_KEY_HAS_CHILDREN	1020	Для параметра реестра, в котором уже содержатся вложенные параметры или значения, нельзя создать символьную ссылку
ERROR_CHILD_MUST_BE_VOLATILE	1021	Для временного родительского параметра нельзя создать статический подпараметр
ERROR_NOTIFY_ENUM_DIR	1022	Запрос на оповещение об изменениях завершается, однако данные не были возвращены в буфер вызывающей процедуры. Теперь для поиска изменений этой процедуре необходим перебор файлов
ERROR_DEPENDENT_SERVICES_RUNNING	1051	Команда остановки была отправлена службе, от которой зависят другие запущенные службы
ERROR_INVALID_SERVICE_CONTROL	1052	Команда для данной службы некорректна
ERROR_SERVICE_REQUEST_TIMEOUT	1053	Служба не ответила на запрос своевременно
ERROR_SERVICE_NO_THREAD	1054	Для службы не удалось создать поток
ERROR_SERVICE_DATABASE_LOCKED	1055	База данных службы заблокирована
ERROR_SERVICE_ALREADY_RUNNING	1056	Один экземпляр службы уже запущен
ERROR_INVALID_SERVICE_ACCOUNT	1057	Имя учетной записи задано неверно или его просто не существует
ERROR_SERVICE_DISABLED	1058	Указанная служба отключена и не может быть запущена
ERROR_CIRCULAR_DEPENDENCY	1059	Была сделана попытка установить циклическую зависимость между службами
ERROR_SERVICE_DOES_NOT_EXIST	1060	Указанная служба не установлена
ERROR_SERVICE_CANNOT_ACCEPT_CTRL	1061	В настоящее время служба не может принимать управляющие сообщения
ERROR_SERVICE_NOT_ACTIVE	1062	Служба не запущена
ERROR_FAILED_SERVICE_CONTROLLER_	1063	Процесс службы не может установить связь с ее контроллером
ERROR_EXCEPTION_IN_SERVICE	1064	При обработке управляющего запроса службой было сгенерировано исключение
ERROR_DATABASE_DOES_NOT_EXIST	1065	Указанной базы данных не существует
ERROR_SERVICE_SPECIFIC_ERROR	1066	Служба возвратила собственный код ошибки
ERROR_PROCESS_ABORTED	1067	Процесс был неожиданно завершен
ERROR_SERVICE_DEPENDENCY_FAIL	1068	Не удалось запустить зависимую службу или группу служб
ERROR_SERVICE_LOGON_FAILED	1069	Служба не запущена из-за сбоя, возникшего во время регистрации
ERROR_SERVICE_START_HANG	1070	Сразу после запуска служба “зависла” в состоянии незавершенного запуска

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_INVALID_SERVICE_LOCK	1071	Блокировка базы данных указанной службы наложена неверно
ERROR_SERVICE_MARKED_FOR_DELETE	1072	Указанная служба была помечена для удаления
ERROR_SERVICE_EXISTS	1073	Указанная служба уже существует
ERROR_ALREADY_RUNNING_LKG	1074	В настоящий момент системой используется последняя удачная конфигурация
ERROR_SERVICE_DEPENDENCY_DELETED	1075	Зависимой службы не существует или она помечена для удаления
ERROR_BOOT_ALREADY_ACCEPTED	1076	Текущая конфигурация уже была принята как последняя удачная конфигурация
ERROR_SERVICE_NEVER_STARTED	1077	С момента последней загрузки попытки запустить службу не предпринимались
ERROR_DUPLICATE_SERVICE_NAME	1078	Имя уже задействовано в качестве имени службы или имени дисплея службы
ERROR_END_OF_MEDIA	1100	Достигнут физический конец магнитной ленты
ERROR_FILEMARK_DETECTED	1101	На ленте достигнута метка файла
ERROR_BEGINNING_OF_MEDIA	1102	Обнаружено начало ленты или начало раздела ленты
ERROR_SETMARK_DETECTED	1103	При доступе к ленте достигнут конец набора файлов
ERROR_NO_DATA_DETECTED	1104	На ленте больше нет данных
ERROR_PARTITION_FAILURE	1105	На ленте не удается создать разделы
ERROR_INVALID_BLOCK_LENGTH	1106	Неправильный текущий размер блока при обращении к новой магнитной ленте из многотомного раздела
ERROR_DEVICE_NOT_PARTITIONED	1107	При загрузке магнитной ленты не найдены сведения о разделах
ERROR_UNABLE_TO_LOCK_MEDIA	1108	Не удается заблокировать механизм извлечения носителя
ERROR_UNABLE_TO_UNLOAD_MEDIA	1109	Не удается загрузить носитель
ERROR_MEDIA_CHANGED	1110	Носитель в устройстве мог быть заменен
ERROR_BUS_RESET	1111	Шина ввода/вывода была сброшена
ERROR_NO_MEDIA_IN_DRIVE	1112	В устройстве отсутствует носитель
ERROR_NO_UNICODE_TRANSLATION	1113	Символ Unicode не имеет отображения в целевой многобайтовой кодовой странице
ERROR_DLL_INIT_FAILED	1114	В процедуре инициализации динамической библиотеки произошел сбой
ERROR_SHUTDOWN_IN_PROGRESS	1115	Идет завершение работы системы

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_NO_SHUTDOWN_IN_PROGRESS	1116	Прервать завершение работы системы невозможно, поскольку оно не было инициализировано
ERROR_IO_DEVICE	1117	Запрос не был выполнен из-за ошибки ввода/вывода на устройстве
ERROR_SERIAL_NO_DEVICE	1118	Последовательные устройства не были успешно инициализированы. Драйвер последовательных устройств будет выгружен
ERROR_IRQ_BUSY	1119	Не удается открыть устройство, использующее общий с другими устройствами запрос на прерывание (IRQ). Как минимум одно устройство, использующее этот же запрос IRQ, уже было открыто
ERROR_MORE_WRITES	1120	Последовательная операция ввода/вывода была завершена в результате следующей операции записи в последовательный порт. (Значение IOCTL_SERIAL_XOFF_COUNTER достигло 0)
ERROR_COUNTER_TIMEOUT	1121	Последовательная операция ввода/вывода была завершена по истечении периода ожидания. (Значение IOCTL_SERIAL_XOFF_COUNTER не достигло 0)
ERROR_FLOPPY_ID_MARK_NOT_FOUND	1122	На гибком диске не обнаружена адресная метка идентификатора
ERROR_FLOPPY_WRONG_CYLINDER	1123	Обнаружено несоответствие между полем идентификатора сектора гибкого диска и адресом дорожки контроллера
ERROR_FLOPPY_UNKNOWN_ERROR	1124	Ошибка, возвращенная контроллером гибких дисков, драйвером не распознается
ERROR_FLOPPY_BAD_REGISTERS	1125	Контроллером гибких дисков возвращены некорректные значения регистров
ERROR_DISK_RECALIBRATE_FAILED	1126	При обращении к жесткому диску зафиксирован многократный сбой операции проверки
ERROR_DISK_OPERATION_FAILED	1127	При обращении к жесткому диску зафиксирован многократный сбой выполнения операции
ERROR_DISK_RESET_FAILED	1128	При обращении к жесткому диску потребовался сброс контроллера, однако даже его выполнить не удалось
ERROR_EOM_OVERFLOW	1129	Достигнут физический конец магнитной ленты
ERROR_NOT_ENOUGH_SERVER_MEMORY	1130	Для обработки команды памяти сервера недостаточно
ERROR_POSSIBLE_DEADLOCK	1131	Обнаружена вероятность возникновения взаимной блокировки

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_MAPPED_ALIGNMENT	1132	Базовый адрес или смещение в файле имеет неверное выравнивание
ERROR_SET_POWER_STATE_VETOED	1140	Попытка изменения режима питания была заблокирована другим приложением или драйвером
ERROR_SET_POWER_STATE_FAILED	1141	Сбой BIOS при попытке изменения режима питания
ERROR_OLD_WIN_VERSION	1150	Для указанной программы требуется более поздняя версия Windows
ERROR_APP_WRONG_OS	1151	Указанная программа не является программой для Windows или MS-DOS
ERROR_SINGLE_INSTANCE_APP	1152	Запуск нескольких экземпляров заданной программы невозможен
ERROR_RMODE_APP	1153	Запуск нескольких экземпляров заданной программы невозможен
ERROR_INVALID_DLL	1154	Поврежден один из файлов библиотек, необходимых для выполнения данного приложения
ERROR_NO_ASSOCIATION	1155	Файлам указанного типа для выполнения данной операции не поставлено в соответствие ни одно приложение
ERROR_DDE_FAIL	1156	Ошибка при пересылке команды приложению
ERROR_DLL_NOT_FOUND	1157	Не найден один из файлов библиотек, необходимых для выполнения данного приложения
ERROR_BAD_USERNAME	2202	Некорректное имя пользователя
ERROR_NOT_CONNECTED	2250	Сетевого подключения не существует
ERROR_OPEN_FILES	2401	Данное сетевое соединение содержит открытые файлы или ожидающие обработки запросы
ERROR_ACTIVE_CONNECTIONS	2402	Некоторые соединения все еще активны
ERROR_DEVICE_IN_USE	2404	Устройство используется одним из активных процессов и не может быть отсоединено
ERROR_BAD_DEVICE	1200	Указано неверное имя устройства
ERROR_CONNECTION_UNAVAIL	1201	Устройство в настоящее время не присоединено, однако сведения о нем в конфигурации присутствуют
ERROR_DEVICE_ALREADY_REMEMBERED	1202	Попытка записать сведения об устройстве, которые уже были записаны
ERROR_NO_NET_OR_BAD_PATH	1203	Для заданного сетевого пути нет ни одного сетевого провайдера
ERROR_BAD_PROVIDER	1204	Имя сетевого провайдера указано некорректно
ERROR_CANNOT_OPEN_PROFILE	1205	Не удается открыть профиль сетевого соединения

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_BAD_PROFILE	1206	Поврежден профиль сетевого соединения
ERROR_NOT_CONTAINER	1207	Перечисление для объектов, не являющихся контейнерами, невозможно
ERROR_EXTENDED_ERROR	1208	Обнаружена расширенная ошибка
ERROR_INVALID_GROUPNAME	1209	Неверный формат имени группы
ERROR_INVALID_COMPUTERNAME	1210	Неверный формат имени компьютера
ERROR_INVALID_EVENTNAME	1211	Неверный формат имени события
ERROR_INVALID_DOMAINNAME	1212	Неверный формат имени домена
ERROR_INVALID_SERVICENAME	1213	Неверный формат имени службы
ERROR_INVALID_NETNAME	1214	Неверный формат сетевого имени
ERROR_INVALID_SHARENAME	1215	Неверный формат имени ресурса
ERROR_INVALID_PASSWORDNAME	1216	Неверный формат пароля
ERROR_INVALID_MESSAGENAME	1217	Неверный формат имени сообщения
ERROR_INVALID_MESSAGEDEST	1218	Неверный формат адреса назначения сообщения
ERROR_SESSION_CREDENTIAL_CONFLIC	1219	Обнаружен конфликт между указанными и существующими данными учетной записи
ERROR_REMOTE_SESSION_LIMIT_EXCEE	1220	Неудачная попытка открытия сеанса на сетевом сервере, поскольку на нем уже открыто слишком много сеансов
ERROR_DUP_DOMAINNAME	1221	Имя рабочей группы или домена уже используется другим компьютером в сети
ERROR_NO_NETWORK	1222	Сеть отсутствует или не запущена
ERROR_CANCELLED	1223	Операция была отменена пользователем
ERROR_USER_MAPPED_FILE	1224	Указанная операция не может быть выполнена для файла с открытым разделом
ERROR_CONNECTION_REFUSED	1225	Удаленная система отклонила запрос на сетевое соединение
ERROR_GRACEFUL_DISCONNECT	1226	Сетевое соединение было корректно закрыто
ERROR_ADDRESS_ALREADY_ASSOCIATED	1227	Конечной точке сетевого транспорта адрес уже назначен
ERROR_ADDRESS_NOT_ASSOCIATED	1228	Конечной точке сети адрес еще не назначен
ERROR_CONNECTION_INVALID	1229	Попытка выполнить операцию для несуществующего сетевого соединения
ERROR_CONNECTION_ACTIVE	1230	Попытка выполнить недопустимую операцию для активного сетевого соединения
ERROR_NETWORK_UNREACHABLE	1231	Этому транспорту удаленная сеть не доступна

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_HOST_UNREACHABLE	1232	Этому транспорту не доступна удаленная система
ERROR_PROTOCOL_UNREACHABLE	1233	Транспортный протокол не может достичь удаленной системы
ERROR_PORT_UNREACHABLE	1234	По назначенному конечному сетевому адресу удаленной системы служба не запущена
ERROR_REQUEST_ABORTED	1235	Запрос был прерван
ERROR_CONNECTION_ABORTED	1236	Сетевое соединение было разорвано локальной системой
ERROR_RETRY	1237	Операция не может быть завершена. Ее следует повторить
ERROR_CONNECTION_COUNT_LIMIT	1238	Соединение с сервером невозможно, поскольку для данной учетной записи уже достигнут предел по числу одновременных подключений
ERROR_LOGIN_TIME_RESTRICTION	1239	Попытка регистрации во время дня, не предусмотренное для этого пользователя (учетной записи)
ERROR_LOGIN_WKSTA_RESTRICTION	1240	С этой рабочей станции данный пользователь зарегистрироваться не может
ERROR_INCORRECT_ADDRESS	1241	Для данной операции нельзя использовать сетевой адрес
ERROR_ALREADY_REGISTERED	1242	Служба уже зарегистрирована
ERROR_SERVICE_NOT_FOUND	1243	Указанной службы не существует
ERROR_NOT_AUTHENTICATED	1244	Запрошенная операция не была выполнена, поскольку пользователь не аутентифицирован
ERROR_NOT_LOGGED_ON	1245	Запрошенная операция не была выполнена, поскольку пользователь не зарегистрирован в сети. Указанной службы не существует
ERROR_CONTINUE	1246	Выполняемую операцию требуется продолжить
ERROR_ALREADY_INITIALIZED	1247	Попытка выполнить операцию инициализации, которая ранее уже была завершена
ERROR_NO_MORE_DEVICES	1248	Больше локальных устройств не существует
ERROR_NOT_ALL_ASSIGNED	1300	Пользователь обладает не всеми используемыми правами доступа
ERROR_SOME_NOT_MAPPED	1301	Между именами пользователей и идентификаторами безопасности не было установлено соответствие
ERROR_NO_QUOTAS_FOR_ACCOUNT	1302	Системные квоты для данной учетной записи не установлены

Константа	Значение	Описание
ERROR_LOCAL_USER_SESSION_KEY	1303	Ключ шифрования недоступен. Возвращается открытый ключ шифрования
ERROR_NULL_LM_PASSWORD	1304	Пароль NT слишком сложен и не может быть преобразован в пароль LAN Manager. Вместо пароля LAN Manager была возвращена пустая строка
ERROR_UNKNOWN_REVISION	1305	Версия изменений неизвестна
ERROR_REVISION_MISMATCH	1306	Две версии несовместимы
ERROR_INVALID_OWNER	1307	Этот код безопасности не может быть присвоен владельцу объекта
ERROR_INVALID_PRIMARY_GROUP	1308	Этот код безопасности не может быть присвоен основной группе объекта
ERROR_NO_IMPERSONATION_TOKEN	1309	Поток, который в данный момент не представляет клиента, предпринял попытку использования признака персонификации
ERROR_CANT_DISABLE_MANDATORY	1310	Группу нельзя отключить
ERROR_NO_LOGON_SERVERS	1311	Отсутствуют серверы, которые могли бы обработать запрос на регистрацию
ERROR_NO_SUCH_LOGON_SESSION	1312	Указанного сеанса регистрации не существует. Возможно, он уже был завершен
ERROR_NO_SUCH_PRIVILEGE	1313	Указанных прав доступа не существует
ERROR_PRIVILEGE_NOT_HELD	1314	Клиент не обладает требуемыми правами
ERROR_INVALID_ACCOUNT_NAME	1315	Указанное имя не является корректным именем учетной записи
ERROR_USER_EXISTS	1316	Пользователь с указанным именем уже существует
ERROR_NO_SUCH_USER	1317	Пользователя с указанным именем не существует
ERROR_GROUP_EXISTS	1318	Указанная группа уже существует
ERROR_NO_SUCH_GROUP	1319	Указанной группы не существует
ERROR_MEMBER_IN_GROUP	1320	Указанный пользователь уже является членом заданной группы, либо группа не может быть удалена, поскольку в ней содержится как минимум один пользователь
ERROR_MEMBER_NOT_IN_GROUP	1321	Указанный пользователь не является членом заданной группы
ERROR_LAST_ADMIN	1322	Последнюю учетную запись группы администраторов нельзя отключить или удалить
ERROR_WRONG_PASSWORD	1323	Не удается обновить пароль. Текущий пароль был задан неверно

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_ILL_FORMED_PASSWORD	1324	Не удается обновить пароль. Новый пароль содержит недопустимые символы
ERROR_PASSWORD_RESTRICTION	1325	Не удается обновить пароль. Было нарушено одно из правил обновления
ERROR_LOGON_FAILURE	1326	Неудачная регистрация: неизвестное имя пользователя или неверный пароль
ERROR_ACCOUNT_RESTRICTION	1327	Неудачная регистрация: доступ к учетной записи ограничен
ERROR_INVALID_LOGON_HOURS	1328	Неудачная регистрация: для учетной записи в данное время суток регистрация запрещена
ERROR_INVALID_WORKSTATION	1329	Неудачная регистрация: пользователь не может зарегистрироваться на этом компьютере
ERROR_PASSWORD_EXPIRED	1330	Неудачная регистрация: истек срок действия указанного пароля
ERROR_ACCOUNT_DISABLED	1331	Неудачная регистрация: учетная запись в настоящее время отключена
ERROR_NONE_MAPPED	1332	Именам пользователей не поставлены в соответствие идентификаторы защиты
ERROR_TOO_MANY_LUIDS_REQUESTED	1333	Одновременно запрошено слишком много локальных идентификаторов пользователей (Local User Identifier — LUID)
ERROR_LUIDS_EXHAUSTED	1334	Дополнительные локальные идентификаторы пользователей недоступны
ERROR_INVALID_SUB_AUTHORITY	1335	Часть идентификатора защиты, определяющая права доступа, запрещает данный тип использования
ERROR_INVALID_ACL	1336	Список управления доступом (ACL) имеет неверную структуру
ERROR_INVALID_SID	1337	Идентификатор защиты имеет неверную структуру
ERROR_INVALID_SECURITY_DESCR	1338	Дескриптор защиты данных имеет неверную структуру
ERROR_BAD_INHERITANCE_ACL	1340	Не удается построить наследуемый список управления доступом (Access Control List — ACL) или элемент этого списка (Access Control Entry — ACE)
ERROR_SERVER_DISABLED	1341	Сервер в настоящее время недоступен
ERROR_SERVER_NOT_DISABLED	1342	Сервер в настоящее время доступен
ERROR_INVALID_ID_AUTHORITY	1343	Указанное значение недопустимо как идентификатор защиты
ERROR_ALLOTTED_SPACE_EXCEEDED	1344	Для обновления данных защиты недостаточно памяти

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_INVALID_GROUP_ATTRIBUTES	1345	Указанные атрибуты неверны или не совместимы с атрибутами группы в целом
ERROR_BAD_IMPERSONATION_LEVEL	1346	Требуемый уровень персонификации либо не обеспечен, либо неверен
ERROR_CANT_OPEN_ANONYMOUS	1347	Не удается открыть элемент защиты данных анонимного уровня
ERROR_BAD_VALIDATION_CLASS	1348	Запрошен неверный класс сведений для проверки
ERROR_BAD_TOKEN_TYPE	1349	Тип элемента не соответствует требуемой операции
ERROR_NO_SECURITY_ON_OBJECT	1350	Операция, связанная с защитой данных, не может быть выполнена для незащищенного объекта
ERROR_CANT_ACCESS_DOMAIN_INFO	1351	Недоступен сервер Windows NT, или объекты внутри домена защищены. Требуемые сведения недоступны
ERROR_INVALID_SERVER_STATE	1352	Диспетчер защиты (SAM) или локальный сервер (LSA) не смог выполнить требуемую операцию, так как находится в неподходящем состоянии
ERROR_INVALID_DOMAIN_STATE	1353	Состояние домена не позволило выполнить нужную операцию
ERROR_INVALID_DOMAIN_ROLE	1354	Эта операция определена только для основного контроллера домена
ERROR_NO_SUCH_DOMAIN	1355	Указанного домена не существует
ERROR_DOMAIN_EXISTS	1356	Указанный домен уже существует
ERROR_DOMAIN_LIMIT_EXCEEDED	1357	Предпринята попытка превысить предельное число доменов, обслуживаемых одним сервером
ERROR_INTERNAL_DB_CORRUPTION	1358	Не удается завершить требуемую операцию из-за аварийного отказа носителя или разрушения структуры данных на диске
ERROR_INTERNAL_ERROR	1359	В базе данных учетных записей содержатся внутренние противоречия
ERROR_GENERIC_NOT_MAPPED	1360	Универсальные типы доступа содержатся в маске доступа, которая должна уже быть связана с нестандартными типами
ERROR_BAD_DESCRIPTOR_FORMAT	1361	Дескриптор безопасности имеет неверный формат (абсолютный или самоотносительный)
ERROR_NOT_LOGON_PROCESS	1362	Требуемое действие может использоваться только процессом регистрации. Вызвавший его процесс таковым не является

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_LOGON_SESSION_EXISTS	1363	Запуск нового сеанса работы с уже использующимся идентификатором невозможен
ERROR_NO_SUCH_PACKAGE	1364	Задан неизвестный пакет аутентификации
ERROR_BAD_LOGON_SESSION_STATE	1365	Текущее состояние сеанса регистрации не согласуется с требуемой операцией
ERROR_LOGON_SESSION_COLLISION	1366	Идентификатор сеанса регистрации уже используется
ERROR_INVALID_LOGON_TYPE	1367	Запрос на регистрацию задает неверный тип регистрации
ERROR_CANNOT_IMPERSONATE	1368	Невозможно обеспечить персонификацию через именованный канал до тех пор, пока из этого канала не считаны данные
ERROR_RXACT_INVALID_STATE	1369	Запрошенная операция несовместима с состоянием транзакции для ветви системного реестра
ERROR_RXACT_COMMIT_FAILURE	1370	Обнаружено повреждение внутренней базы данных защиты
ERROR_SPECIAL_ACCOUNT	1371	Данную операцию нельзя выполнить в отношении встроенных учетных записей
ERROR_SPECIAL_GROUP	1372	Операцию нельзя выполнить в отношении встроенной специальной группы
ERROR_SPECIAL_USER	1373	Операцию нельзя выполнить в отношении встроенного специального пользователя
ERROR_MEMBERS_PRIMARY_GROUP	1374	Нельзя удалить пользователя из группы, поскольку она является для него основной
ERROR_TOKEN_ALREADY_IN_USE	1375	Элемент уже используется в качестве первичного
ERROR_NO_SUCH_ALIAS	1376	Указанной локальной группы не существует
ERROR_MEMBER_NOT_IN_ALIAS	1377	Указанная учетная запись не входит в локальную группу
ERROR_MEMBER_IN_ALIAS	1378	Указанный пользователь уже является членом локальной группы
ERROR_ALIAS_EXISTS	1379	Указанная локальная группа уже существует
ERROR_LOGON_NOT_GRANTED	1380	Неудачная регистрация: для данного пользователя на этом компьютере выбранный режим входа не предусмотрен
ERROR_TOO_MANY_SECRETS	1381	Достигнуто предельное количество защищенных данных/ресурсов для одной системы
ERROR_SECRET_TOO_LONG	1382	Длина защищенных данных превышает максимально возможную
ERROR_INTERNAL_DB_ERROR	1383	В локальной базе данных защиты содержатся внутренние несоответствия

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_TOO_MANY_CONTEXT_IDS	1384	В процессе регистрации было использовано слишком много идентификаторов защиты
ERROR_LOGON_TYPE_NOT_GRANTED	1385	Неудачная регистрация: для данного пользователя выбранный режим на этом компьютере не предусмотрен
ERROR_NT_CROSS_ENCRYPTION_REQUIR	1386	Для смены пароля необходим зашифрованный пароль
ERROR_NO_SUCH_MEMBER	1387	Добавление нового члена в локальную группу невозможно, поскольку его не существует
ERROR_INVALID_MEMBER	1388	Добавление нового члена в локальную группу невозможно, поскольку он имеет неправильный тип учетной записи
ERROR_TOO_MANY_SIDS	1389	Задано слишком много идентификаторов защиты
ERROR_LM_CROSS_ENCRYPTION_REQUIR	1390	Для смены пароля необходим перекрестно зашифрованный пароль
ERROR_NO_INHERITANCE	1391	Список управления доступом не содержит наследуемых компонентов
ERROR_FILE_CORRUPT	1392	Файл или каталог повреждены. Чтение невозможно
ERROR_DISK_CORRUPT	1393	Структура диска повреждена. Чтение невозможно
ERROR_NO_USER_SESSION_KEY	1394	Для заданного сеанса регистрации отсутствует параметр сеанса пользователя
ERROR_LICENSE_QUOTA_EXCEEDED	1395	Количество соединений со службой ограничено лицензией. Дополнительные соединения в настоящее время невозможны
ERROR_INVALID_WINDOW_HANDLE	1400	Неверный дескриптор окна
ERROR_INVALID_MENU_HANDLE	1401	Неверный дескриптор меню
ERROR_INVALID_CURSOR_HANDLE	1402	Неверный дескриптор курсора
ERROR_INVALID_ACCEL_HANDLE	1403	Неверный дескриптор таблицы клавиатурных эквивалентов
ERROR_INVALID_HOOK_HANDLE	1404	Неверный дескриптор захвата
ERROR_INVALID_DWP_HANDLE	1405	Неверный дескриптор многооконной структуры
ERROR_TLW_WITH_WSCHILD	1406	Не удается создать дочернее окно верхнего уровня
ERROR_CANNOT_FIND_WND_CLASS	1407	Не удается найти класс окна
ERROR_WINDOW_OF_OTHER_THREAD	1408	Окно недоступно, оно принадлежит другому потоку
ERROR_HOTKEY_ALREADY_REGISTERED	1409	Клавиатурный эквивалент уже зарегистрирован

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_CLASS_ALREADY_EXISTS	1410	Класс уже существует
ERROR_CLASS_DOES_NOT_EXIST	1411	Класса не существует
ERROR_CLASS_HAS_WINDOWS	1412	У класса все еще имеются открытые окна
ERROR_INVALID_INDEX	1413	Неверный индекс
ERROR_INVALID_ICON_HANDLE	1414	Неверный дескриптор пиктограммы
ERROR_PRIVATE_DIALOG_INDEX	1415	Используются закрытые ключевые слова диалогового окна
ERROR_LISTBOX_ID_NOT_FOUND	1416	Идентификатор списка не найден
ERROR_NO_WILDCARD_CHARACTERS	1417	Символы подстановки не обнаружены
ERROR_CLIPBOARD_NOT_OPEN	1418	У потока отсутствует открытый буфер обмена
ERROR_HOTKEY_NOT_REGISTERED	1419	Клавиатурный эквивалент не зарегистрирован
ERROR_WINDOW_NOT_DIALOG	1420	Окно не является диалоговым
ERROR_CONTROL_ID_NOT_FOUND	1421	Не найден идентификатор управляющего элемента
ERROR_INVALID_COMBOBOX_MESSAGE	1422	Неверное сообщение для поля со списком, поскольку в нем нет строки ввода
ERROR_WINDOW_NOT_COMBOBOX	1423	Окно не является полем со списком
ERROR_INVALID_EDIT_HEIGHT	1424	Высота должна составлять меньше чем 256 пикселей
ERROR_DC_NOT_FOUND	1425	Неверный дескриптор контекста устройства
ERROR_INVALID_HOOK_FILTER	1426	Неверный тип процедуры захвата
ERROR_INVALID_FILTER_PROC	1427	Неверная процедура обработки
ERROR_HOOK_NEEDS_HMOD	1428	Без дескриптора модуля невозможно установить нелокальный захват
ERROR_GLOBAL_ONLY_HOOK	1429	Эта захват может быть установлен только глобально
ERROR_JOURNAL_HOOK_SET	1430	Процедура для обработки журнала уже установлена
ERROR_HOOK_NOT_INSTALLED	1431	Процедура обработки не установлена
ERROR_INVALID_LB_MESSAGE	1432	Для списка с одиночной выборкой это сообщение недопустимо
ERROR_SETCOUNT_ON_BAD_LB	1433	Параметр LB_SETCOUNT отправлен списку неверного типа
ERROR_LB_WITHOUT_TABSTOPS	1434	Список не поддерживает символов табуляции
ERROR_DESTROY_OBJECT_OF_OTHER_TH	1435	Нельзя уничтожить объект, созданный другим потоком
ERROR_CHILD_WINDOW_MENU	1436	Дочерние окна не могут иметь меню
ERROR_NO_SYSTEM_MENU	1437	Окно не имеет системного меню

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_INVALID_MSGBOX_STYLE	1438	Неверный стиль окна сообщения
ERROR_INVALID_SPI_VALUE	1439	Неверный общесистемный параметр (SPI_*)
ERROR_SCREEN_ALREADY_LOCKED	1440	Экран уже заблокирован
ERROR_HWNDS_HAVE_DIFF_PARENT	1441	Дескрипторы всех окон, входящих в многооконную структуру, должны иметь общий родительский дескриптор
ERROR_NOT_CHILD_WINDOW	1442	Окно не является дочерним
ERROR_INVALID_GW_COMMAND	1443	Неверная команда GW_*
ERROR_INVALID_THREAD_ID	1444	Неверный идентификатор потока
ERROR_NON_MDICHILD_WINDOW	1445	Невозможно обработать сообщение от окна, не являющегося компонентом многодокументного интерфейса
ERROR_POPUP_ALREADY_ACTIVE	1446	Всплывающее меню уже активно
ERROR_NO_SCROLLBARS	1447	Окно не имеет полос прокрутки
ERROR_INVALID_SCROLLBAR_RANGE	1448	Диапазон значений для полосы прокрутки не может превышать 0x7FFF
ERROR_INVALID_SHOWWIN_COMMAND	1449	Отобразить или удалить окно указанным способом невозможно
ERROR_EVENTLOG_FILE_CORRUPT	1500	Журнал событий поврежден
ERROR_EVENTLOG_CANT_START	1501	Не удается найти файл журнала событий, и поэтому служба протоколирования событий не запускается
ERROR_LOG_FILE_FULL	1502	Журнал событий переполнен
ERROR_EVENTLOG_FILE_CHANGED	1503	Журнал событий был изменен между двумя операциями чтения
ERROR_INVALID_USER_BUFFER	1784	Предоставленный пользователем буфер не подходит для указанной операции
ERROR_UNRECOGNIZED_MEDIA	1785	Не удается определить тип диска. Вероятно, он не отформатирован
ERROR_NO_TRUST_LSA_SECRET	1786	Рабочая станция не может участвовать в установлении доверительных отношений
ERROR_NO_TRUST_SAM_ACCOUNT	1787	База данных диспетчера учетных записей на сервере Windows NT не содержит записи для регистрации этого компьютера как рабочей станции через отношения доверительности
ERROR_TRUSTED_DOMAIN_FAILURE	1788	Установка доверительных отношений между первичным доменом и доменом-доверителем не состоялась

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_TRUSTED_RELATIONSHIP_FAILU	1789	Не удалось установить доверительные отношения между этой рабочей станцией и основным доменом
ERROR_TRUST_FAILURE	1790	Сбой при регистрации в сети
ERROR_NETLOGON_NOT_STARTED	1792	Вызываемая удаленная процедура для данного потока уже выполняется. Была предпринята попытка регистрации, однако сетевая служба регистрации не была запущена
ERROR_ACCOUNT_EXPIRED	1793	Срок действия учетной записи пользователя истек
ERROR_REDIRECTOR_HAS_OPEN_HANDLE	1794	Клиент сети занят и поэтому не может быть выгружен
ERROR_PRINTER_DRIVER_ALREADY_INS	1795	Указанный драйвер принтера уже установлен
ERROR_UNKNOWN_PORT	1796	Неизвестный порт
ERROR_UNKNOWN_PRINTER_DRIVER	1797	Неизвестный драйвер принтера
ERROR_UNKNOWN_PRINTPROCESSOR	1798	Неизвестный процессор печати
ERROR_INVALID_SEPARATOR_FILE	1799	Файл-разделитель задан неверно
ERROR_INVALID_PRIORITY	1800	Приоритет задан неверно
ERROR_INVALID_PRINTER_NAME	1801	Имя принтера задано неверно
ERROR_PRINTER_ALREADY_EXISTS	1802	Принтер уже существует
ERROR_INVALID_PRINTER_COMMAND	1803	Неверная команда принтера
ERROR_INVALID_DATATYPE	1804	Неверно задан тип данных
ERROR_INVALID_ENVIRONMENT	1805	Неверно задана среда
ERROR_NOLOGON_INTERDOMAIN_TRUST	1807	Других связей не существует. Используется доверительная учетная запись домена. Для доступа к серверу требуется глобальная или локальная учетная запись пользователя
ERROR_NOLOGON_WORKSTATION_TRUST	1808	Используется учетная запись компьютера. Для доступа к серверу требуется глобальная или локальная учетная запись пользователя
ERROR_NOLOGON_SERVER_TRUST_ACCOU	1809	Указанное имя является именем доверительного сервера. Для доступа к серверу воспользуйтесь глобальным или локальным именем пользователя
ERROR_DOMAIN_TRUST_INCONSISTENT	1810	Указанное имя или идентификатор защиты (SID) домена не совместимы со сведениями, полученными о домене через отношения доверенности
ERROR_SERVER_HAS_OPEN_HANDLES	1811	Сервер используется и не может быть выгружен

Продолжение табл. А.1

Константа	Значение	Описание
ERROR_RESOURCE_DATA_NOT_FOUND	1812	Файл образа не содержит раздела с ресурсами
ERROR_RESOURCE_TYPE_NOT_FOUND	1813	Указанный тип ресурса в файле образа отсутствует
ERROR_RESOURCE_NAME_NOT_FOUND	1814	Указанное имя ресурса в файле образа не найдено
ERROR_RESOURCE_LANG_NOT_FOUND	1815	Код языка для ресурсов в файле образа не найден
ERROR_NOT_ENOUGH_QUOTA	1816	Не удается обработать команду
ERROR_INVALID_TIME	1901	Время задано некорректно
ERROR_INVALID_FORM_NAME	1902	Имя формы задано некорректно
ERROR_INVALID_FORM_SIZE	1903	Размер формы задан некорректно
ERROR_ALREADY_WAITING	1904	Указанный дескриптор принтера уже ожидается
ERROR_PRINTER_DELETED	1905	Указанный принтер был удален
ERROR_INVALID_PRINTER_STATE	1906	Некорректное состояние принтера
ERROR_PASSWORD_MUST_CHANGE	1907	При первой регистрации пользователь должен сменить свой пароль
ERROR_DOMAIN_CONTROLLER_NOT_FOUN	1908	Не удается найти контроллер этого домена
ERROR_ACCOUNT_LOCKED_OUT	1909	Учетная запись пользователя заблокирована и ее нельзя использовать для регистрации
ERROR_NO_BROWSER_SERVERS_FOUND	6118	Для этой рабочей группы список серверов в данный момент недоступен
ERROR_INVALID_PIXEL_FORMAT	2000	Неверный формат пикселя
ERROR_BAD_DRIVER	2001	Выбран неверный драйвер
ERROR_INVALID_WINDOW_STYLE	2002	Тип или атрибут класса окна задан неверно
ERROR_METAFILE_NOT_SUPPORTED	2003	Требуемая операция для метафайлов не поддерживается
ERROR_TRANSFORM_NOT_SUPPORTED	2004	Требуемая операция преобразования не поддерживается
ERROR_CLIPPING_NOT_SUPPORTED	2005	Требуемая операция обрезки рисунка не поддерживается
ERROR_UNKNOWN_PRINT_MONITOR	3000	Указан неизвестный монитор печати
ERROR_PRINTER_DRIVER_IN_USE	3001	Указанный драйвер принтера уже используется
ERROR_SPOOL_FILE_NOT_FOUND	3002	Указанный файл спула не найден
ERROR_SPL_NO_STARTDOC	3003	Не был произведен вызов процедуры StartDocPrinter
ERROR_SPL_NO_ADDJOB	3004	Не был произведен вызов процедуры AddJob
ERROR_PRINT_PROCESSOR_ALREADY_IN	3005	Указанный процессор печати уже установлен

Окончание табл. А.1

Константа	Значение	Описание
ERROR_PRINT_MONITOR_ALREADY_INST	3006	Указанный монитор печати уже установлен
ERROR_WINS_INTERNAL	4000	При обработке команды произошла ошибка сервера WINS
ERROR_CAN_NOT_DEL_LOCAL_WINS	4001	Нельзя удалить локальную часть сервера WINS
ERROR_STATIC_INIT	4002	Ошибка при импортировании из файла
ERROR_INC_BACKUP	4003	Ошибка при архивации данных. Производилась ли ранее полная архивация?
ERROR_FULL_BACKUP	4004	Ошибка при архивации данных. Проверьте каталог, в котором архивируется база данных
ERROR_REC_NON_EXISTENT	4005	В базе данных сервера WINS имени не существует
ERROR_RPL_NOT_ALLOWED	4006	Без предварительной настройки партнера репликация невозможна

Приложение Б

Коды ошибок Borland Database Engine

Коды ошибок Borland Database Engine 936

Коды ошибок Borland Database Engine

При работе с Borland Database Engine (BDE) иногда может появиться диалоговое окно с сообщением об ошибке. В основном это происходит в том случае, если при установке программного обеспечения на машине заказчика или пользователя имели место различные проблемы и ошибки настройки. Обычно в таком диалоговом окне выводится шестнадцатеричный код ошибки, по которому можно будет найти ее описание. Однако преобразовать полученный код в осмысленное сообщение об ошибке — не самая простая задача. Чтобы упростить ее решение, ниже приводится описание отдельных значений. В табл. Б.1 перечислены все возможные коды ошибок и их описания (они приведены в двух вариантах: в стандартном, т.е. выводимом BDE, и в переводе на русский язык).

Таблица Б.1. Коды ошибок Borland Database Engine

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
0	0000	Successful completion	Успешное завершение
33	0021	System error	Системная ошибка
34	0022	Object of interest not found	Интересующий объект не найден
35	0023	Physical data corruption	Физическое повреждение данных
36	0024	I/O-related error	Ошибка ввода/вывода
37	0025	Resource or limit error	Ошибка ресурса или ограничений
38	0026	Data integrity violation	Нарушение целостности данных
39	0027	Invalid request	Некорректный запрос
40	0028	Lock violation	Нарушение блокировки
41	0029	Access/security Violation	Нарушение системы безопасности
42	002A	Invalid context	Неверный контекст
43	002B	OS error	Ошибка операционной системы
44	002C	Network error	Сетевая ошибка
45	002D	Optional parameter	Необязательный параметр
46	002E	Query processor	Процессор запросов
47	002F	Version mismatch	Несоответствие версии
48	0030	Capability not supported	Совместимость не поддерживается
49	0031	System configuration error	Ошибка конфигурации системы
50	0032	Warning	Предупреждение
51	0033	Miscellaneous	Разное
52	0034	Compatibility error	Ошибка совместимости
62	003E	Driver-specific error	Ошибка драйвера
63	003F	Internal symbol	Внутренний символ
256	0100	KEYVIOL	KEYVIOL — нарушение уникальности ключа
257	0101	PROBLEMS	PROBLEMS — проблема

Продолжение табл. Б.1

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
258	0102	CHANGED	CHANGED — изменено
512	0200	Production index file missing, corrupt, or cannot interpret index key	Отсутствие или повреждение индексного файла либо ошибка интерпретации индекса
513	0201	Open read-only	Открытие “только для чтения”
514	0202	Open the table in read only mode	Открытие таблицы в режиме “только для чтения”
515	0203	Open and detach	Открытие и отсоединение
516	0204	Open the table and detach the production index file	Открытие таблицы и отсоединение индексного файла
517	0205	Fail open	Ошибка открытия
518	0206	Do not open the table	Таблица не открыта
519	0207	Convert non-dBASE index	Конвертирование не dBASE-индекса
520	0208	Convert production index to dBASE format	Конвертирование индекса в dBASE-формат
521	0209	BLOB file not found	BLOB-файл не найден
522	020A	Open without BLOB file	Открытие без файла BLOB
523	020B	Open the table without the BLOB file	Открытие таблицы без файла BLOB
524	020C	Empty all BLOB fields	Все поля BLOB пусты
525	020D	Reinitialize BLOB file and LOSE all BLOBs	Реинициализация BLOB-файла и потеря всех объектов
526	020E	Fail open	Ошибка открытия
527	020F	Do not open the table	Таблица не открыта
528	0210	Import non-dBASE BLOB file	Импортирование не-dBASE BLOB-файла
529	0211	Import BLOB file to dBASE format	Импортирование BLOB-файла в формат dBASE
530	0212	Open as non-dBASE table	Открытие как не dBASE-таблицы
531	0213	Open table and BLOB file in its native format	Открытие таблицы и BLOB-файла в “родном” формате
532	0214	Production index language driver mismatch	Драйвер языка не соответствует используемому индексу
533	0215	Production index damaged	Используемый индекс поврежден
534	0216	Rebuild production Index	Перестройка используемого индекса
535	0217	Rebuild all the production Indexes	Перестройка всех используемых индексов
1024	0400	Lookup table not found or corrupt	Справочная таблица не найдена или повреждена
1025	0401	BLOB file not found or corrupt	BLOB-файл не найден или поврежден
1026	0402	Open read only	Открытие в режиме “только для чтения”

Продолжение табл. Б.1

Код ошибки		Строка сообщения	
Десятичный	**Шестнадцатеричный**	**Оригинал**	**Перевод**
1027	0403	Open the table in read only mode	Открытие таблицы в режиме “только для чтения”
1028	0404	Fail open	Ошибка открытия
1029	0405	Do not open the table	Не открывать таблицу
1030	0406	Remove lookup	Удалить справочную таблицу
1031	0407	Remove link to lookup table	Удалить связь со справочной таблицей
1280	0500	Dictionary object exists	Объект словаря существует
1281	0501	Skip this object	Пропустить этот объект
1282	0502	Skip importing this object and its associated relationships	Пропустить импортирование этого объекта и ассоциированных с ним связей
1283	0503	Use existing object	Использовать существующий объект
1284	0504	Use existing dictionary object for relationships	Использовать для связей существующий словарь
1285	0505	Abort	Останов
1286	0506	Abort the operation	Прекратить операцию
1287	0507	Dictionary object import failed	Ошибочное импортирование объекта словаря
4608	1200	SQL Unknown	SQL — не распознано
4609	1201	SQL Prepare	SQL — подготовка
4610	1202	SQL Execute	SQL — выполнение
4611	1203	SQL Error	SQL — ошибка
4612	1204	SQL STMT	SQL — команда
4613	1205	SQL Connect	SQL — подключение
4614	1206	SQL Transact	SQL — транзакция
4615	1207	SQL Blob I/O	SQL — ввод Blob
4616	1208	SQL Misc	SQL — прочее
4617	1209	SQL Vendor	SQL — поставщик
4618	120A	ORACLE — orlon	ORACLE — операция orlon
4619	120B	ORACLE — olon	ORACLE — операция olon
4620	120C	ORACLE — ologof	ORACLE — операция ologof
4621	120D	ORACLE — ocon	ORACLE — операция ocon
4622	120E	ORACLE — ocof	ORACLE — операция ocof
4623	120F	ORACLE — oopen	ORACLE — операция oopen
4624	1210	ORACLE — osql3	ORACLE — операция osql3
4625	1211	ORACLE — odsc	ORACLE — операция odsc
4626	1212	ORACLE — odefin	ORACLE — операция odefin
4627	1213	ORACLE — obndrv	ORACLE — операция obndrv

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
4628	1214	ORACLE — obndrvn	ORACLE — операция obndrvn
4629	1215	ORACLE — oexec	ORACLE — операция oexec
4630	1216	ORACLE — ofetch	ORACLE — операция ofetch
4631	1217	ORACLE — ofen	ORACLE — операция ofen
4632	1218	ORACLE — ocan	ORACLE — операция ocan
4633	1219	ORACLE — oclose	ORACLE — операция oclose
4634	121A	ORACLE — oerhms	ORACLE — операция oerhms
4635	121B	ORACLE — oparse	ORACLE — операция oparse
4636	121C	ORACLE — oflng	ORACLE — операция oflng
4637	121D	ORACLE — odessp	ORACLE — операция odessp
4638	121E	ORACLE — odescr	ORACLE — операция odescr
4639	121F	ORACLE — oexn	ORACLE — операция oexn
4648	1228	INTRBASE — isc_attach_database	INTRBASE — операция isc_attach_database
4649	1229	INTRBASE — isc_blob_default_desc	INTRBASE — операция isc_blob_default_desc
4650	122A	INTRBASE — isc_blob_gen_bpb	INTRBASE — операция isc_blob_gen_bpb
4651	122B	INTRBASE — isc_blob_info	INTRBASE — операция isc_blob_info
4652	122C	INTRBASE — isc_blob_lookup_desc	INTRBASE — операция isc_blob_lookup_desc
4653	122D	INTRBASE — isc_close_blob	INTRBASE — операция isc_close_blob
4654	122E	INTRBASE — isc_commit_retaining	INTRBASE — операция isc_commit_retaining
4655	122F	INTRBASE — isc_commit_transaction	INTRBASE — операция isc_commit_transaction
4656	1230	INTRBASE — isc_create_blob	INTRBASE — операция isc_create_blob
4657	1231	INTRBASE — isc_create_blob2	INTRBASE — операция isc_create_blob2
4658	1232	INTRBASE — isc_decode_date	INTRBASE — операция isc_decode_date
4659	1233	INTRBASE — isc_detach_database	INTRBASE — операция isc_detach_database
4660	1234	INTRBASE — isc_dsql_allocate_statement	INTRBASE — операция isc_dsql_allocate_statement
4661	1235	INTRBASE — isc_dsql_execute	INTRBASE — операция isc_dsql_execute
4662	1236	INTRBASE — isc_dsql_execute2	INTRBASE — операция isc_dsql_execute2

Продолжение табл. Б.1

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
4663	1237	INTRBASE — isc_dsql_fetch	INTRBASE — операция isc_dsql_fetch
4664	1238	INTRBASE — isc_dsql_free_statement	INTRBASE — операция isc_dsql_free_statement
4665	1239	INTRBASE — isc_dsql_prepare	INTRBASE — операция isc_dsql_prepare
4666	123A	INTRBASE — isc_dsql_set_cursor_name	INTRBASE — операция isc_dsql_set_cursor_name
4667	123B	INTRBASE — isc_dsql_sql_info	INTRBASE — операция isc_dsql_sql_info
4668	123C	INTRBASE — isc_encode_date	INTRBASE — операция isc_encode_date
4669	123D	INTRBASE — isc_get_segment	INTRBASE — операция isc_get_segment
4670	123E	INTRBASE — isc_interprete	INTRBASE — операция isc_interprete
4671	123F	INTRBASE — isc_open_blob	INTRBASE — операция isc_open_blob
4672	1240	INTRBASE — isc_open_blob2	INTRBASE — операция isc_open_blob2
4673	1241	INTRBASE — isc_put_segment	INTRBASE — операция isc_put_segment
4674	1242	INTRBASE — isc_rollback_transaction	INTRBASE — операция isc_rollback_transaction
4675	1243	INTRBASE — isc_sqlcode	INTRBASE — операция isc_sqlcode
4676	1244	INTRBASE — isc_start_transaction	INTRBASE — операция isc_start_transaction
4677	1245	INTRBASE — isc_vax_integer	INTRBASE — операция isc_vax_integer
4688	1250	MSSQL — dbbind	MSSQL — операция dbbind
4689	1251	MSSQL — dbcmd	MSSQL — операция dbcmd
4690	1252	MSSQL — dbcancel	MSSQL — операция dbcancel
4691	1253	MSSQL — dbclose	MSSQL — операция dbclose
4692	1254	MSSQL — dbcollen	MSSQL — операция dbcollen
4693	1255	MSSQL — dbcolname	MSSQL — операция dbcolname
4694	1256	MSSQL — dbcoltype	MSSQL — операция dbcoltype
4695	1257	MSSQL — dbconvert	MSSQL — операция dbconvert
4696	1258	MSSQL — dbdataready	MSSQL — операция dbdataready
4697	1259	MSSQL — dbdatlen	MSSQL — операция dbdatlen
4698	125A	MSSQL — dberrhandle	MSSQL — операция dberrhandle
4699	125B	MSSQL — dbfreebuf	MSSQL — операция dbfreebuf
4700	125C	MSSQL — dbfreelogin	MSSQL — операция dbfreelogin
4701	125D	MSSQL — dbhasretstat	MSSQL — операция dbhasretstat

Продолжение табл. Б.1

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
4702	125E	MSSQL — dbinit	MSSQL — операция dbinit
4703	125F	MSSQL — dblogin	MSSQL — операция dblogin
4704	1260	MSSQL — dbmoretext	MSSQL — операция dbmoretext
4705	1261	MSSQL — dbmsghandle	MSSQL — операция dbmsghandle
4706	1262	MSSQL — dbnextrow	MSSQL — операция dbnextrow
4707	1263	MSSQL — dbnumcols	MSSQL — операция dbnumcols
4708	1264	MSSQL — dbnumrets	MSSQL — операция dbnumrets
4709	1265	MSSQL — dbopen	MSSQL — операция dbopen
4710	1266	MSSQL — dbresults	MSSQL — операция dbresults
4711	1267	MSSQL — dbretdata	MSSQL — операция dbretdata
4712	1268	MSSQL — dbretlen	MSSQL — операция dbretlen
4713	1269	MSSQL — dbretstatus	MSSQL — операция dbretstatus
4714	126A	MSSQL — dbrpcinit	MSSQL — операция dbrpcinit
4715	126B	MSSQL — dbrpcparam	MSSQL — операция dbrpcparam
4716	126C	MSSQL — dbrpcsend	MSSQL — операция dbrpcsend
4717	126D	MSSQL — dbsetlogintime	MSSQL — операция dbsetlogintime
4718	126E	MSSQL — dbsetmaxprocs	MSSQL — операция dbsetmaxprocs
4719	126F	MSSQL — dbsetopt	MSSQL — операция dbsetopt
4720	1270	MSSQL — dbsettime	MSSQL — операция dbsettime
4721	1271	MSSQL — dbsqlexec	MSSQL — операция dbsqlexec
4722	1272	MSSQL — dbsqlok	MSSQL — операция dbsqlok
4723	1273	MSSQL — dbsqlsend	MSSQL — операция dbsqlsend
4724	1274	MSSQL — dbtxptr	MSSQL — операция dbtxptr
4725	1275	MSSQL — dbtxtimestamp	MSSQL — операция dbtxtimestamp
4726	1276	MSSQL — dbtxtsnewval	MSSQL — операция dbtxtsnewval
4727	1277	MSSQL — dbuse	MSSQL — операция dbuse
4728	1278	MSSQL — dbwinexit	MSSQL — операция dbwinexit
4729	1279	MSSQL — dbwritetext	MSSQL — операция dbwritetext
4738	1282	ODBC — SQLAllocConnect	ODBC — операция SQLAllocConnect
4739	1283	ODBC — SQLAllocEnv	ODBC — операция SQLAllocEnv
4740	1284	ODBC — SQLAllocStmt	ODBC — операция SQLAllocStmt
4741	1285	ODBC — SQLBindCol	ODBC — операция SQLBindCol
4742	1286	ODBC — SQLBindParameter	ODBC — операция SQLBindParameter
4743	1287	ODBC — SQLCancel	ODBC — операция SQLCancel
4744	1288	ODBC — SQLColumns	ODBC — операция SQLColumns

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
4745	1289	ODBC — SQLConnect	ODBC — операция SQLConnect
4746	128A	ODBC — SQLDataSources	ODBC — операция SQLDataSources
4747	128B	ODBC — SQLDescribeCol	ODBC — операция SQLDescribeCol
4748	128C	ODBC — SQLDisconnect	ODBC — операция SQLDisconnect
4750	128E	ODBC — SQLError	ODBC — операция SQLError
4751	128F	ODBC — SQLExecDirect	ODBC — операция SQLExecDirect
4752	1290	ODBC — SQLExtendedFetch	ODBC — операция SQLExtendedFetch
4753	1291	ODBC — SQLFetch	ODBC — операция SQLFetch
4754	1292	ODBC — SQLFreeConnect	ODBC — операция SQLFreeConnect
4755	1293	ODBC — SQLFreeEnv	ODBC — операция SQLFreeEnv
4756	1294	ODBC — SQLFreeStmt	ODBC — операция SQLFreeStmt
4757	1295	ODBC — SQLGetConnectOption	ODBC — операция SQLGetConnectOption
4758	1296	ODBC — SQLGetCursorName	ODBC — операция SQLGetCursorName
4760	1298	ODBC — SQLGetFunctions	ODBC — операция SQLGetFunctions
4761	1299	ODBC — SQLGetInfo	ODBC — операция SQLGetInfo
4762	129A	ODBC — SQLGetTypeInfo	ODBC — операция SQLGetTypeInfo
4763	129B	ODBC — SQLNumResultCols	ODBC — операция SQLNumResultCols
4764	129C	ODBC — SQLProcedures	ODBC — операция SQLProcedures
4765	129D	ODBC — SQLProcedureColumns	ODBC — операция SQLProcedureColumns
4766	129E	ODBC — SQLRowCount	ODBC — операция SQLRowCount
4767	129F	ODBC — SQLSetConnectOption	ODBC — операция SQLSetConnectOption
4768	12A0	ODBC — SQLSetCursorName	ODBC — операция SQLSetCursorName
4769	12A1	ODBC — SQLSetParam	ODBC — операция SQLSetParam
4770	12A2	ODBC — SQLSetStmtOption	ODBC — операция SQLSetStmtOption
4771	12A3	ODBC — SQLStatistics	ODBC — операция SQLStatistics
4772	12A4	ODBC — SQLTables	ODBC — операция SQLTables
4773	12A5	ODBC — SQLTransact	ODBC — операция SQLTransact
4788	12B4	SYBASE — dbbind	SYBASE — операция dbbind
4789	12B5	SYBASE — dbcmd	SYBASE — операция dbcmd
4790	12B6	SYBASE — dbcancel	SYBASE — операция dbcancel
4791	12B7	SYBASE — dbclose	SYBASE — операция dbclose
4792	12B8	SYBASE — dbcollen	SYBASE — операция dbcollen
4793	12B9	SYBASE — dbcolname	SYBASE — операция dbcolname
4794	12BA	SYBASE — dbcoltype	SYBASE — операция dbcoltype
4795	12BB	SYBASE — dbconvert	SYBASE — операция dbconvert

Продолжение табл. Б.1

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
4796	12BC	SYBASE — dbpoll	SYBASE — операция dbpoll
4797	12BD	SYBASE — dbdatlen	SYBASE — операция dbdatlen
4798	12BE	SYBASE — dberrhandle	SYBASE — операция dberrhandle
4799	12BF	SYBASE — dbfreebuf	SYBASE — операция dbfreebuf
4800	12C0	SYBASE — dbloginfree	SYBASE — операция dbloginfree
4801	12C1	SYBASE — dbhasretstat	SYBASE — операция dbhasretstat
4802	12C2	SYBASE — dbinit	SYBASE — операция dbinit
4803	12C3	SYBASE — dblogin	SYBASE — операция dblogin
4804	12C4	SYBASE — dbmoretext	SYBASE — операция dbmoretext
4805	12C5	SYBASE — dbmsghandle	SYBASE — операция dbmsghandle
4806	12C6	SYBASE — dbnextrow	SYBASE — операция dbnextrow
4807	12C7	SYBASE — dbnumcols	SYBASE — операция dbnumcols
4808	12C8	SYBASE — dbnumrets	SYBASE — операция dbnumrets
4809	12C9	SYBASE — dbopen	SYBASE — операция dbopen
4810	12CA	SYBASE — dbresults	SYBASE — операция dbresults
4811	12CB	SYBASE — dbretdata	SYBASE — операция dbretdata
4812	12CC	SYBASE — dbretlen	SYBASE — операция dbretlen
4813	12CD	SYBASE — dbretstatus	SYBASE — операция dbretstatus
4814	12CE	SYBASE — dbrpcinit	SYBASE — операция dbrpcinit
4815	12CF	SYBASE — dbrpcparam	SYBASE — операция dbrpcparam
4816	12D0	SYBASE — dbrpcsend	SYBASE — операция dbrpcsend
4817	12D1	SYBASE — dbsetlogintime	SYBASE — операция dbsetlogintime
4818	12D2	SYBASE — dbsetmaxprocs	SYBASE — операция dbsetmaxprocs
4819	12D3	SYBASE — dbsetopt	SYBASE — операция dbsetopt
4820	12D4	SYBASE — dbsettime	SYBASE — операция dbsettime
4821	12D5	SYBASE — dbsqlexec	SYBASE — операция dbsqlexec
4822	12D6	SYBASE — dbsqlok	SYBASE — операция dbsqlok
4823	12D7	SYBASE — dbsqlsend	SYBASE — операция dbsqlsend
4824	12D8	SYBASE — dbtxptr	SYBASE — операция dbtxptr
4825	12D9	SYBASE — dbtxtimestamp	SYBASE — операция dbtxtimestamp
4826	12DA	SYBASE — dbtxtsnewval	SYBASE — операция dbtxtsnewval
4827	12DB	SYBASE — dbuse	SYBASE — операция dbuse
4828	12DC	SYBASE — dbwinexit	SYBASE — операция dbwinexit
4829	12DD	SYBASE — dbwritetext	SYBASE — операция dbwritetext
4830	12DE	SYBASE — dbcount	SYBASE — операция dbcount

Продолжение табл. Б.1

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
4831	12DF	SYBASE — dbdead	SYBASE — операция dbdead
4942	134E	Unmapped SQL error code	Неотображенная ошибка SQL
8449	2101	Cannot open a system file	Невозможно открыть системный файл
8450	2102	I/O error on a system file	Ошибка ввода/вывода при работе с системным файлом
8451	2103	Data structure corruption	Повреждение структуры данных
8452	2104	Cannot find engine configuration file	Невозможно найти конфигурационный файл BDE
8453	2105	Cannot write to engine configuration file	Невозможно записать в конфигурационный файл BDE
8454	2106	Cannot initialize with different configuration file	Невозможно инициализировать с другим конфигурационным файлом
8455	2107	System has been illegally reentered	Некорректный повторный вход в систему
8456	2108	Cannot locate IDAPI32.DLL	Невозможно найти библиотеку IDAPI32.DLL
8457	2109	Cannot load IDAPI32.DLL	Невозможно загрузить библиотеку IDAPI32.DLL
8458	210A	Cannot load an IDAPI service library	Невозможно загрузить библиотеку IDAPI
8459	210B	Cannot create or open temporary file	Невозможно создать или открыть временный файл
8705	2201	At beginning of table	В начале таблицы
8706	2202	At end of table	В конце таблицы
8707	2203	Record moved because key value changed	Запись перемещена из-за изменения значения ключа
8708	2204	Record/key deleted	Запись или ключ удалены
8709	2205	No current record	Нет текущей записи
8710	2206	Could not find record	Невозможно найти запись
8711	2207	End of BLOB	Конец BLOB
8712	2208	Could not find object	Невозможно найти объект
8713	2209	Could not find family member	Невозможно найти члена семейства
8714	220A	BLOB file is missing	BLOB-файл отсутствует
8715	220B	Could not find language driver	Невозможно найти драйвер языка
8961	2301	Corrupt table/index header	Повреждение заголовка таблицы/индекса
8962	2302	Corrupt file — other than header	Поврежденный файл не соответствует заголовку
8963	2303	Corrupt memo/BLOB file	Повреждение memo/BLOB-файла
8965	2305	Corrupt index	Повреждение индекса

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
8966	2306	Corrupt lock file	Повреждение файла блокировок
8967	2307	Corrupt family file	Повреждение файла семейства
8968	2308	Corrupt or missing .VAL file	Повреждение или отсутствие .VAL-файла
8969	2309	Foreign index file format	Файл формата внешнего индекса
9217	2401	Read failure	Ошибка чтения
9218	2402	Write failure	Ошибка записи
9219	2403	Cannot access directory	Ошибка доступа к каталогу
9220	2404	File Delete operation failed	Ошибка удаления файла
9221	2405	Cannot access file	Доступ к файлу невозможен
9222	2406	Access to table disabled because of previous error	Доступ к таблице запрещен из-за предыдущей ошибки
9473	2501	Insufficient memory for this operation	Недостаточно памяти для выполнения этой операции
9474	2502	Not enough file handles	Недостаточно дескрипторов файлов
9475	2503	Insufficient disk space	Недостаточно дискового пространства
9476	2504	Temporary table resource limit	Предел ресурса временной таблицы
9477	2505	Record size is too big for table	Размер записи слишком велик для таблицы
9478	2506	Too many open cursors	Слишком много открытых курсоров
9479	2507	Table is full	Таблица заполнена
9480	2508	Too many sessions from this workstation	Слишком много сеансов открыто на этой рабочей станции
9481	2509	Serial number limit (Paradox)	Ограничение серийного номера (Paradox)
9482	250A	Some internal limit (see context)	Некоторое внутреннее ограничение (зависит от контекста)
9483	250B	Too many open tables	Слишком много открытых таблиц
9484	250C	Too many cursors per table	Слишком много курсоров на одну таблицу
9485	250D	Too many record locks on table	Слишком много блокированных записей в таблице
9486	250E	Too many clients	Слишком много клиентов
9487	250F	Too many indexes on table	Слишком много индексов в таблице
9488	2510	Too many sessions	Слишком много сеансов
9489	2511	Too many open databases	Слишком много открытых баз данных
9490	2512	Too many passwords	Слишком много паролей
9491	2513	Too many active drivers	Слишком много активных драйверов
9492	2514	Too many fields in Table Create	Слишком много полей в создаваемой таблице

Продолжение табл. Б.1

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
9493	2515	Too many table locks	Слишком много блокировок таблицы
9494	2516	Too many open BLOBs	Слишком много открытых BLOB
9495	2517	Lock file has grown too large	Размер блокированного файла слишком возрос
9496	2518	Too many open queries	Слишком много открытых запросов
9498	251A	Too many BLOBs	Слишком много BLOB
9499	251B	Filename is too long for a Paradox version 5.0 table	Имя файла слишком велико для таблицы Paradox версии 5.0
9500	251C	Row fetch limit exceeded	Превышено предельно допустимое количество выбираемых строк
9501	251D	Long name not allowed for this table level	Длинные имена не разрешены на этом уровне таблицы
9729	2601	Key violation	Нарушение ключа
9730	2602	Minimum validity check failed	Ошибка минимальной проверки корректности
9731	2603	Maximum validity check failed	Ошибка максимальной проверки корректности
9732	2604	Field value required	Требуется значение поля
9733	2605	Master record missing	Отсутствует главная запись
9734	2606	Master has detail records. Cannot delete or modify	Для главной записи имеются детальные записи. Удаление главной записи невозможно
9735	2607	Master table level is incorrect	Уровень главной таблицы некорректен
9736	2608	Field value out of lookup table range	Значение поля за пределами ключа справочной таблицы
9737	2609	Lookup Table Open operation failed	Ошибка открытия справочной таблицы
9738	260A	Detail Table Open operation failed	Ошибка открытия детальной таблицы
9739	260B	Master Table Open operation failed	Ошибка открытия главной таблицы
9740	260C	Field is blank	Поле пусто
9741	260D	Link to master table already defined	Связь с главной таблицей уже определена
9742	260E	Master table is open	Главная таблица открыта
9743	260F	Detail table(s) exist	Существуют детальные таблицы
9744	2610	Master has detail records. Cannot empty it	У главной записи имеются детальные. Сделать ее пустой невозможно
9745	2611	Self referencing referential integrity must be entered one at a time with no other changes to the table	Ссылка на себя должна быть введена сразу, без других изменений в таблице
9746	2612	Detail table is open	Открыта детальная таблица

Продолжение табл. Б.1

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
9747	2613	Cannot make this master a detail of another table if its details are not empty	Главную таблицу нельзя сделать детальной таблицей другой таблицы, если ее детальные таблицы не пусты
9748	2614	Referential integrity fields must be indexed	Поля с контролем ссылочной целостности должны быть индексированы
9749	2615	A table linked by referential integrity requires password to open	Для открытия таблицы, связанной с контролем ссылочной целостности требуется пароль
9750	2616	Field(s) linked to more than one master	Поля связаны более чем с одной главной таблицей
9985	2701	Number is out of range	Число за пределами диапазона
9986	2702	Invalid parameter	Неверный параметр
9987	2703	Invalid file name	Неверное имя файла
9988	2704	File does not exist	Файла не существует
9989	2705	Invalid option	Неверный параметр
9990	2706	Invalid handle to the function	Неверный дескриптор функции
9991	2707	Unknown table type	Неизвестный тип таблицы
9992	2708	Cannot open file	Нельзя открыть файл
9993	2709	Cannot redefine primary key	Переопределить первичный ключ невозможно
9994	270A	Cannot change this `RINTDesc`	Невозможно изменить данный `RINTDesc`
9995	270B	Foreign and primary key do not match	Внешний и первичный ключ не совпадают
9996	270C	Invalid modify request	Неверный запрос на изменение
9997	270D	Index does not exist	Индекса не существует
9998	270E	Invalid offset into the BLOB	Неверное смещение в BLOB
9999	270F	Invalid descriptor number	Неверный номер дескриптора
10000	2710	Invalid field type	Неверный тип поля
10001	2711	Invalid field descriptor	Неверный дескриптор поля
10002	2712	Invalid field transformation	Неверное преобразование поля
10003	2713	Invalid record structure	Неверная структура записи
10004	2714	Invalid descriptor	Неверный дескриптор
10005	2715	Invalid array of index descriptors	Некорректный массив дескрипторов индексов
10006	2716	Invalid array of validity check descriptors	Некорректный массив дескрипторов проверки
10007	2717	Invalid array of referential integrity descriptors	Некорректный массив дескрипторов ссылочной целостности

Продолжение табл. Б.1

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
10008	2718	Invalid ordering of tables during restructure	Неверный порядок таблиц во время реструктуризации
10009	2719	Name not unique in this context	В данном контексте имя не является уникальным
10010	271A	Index name required	Требуется имя индекса
10011	271B	Invalid session handle	Некорректный дескриптор сеанса
10012	271C	Invalid restructure operation	Некорректная операция реструктуризации
10013	271D	Driver not known to system	Драйвер неизвестен системе
10014	271E	Unknown database	Неизвестная база данных
10015	271F	Invalid password given	Дан некорректный пароль
10016	2720	No callback function	Нет функции обратного вызова
10017	2721	Invalid callback buffer length	Некорректная длина буфера обратного вызова
10018	2722	Invalid directory	Неверный каталог
10019	2723	Translate Error. Value out of bounds	Ошибка трансляции: значение за пределами диапазона
10020	2724	Cannot set cursor of one table to another	Невозможно установить курсор одной таблицы в другой
10021	2725	Bookmarks do not match table	Закладка не соответствует таблице
10022	2726	Invalid index/tag name	Некорректное имя индекса/дескриптора
10023	2727	Invalid index descriptor	Некорректный дескриптор индекса
10024	2728	Table does not exist	Таблицы не существует
10025	2729	Table has too many users	У таблицы слишком много пользователей
10026	272A	Cannot evaluate Key or Key does not pass filter condition	Вычислить ключ невозможно, или ключ не удовлетворяет условиям фильтра
10027	272B	Index already exists	Индекс уже существует
10028	272C	Index is open	Индекс открыт
10029	272D	Invalid BLOB length	Некорректная длина BLOB
10030	272E	Invalid BLOB handle in record buffer	В буфере записи некорректный дескриптор BLOB
10031	272F	Table is open	Таблица открыта
10032	2730	Need to do (hard) restructure	Требуется реструктуризация
10033	2731	Invalid mode	Некорректный режим
10034	2732	Cannot close index	Невозможно закрыть индекс
10035	2733	Index is being used to order table	Индекс используется таблицей
10036	2734	Unknown user name or password	Неизвестное имя пользователя или пароль
10037	2735	Multi-level cascade is not supported	Многоуровневое каскадирование не поддерживается
10038	2736	Invalid field name	Неверное имя поля

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
10039	2737	Invalid table name	Неверное имя таблицы
10040	2738	Invalid linked cursor expression	Неверное выражение связанного курсора
10041	2739	Name is reserved	Имя зарезервировано
10042	273A	Invalid file extension	Неверное расширение файла
10043	273B	Invalid language Driver	Неверный драйвер языка
10044	273C	Alias is not currently opened	Псевдоним в настоящее время не открыт
10045	273D	Incompatible record structures	Несовместимые структуры записей
10046	273E	Name is reserved by DOS	Имя зарезервировано DOS
10047	273F	Destination must be indexed	Целевой объект должен быть индексирован
10048	2740	Invalid index type	Неверный тип индекса
10049	2741	Language drivers of table and index do not match	Драйверы языка таблицы и индекса не совпадают
10050	2742	Filter handle is invalid	Некорректный дескриптор фильтра
10051	2743	Invalid filter	Некорректный фильтр
10052	2744	Invalid Table Create request	Неверный запрос создания таблицы
10053	2745	Invalid Table Delete request	Неверный запрос удаления таблицы
10054	2746	Invalid Index Create request	Неверный запрос создания индекса
10055	2747	Invalid Index Delete request	Неверный запрос удаления индекса
10056	2748	Invalid table specified	Определена некорректная таблица
10058	274A	Invalid time	Неверное время
10059	274B	Invalid date	Неверная дата
10060	274C	Invalid date/time	Неверные дата и/или время
10061	274D	Tables in different directories	Таблицы в разных каталогах
10062	274E	Mismatch in the number of arguments	Несоответствие в количестве аргументов
10063	274F	Function not found in service library	Функция не найдена в библиотеке
10064	2750	Must use `baseorder` for this operation	Для этой операции необходимо использовать `baseorder`
10065	2751	Invalid procedure name	Неверное имя процедуры
10066	2752	The field map is invalid	Карта поля неверна
10241	2801	Record locked by another user	Запись блокирована другим пользователем
10242	2802	Unlock failed	Разблокирование не выполнено
10243	2803	Table is busy	Таблица занята
10244	2804	Directory is busy	Каталог занят
10245	2805	File is locked	Файл блокирован
10246	2806	Directory is locked	Каталог блокирован

Продолжение табл. Б.1

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
10247	2807	Record already locked by this session	В этом сеансе запись уже заблокирована
10248	2808	Object not locked	Объект не заблокирован
10249	2809	Lock time out	Тайм-аут блокировки
10250	280A	Key group is locked	Группа ключей блокирована
10251	280B	Table lock was lost	Потеря блокировки таблицы
10252	280C	Exclusive access was lost	Исключительный доступ потерян
10253	280D	Table cannot be opened for exclusive use	Таблица не может быть открыта для исключительного использования
10254	280E	Conflicting record lock in this session	Конфликтная запись в этом сеансе блокирована
10255	280F	A deadlock was detected	Обнаружен тупик
10256	2810	A user transaction is already in progress	Пользовательская транзакция в процессе выполнения
10257	2811	No user transaction is currently in progress	В настоящее время пользовательские транзакции не выполняются
10258	2812	Record lock failed	Ошибка блокировки поля
10259	2813	Couldn't perform the edit because another user changed the record	Редактирование выполнить невозможно из-за изменения записи другим пользователем
10260	2814	Couldn't perform the edit because another user deleted or moved the record	Редактирование выполнить невозможно из-за удаления или перемещения записи другим пользователем
10497	2901	Insufficient field rights for operation	Для выполнения операции недостаточно прав поля
10498	2902	Insufficient table rights for operation. Password required	Для выполнения операции недостаточно прав таблицы. Требуется пароль
10499	2903	Insufficient family rights for operation	Для выполнения операции недостаточно прав семейства
10500	2904	This directory is read-only	Каталог “только для чтения”
10501	2905	Database is read-only	База данных “только для чтения”
10502	2906	Trying to modify read-only field	Попытка изменения поля “только для чтения”
10503	2907	Encrypted dBASE tables not supported	Зашифрованные таблицы dBASE не поддерживаются
10504	2908	Insufficient SQL rights for operation	Для выполнения операции недостаточно прав SQL
10753	2A01	Field is not a BLOB	Поле не является BLOB
10754	2A02	BLOB already opened	BLOB уже открыт
10755	2A03	BLOB not opened	BLOB не открыт

Код ошибки		Строка сообщения	
Десятичный	**Шестнадцатеричный**	**Оригинал**	**Перевод**
10756	2A04	Operation not applicable	Операция не применима
10757	2A05	Table is not indexed	Таблица не индексирована
10758	2A06	Engine not initialized	Система не инициализирована
10759	2A07	Attempt to re-initialize Engine	Попытка повторной инициализации системы
10760	2A08	Attempt to mix objects from different sessions	Попытка смешивания объектов разных сеансов
10761	2A09	Paradox driver not active	Драйвер Paradox неактивен
10762	2A0A	Driver not loaded	Драйвер не загружен
10763	2A0B	Table is read-only	Таблица “только для чтения”
10764	2A0C	No associated index	Нет ассоциированного индекса
10765	2A0D	Table(s) open. Cannot perform this operation	Таблица открыта. Выполнение операции невозможно
10766	2A0E	Table does not support this operation	Эту операцию таблица не поддерживает
10767	2A0F	Index is read-only	Индекс “только для чтения”
10768	2A10	Table does not support this operation because it is not uniquely indexed	Таблица не поддерживает операцию, так как не индексирована
10769	2A11	Operation must be performed on the current session	Операция должна быть выполнена в текущем сеансе
10770	2A12	Invalid use of keyword	Неверное использование ключевого слова
10771	2A13	Connection is in use by another statement	Соединение используется другим оператором
10772	2A14	Pass-through SQL connection must be shared	Проходящее соединение SQL должно быть совместно используемым
11009	2B01	Invalid function number	Неверный номер функции
11010	2B02	File or directory does not exist	Файла или каталога не существует
11011	2B03	Path not found	Путь не найден
11012	2B04	Too many open files. You may need to increase MAXFILEHANDLE limit in IDAPI configuration	Слишком много открытых файлов. В конфигурации IDAPI следует увеличить значение MAXFILEHANDLE
11013	2B05	Permission denied	Доступ закрыт
11014	2B06	Bad file number	Неверный номер файла
11015	2B07	Memory blocks destroyed	Блоки памяти разрушены
11016	2B08	Not enough memory	Недостаточно памяти
11017	2B09	Invalid memory block address	Неверный адрес блока памяти
11018	2B0A	Invalid environment	Неверная среда
11019	2B0B	Invalid format	Неверный формат

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
11020	2B0C	Invalid access code	Неверный код доступа
11021	2B0D	Invalid data	Неверные данные
11023	2B0F	Device does not exist	Устройства не существует
11024	2B10	Attempt to remove current directory	Попытка удаления текущего каталога
11025	2B11	Not same device	Другое устройство
11026	2B12	No more files	Больше нет файлов
11027	2B13	Invalid argument	Неверный аргумент
11028	2B14	Argument list is too long	Слишком много аргументов
11029	2B15	Execution format error	Ошибка формата выполнения
11030	2B16	Cross-device link	Связь перекрестных устройств
11041	2B21	Math argument	Математический аргумент
11042	2B22	Result is too large	Слишком велико значение результата
11043	2B23	File already exists	Файл уже существует
11047	2B27	Unknown internal operating system error	Неизвестная внутренняя системная ошибка
11058	2B32	Share violation	Нарушение совместного использования
11059	2B33	Lock violation	Нарушение блокировки
11060	2B34	Critical DOS Error	Критическая ошибка DOS
11061	2B35	Drive not ready	Устройство не готово
11108	2B64	Not exact read/write	Неточное чтение/запись
11109	2B65	Operating system network error	Сетевая ошибка операционной системы
11110	2B66	Error from NOVELL file server	Ошибка файлового сервера NOVELL
11111	2B67	NOVELL server out of memory	Нехватка памяти на сервере NOVELL
11112	2B68	Record already locked by this workstation	Запись уже блокирована рабочей станцией
11113	2B69	Record not locked	Запись не заблокирована
11265	2C01	Network initialization failed	Ошибка инициализации сети
11266	2C02	Network user limit exceeded	Превышен предел количества пользователей сети
11267	2C03	Wrong NET file version	Некорректная версия .NET-файла
11268	2C04	Cannot lock network file	Невозможно блокировать сетевой файл
11269	2C05	Directory is not private	Каталог не является закрытым
11270	2C06	Directory is controlled by other NET file	Каталог контролируется другим .NET-файлом
11271	2C07	Unknown network error	Неизвестная сетевая ошибка
11272	2C08	Not initialized for accessing network files	Сетевые файлы доступа не инициализированы

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
11273	2C09	SHARE not loaded. It is required to share local files	Программа SHARE не загружена. Требуется для совместного использования локальных файлов
11274	2C0A	Not on a network. Not logged in or wrong network driver	Не в сети. Не зарегистрированы в сети или некорректный сетевой драйвер
11275	2C0B	Lost communication with SQL server	Потеря соединения с SQL-сервером
11277	2C0D	Cannot locate or connect to SQL server	Обнаружение или соединение с SQL-сервером невозможно
11278	2C0E	Cannot locate or connect to network server	Обнаружение или соединение с сетевым сервером невозможно
11521	2D01	Optional parameter is required	Требуется необязательный параметр
11522	2D02	Invalid optional parameter	Некорректный необязательный параметр
11777	2E01	*Obsolete*	*Устарело*
11778	2E02	*Obsolete*	*Устарело*
11779	2E03	Ambiguous use of ! (inclusion operator)	Неоднозначное использование оператора !
11780	2E04	*Obsolete*	*Устарело*
11781	2E05	*Obsolete*	*Устарело*
11782	2E06	A SET operation cannot be included in its own grouping	Операция SET не может быть включена в свою собственную группировку
11783	2E07	Only numeric and date/time fields can be averaged	Усреднены могут быть только поля даты/времени и числовые
11784	2E08	Invalid expression	Некорректное выражение
11785	2E09	Invalid OR expression	Некорректное выражение OR
11786	2E0A	*Obsolete*	*Устарело*
11787	2E0B	Bitmap	Растровое изображение
11788	2E0C	CALC expression cannot be used in INSERT, DELETE, CHANGETO, and SET rows	Выражение CALC не может использоваться строками INSERT, DELETE, CHANGETO и SET
11789	2E0D	Type error in CALC expression	Ошибка типа в выражении CALC
11790	2E0E	CHANGETO can be used in only one query form at a time	Выражение CHANGETO одновременно может использоваться только в одной форме запроса
11791	2E0F	Cannot modify CHANGED table	Изменение таблицы GHANGED невозможно
11792	2E10	A field can contain only one CHANGETO expression	Поле может содержать только одно выражение GHANGETO
11793	2E11	A field cannot contain more than one expression to be inserted	Поле не может содержать более одного вставляемого выражения

Код ошибки		Строка сообщения	
Десятичный	**Шестнадцатеричный**	**Оригинал**	**Перевод**
11794	2E12	*Obsolete*	*Устарело*
11795	2E13	CHANGETO must be followed by the new value for the field	За выражением CHANGETO должно следовать новое значение поля
11796	2E14	Checkmark or CALC expressions cannot be used in FIND queries	Контрольные отметки и выражения CALC не могут использоваться в запросах FIND
11797	2E15	Cannot perform operation on CHANGED table together with a CHANGETO query	Выполнение операции в таблице CHANGED совместно с запросом CHANGETO невозможно
11798	2E16	Chunk	Разделение
11799	2E17	More than 255 fields in ANSWER table	В таблице ANSWER более 255 полей
11800	2E18	AS must be followed by the name for the field in the ANSWER table	В таблице ANSWER после AS должно следовать имя
11801	2E19	DELETE can be used in only one query form at a time	DELETE одновременно может использоваться только в одной форме запроса
11802	2E1A	Cannot perform operation on DELETED table together with a DELETE query	Выполнение операции с таблицей DELETED вместе с запросом DELETE невозможно
11803	2E1B	Cannot delete from the DELETED table	Удаление из таблицы DELETED невозможно
11804	2E1C	Example element is used in two fields with incompatible types or with a BLOB	Образец используется в двух полях с несовместимыми типами или с BLOB
11805	2E1D	Cannot use example elements in an OR expression	Использование образца в выражении OR невозможно
11806	2E1E	Expression in this field has the wrong type	Выражение в этом поле имеет неверный тип
11807	2E1F	Extra comma found	Найдена лишняя запятая
11808	2E20	Extra OR found	Найдено лишнее выражение OR
11809	2E21	One or more query rows do not contribute to the ANSWER	Одна или более строк запроса не входят в ANSWER
11810	2E22	FIND can be used in only one query form at a time	FIND одновременно может использоваться только в одной форме запроса
11811	2E23	FIND cannot be used with the ANSWER table	FIND не может использоваться с таблицей ANSWER
11812	2E24	A row with GROUPBY must contain SET operations	Строка с GROUPBY должна содержать операцию SET
11813	2E25	GROUPBY can be used only in SET rows	GROUPBY может использоваться только в строках SET

Продолжение табл. Б.1

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
11814	2E26	Use only INSERT, DELETE, SET, or FIND in leftmost column	В левом столбце могут использоваться только INSERT, DELETE, SET или FIND
11815	2E27	Use only one INSERT, DELETE, SET, or FIND per line	В строке может находиться только один оператор INSERT, DELETE, SET или FIND
11816	2E28	Syntax error in expression	Синтаксическая ошибка в выражении
11817	2E29	INSERT can be used in only one query form at a time	INSERT одновременно может использоваться только в одной форме запроса
11818	2E2A	Cannot perform operation on INSERTED table together with an INSERT query	Выполнение запроса INSERT вместе с операцией с таблицей INSERTED невозможно
11819	2E2B	INSERT, DELETE, CHANGETO, and SET rows may not be checked	Строки INSERT, DELETE, CHANGETO и SET могут быть не проверены
11820	2E2C	Field must contain an expression to insert (or be blank)	Поле должно содержать выражение для вставки (или быть пустым)
11821	2E2D	Cannot insert into the INSERTED table	Вставка в таблицу INSERTED невозможна
11822	2E2E	Variable is an array and cannot be accessed	Переменная представляет собой массив и не может быть доступна
11823	2E2F	Label	Метка
11824	2E30	Rows of example elements in CALC expression must be linked	Строки образцов в выражении CALC должны быть связаны
11825	2E31	Variable name is too long	Слишком длинное имя переменной
11826	2E32	Query may take a long time to process	Запрос может потребовать длительной обработки
11827	2E33	Reserved word or one that can't be used as a variable name	Зарезервированное слово или другой символ не может использоваться в качестве имени переменной
11828	2E34	Missing comma	Отсутствует запятая
11829	2E35	Missing right parenthesis	Отсутствует закрывающая скобка
11830	2E36	Missing right quote	Отсутствует правая кавычка
11831	2E37	Cannot specify duplicate column names	Недопустимо наличие столбцов с одинаковыми именами
11832	2E38	Query has no checked fields	Запрос не содержит проверяемых полей
11833	2E39	Example element has no defining occurrence	Образец не имеет определения
11834	2E3A	No grouping is defined for SET operation	Для операции SET не определена группировка
11835	2E3B	Query makes no sense	Бессмысленный запрос
11836	2E3C	Cannot use patterns in this context	В этом контексте нельзя использовать шаблоны

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
11837	2E3D	Date does not exist	Даты не существует
11838	2E3E	Variable has not been assigned a value	Переменной не присвоено значение
11839	2E3F	Invalid use of example element in summary expression	Некорректное использование образца в итоговом выражении
11840	2E40	Incomplete query statement. Query only contains a SET definition	Некорректный запрос. Запрос содержит только определения SET
11841	2E41	Example element with ! makes no sense in expression	Образец с ! делает выражение бессмысленным
11842	2E42	Example element cannot be used more than twice with a ! query	В запросе ! образец не может использоваться более двух раз
11843	2E43	Row cannot contain expression	Строка не может содержать выражение
11844	2E44	*Obsolete*	*Устарело*
11845	2E45	*Obsolete*	*Устарело*
11846	2E46	No permission to insert or delete records	Нет прав доступа на вставку или удаление записей
11847	2E47	No permission to modify field	Нет прав доступа на изменение поля
11848	2E48	Field not found in table	Поле в таблице не найдено
11849	2E49	Expecting a column separator in table header	В заголовке таблицы ожидается разделитель столбцов
11850	2E4A	Expecting a column separator in table	В таблице ожидается разделитель столбцов
11851	2E4B	Expecting column name in table	В таблице ожидается имя столбца
11852	2E4C	Expecting table name	Ожидается имя таблицы
11853	2E4D	Expecting consistent number of columns in all rows of table	Во всех строках таблицы должно быть согласованное количество полей
11854	2E4E	Cannot open table	Невозможно открыть таблицу
11855	2E4F	Field appears more than once in table	Поле встречается в таблице несколько раз
11856	2E50	This DELETE, CHANGE, or INSERT query has no ANSWER	Запрос DELETE, CHANGE или INSERT не имеет ответа
11857	2E51	Query is not prepared. Properties unknown	Запрос не подготовлен. Свойства неизвестны
11858	2E52	DELETE rows cannot contain quantifier expression	Строки DELETE не могут содержать числовых выражений
11859	2E53	Invalid expression in INSERT row	Некорректное выражение в строке INSERT
11860	2E54	Invalid expression in INSERT row	Некорректное выражение в строке INSERT
11861	2E55	Invalid expression in SET definition	Некорректное выражение в определении SET

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
11862	2E56	Row use	Строка используется
11863	2E57	SET keyword expected	Ожидается ключевое слово SET
11864	2E58	Ambiguous use of example element	Неоднозначность в использовании образца
11865	2E59	*Obsolete*	*Устарело*
11866	2E5A	*Obsolete*	*Устарело*
11867	2E5B	Only numeric fields can be summed	Суммироваться могут только числовые поля
11868	2E5C	Table is write protected	Таблица защищена от записи
11869	2E5D	Token not found	Лексема не найдена
11870	2E5E	Cannot use example element with ! more than once in a single row	Образец с ! нельзя использовать более одного раза в одной строке
11871	2E5F	Type mismatch in expression	Несоответствие типа в выражении
11872	2E60	Query appears to ask two unrelated questions	В запросе содержится два несвязанных вопроса
11873	2E61	Unused SET row	Неиспользуемая строка SET
11874	2E62	INSERT, DELETE, FIND, and SET can be used only in the left-most column	INSERT, DELETE, FIND и SET могут использоваться только в самом левом столбце
11875	2E63	CHANGETO cannot be used with INSERT, DELETE, SET, or FIND	CHANGETO не может использоваться вместе с INSERT, DELETE, SET или FIND
11876	2E64	Expression must be followed by an example element defined in a SET	За выражением должен следовать образец, определенный в SET
11877	2E65	Lock failure	Ошибка блокировки
11878	2E66	Expression is too long	Слишком длинное выражение
11879	2E67	Refresh exception during query	Исключительная ситуация обновления во время запроса
11880	2E68	Query canceled	Запрос отменен
11881	2E69	Unexpected database engine error	Неожиданная ошибка Database Engine
11882	2E6A	Not enough memory to finish operation	Недостаточно памяти для завершения операции
11883	2E6B	Unexpected exception	Неожиданное исключение
11884	2E6C	Feature not implemented yet in query	Запрошенные в запросе функции еще не реализованы
11885	2E6D	Query format is not supported	Формат запроса не поддерживается
11886	2E6E	Query string is empty	Пустая строка запроса
11887	2E6F	Attempted to prepare an empty query	Попытка подготовки пустого запроса
11888	2E70	Buffer too small to contain query string	Буфер слишком мал для содержания строки запроса

Продолжение табл. Б.1

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
11889	2E71	Query was not previously parsed or prepared	Запрос не был предварительно разобран или подготовлен
11890	2E72	Function called with bad query handle	Функция вызвана с некорректным дескриптором запроса
11891	2E73	QBE syntax error	Синтаксическая ошибка QBE
11892	2E74	Query extended syntax field count error	Расширенная синтаксическая ошибка количества полей
11893	2E75	Field name in sort or field clause not found	В операторе сортировки или поиска не найдено имя поля
11894	2E76	Table name in sort or field clause not found	В операторе сортировки или поиска не найдено имя таблицы
11895	2E77	Operation is not supported on BLOB fields	Операция над полями BLOB не поддерживается
11896	2E78	General BLOB error	Общая ошибка BLOB
11897	2E79	Query must be restarted	Запрос должен быть запущен повторно
11898	2E7A	Unknown answer table type	Неизвестный тип таблицы ответа
11926	2E96	Blob cannot be used as grouping field	BLOB не может использоваться в качестве поля группировки
11927	2E97	Query properties have not been fetched	Не были извлечены свойства запроса
11928	2E98	Answer table is of unsuitable type	Таблица ответа неподходящего типа
11929	2E99	Answer table is not yet supported under server alias	Таблица ответа под псевдонимом сервера не поддерживается
11930	2E9A	Non-null blob field required. Can't insert records	Требуется ненулевое поле BLOB. Вставка записей невозможна
11931	2E9B	Unique index required to perform CHANGETO	Для выполнения CHANGETO требуется уникальный индекс
11932	2E9C	Unique index required to delete records	Для удаления требуется уникальный индекс
11933	2E9D	Update of table on the server failed	Ошибка обновления таблицы на сервере
11934	2E9E	Can't process this query remotely	Удаленная обработка этого запроса невозможна
11935	2E9F	Unexpected end of command	Неожиданный конец команды
11936	2EA0	Parameter not set in query string	В строке не указаны параметры запроса
11937	2EA1	Query string is too long	Слишком длинная строка запроса
11946	2EAA	No such table or correlation name	Такой таблицы или соответствующего имени не существует
11947	2EAB	Expression has ambiguous data type	Выражение содержит неоднозначные типы данных

Продолжение табл. Б.1

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
11948	2EAC	Field in ORDER BY must be in result set	В результирующем наборе поля должны быть отсортированы
11949	2EAD	General parsing error	Общая синтаксическая ошибка
11950	2EAE	Record or field constraint failed	Ошибка ограничения записи или поля
11951	2EAF	Field in GROUP BY must be in result set	Поля в результирующем наборе должны быть сгруппированы
11952	2EB0	User-defined function is not defined	Пользовательская функция не определена
11953	2EB1	Unknown error from user-defined function	Неизвестная ошибка в пользовательской функции
11954	2EB2	Single-row subquery produced more than one row	Однострочным подзапросом сгенерировано более одной строки
11955	2EB3	Expressions in GROUP BY are not supported	В операторе GROUP BY выражения не поддерживаются
11956	2EB4	Queries on text or ASCII tables are not supported	Запрос в тексте или ASCII-таблицах не поддерживается
11957	2EB5	ANSI join keywords USING and NATURAL are not supported in this release	Предусмотренные стандартом ANSI ключевые слова USING и NATURAL в данной версии не поддерживается
11958	2EB6	SELECT DISTINCT may not be used with UNION unless UNION ALL is used	SELECT DISTINCT не может использоваться вместе с UNION до тех пор, пока применяется UNION ALL
11959	2EB7	GROUP BY is required when both aggregate and non-aggregate fields are used in result set	При использовании в результирующем наборе и агрегированных, и неагрегированных данных требуется фраза GROUP BY
11960	2EB8	INSERT and UPDATE operations are not supported on auto-increment field type	Операции INSERT и UPDATE в автоинкрементных полях не поддерживаются
11961	2EB9	UPDATE on primary key of a master table may modify more than one record	Операция UPDATE над первичным ключом главной таблицы может изменить несколько записей
12033	2F01	Interface mismatch. Engine version different	Несоответствие интерфейса. Различные версии системы
12034	2F02	Index is out of date	Индекс за пределами диапазона даты
12035	2F03	Older version (see context)	Старая версия
12036	2F04	.VAL file is out of date	.VAL-файл за пределами диапазона даты
12037	2F05	BLOB file version is too old	Слишком старая версия BLOB-файла
12038	2F06	Query and engine DLLs are mismatched	Запрос и библиотеки DLL ядра не соответствуют друг другу
12039	2F07	Server is incompatible version	Несовместимая версия сервера
12040	2F08	Higher table level required	Требуется более высокий уровень таблицы

Продолжение табл. Б.1

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
12289	3001	Capability not supported	Совместимость не поддерживается
12290	3002	Not implemented yet	Еще не реализовано
12291	3003	SQL replicas not supported	Реплики SQL не поддерживаются
12292	3004	Non-BLOB column in table required to perform operation	Для выполнения операции требуется не BLOB-столбец таблицы
12293	3005	Multiple connections not supported	Множественные соединения не поддерживаются
12294	3006	Full dBASE expressions not supported	Полный набор выражения dBASE не поддерживаются
12545	3101	Invalid database alias specification	Неверная спецификация псевдонима базы данных
12546	3102	Unknown database type	Неизвестный тип базы данных
12547	3103	Corrupt system configuration file	Поврежден системный файл конфигурации
12548	3104	Network type unknown	Неизвестный тип сети
12549	3105	Not on the network	Не в сети
12550	3106	Invalid configuration parameter	Неверный параметр конфигурации
12801	3201	Object implicitly dropped	Объект неявно исключен
12802	3202	Object may be truncated	Объект может быть усечен
12803	3203	Object implicitly modified	Объект неявно изменен
12804	3204	Should field constraints be checked?	Должно ли проверяться ограничение поля?
12805	3205	Validity check field modified	Изменение проверки корректности поля
12806	3206	Table level changed	Изменение уровня таблицы
12807	3207	Copy linked tables?	Копировать связанные таблицы?
12809	3209	Object implicitly truncated	Объект неявно усечен
12810	320A	Validity check will not be enforced	Проверка корректности не будет выполнена
12811	320B	Multiple records found, but only one was expected	Найдено несколько записей, а не одна
12812	320C	Field will be trimmed. Cannot put master records into PROBLEM table	Поле будет усечено. В таблицу PROBLEM главные записи поместить невозможно
13057	3301	File already exists	Файл уже существует
13058	3302	BLOB has been modified	BLOB был изменен
13059	3303	General SQL error	Общая ошибка SQL
13060	3304	Table already exists	Таблица уже существует
13061	3305	Paradox 1.0 tables are not supported	Таблицы Paradox 1.0 не поддерживаются
13062	3306	Update aborted	Обновление прекращено

Продолжение табл. Б.1

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
13313	3401	Different sort order	Другой порядок сортировки
13314	3402	Directory in use by earlier version of Paradox	Каталог используется более ранней версией Paradox
13315	3403	Needs Paradox 3.5-compatible language driver	Требуется драйвер языка, совместимый с Paradox 3.5
13569	3501	Data Dictionary is corrupt	Словарь данных поврежден
13570	3502	Data Dictionary info BLOB corrupted	Информация BLOB, содержащаяся в словаре данных, повреждена
13571	3503	Data Dictionary schema is corrupt	Схема словаря данных повреждена
13572	3504	Attribute type exists	Тип атрибута уже существует
13573	3505	Invalid object type	Неверный тип объекта
13574	3506	Invalid relation type	Неверный тип отношения
13575	3507	View already exists	Представление уже существует
13576	3508	No such view exists	Такого представления не существует
13577	3509	Invalid record constraint	Неверное ограничение записи
13578	350A	Object is in a logical DB	Объект находится в логической базе данных
13579	350B	Dictionary already exists	Словарь уже существует
13580	350C	Dictionary does not exist	Словаря не существует
13581	350D	Dictionary database does not exist	База данных словаря не существует
13582	350E	Dictionary info is out of date. Needs refreshed	Информация словаря за пределами даты. Требуется обновление
13584	3510	Invalid dictionary name	Неверное имя словаря
13585	3511	Dependent objects exist	Существуют зависимые объекты
13586	3512	Too many relationships for this object type	Для этого типа объекта слишком много связей
13587	3513	Relationships to the object exist	Существуют связи с объектом
13588	3514	Dictionary exchange file is corrupt	Файл обмена словаря поврежден
13589	3515	Dictionary exchange file version mismatch	Несоответствие версии файла обмена словаря
13590	3516	Dictionary object type mismatch	Несоответствие типа объекта словаря
13591	3517	Object exists in target dictionary	Объект имеется в выходном словаре
13592	3518	Cannot access data dictionary	Доступ к словарю данных невозможен
13593	3519	Cannot create data dictionary	Создание словаря данных невозможно
13594	351A	Cannot open database	Открыть базу данных невозможно
15873	3E01	Wrong driver name	Неверное имя драйвера
15874	3E02	Wrong system version	Неверная версия системы
15875	3E03	Wrong driver version	Неверная версия драйвера

Окончание табл. Б.1

Код ошибки		Строка сообщения	
Десятичный	Шестнадцатеричный	Оригинал	Перевод
15876	3E04	Wrong driver type	Неверный тип драйвера
15877	3E05	Cannot load driver	Невозможно загрузить драйвер
15878	3E06	Cannot load language driver	Невозможно загрузить драйвер языка
15879	3E07	Vendor initialization failed	Ошибка инициализации поставщика
15880	3E08	Your application is not enabled for use with this driver	Ваше приложение не может использоваться с этим драйвером

Рекомендуемая литература

Приложение

В

Программирование на Delphi 964

Разработка компонентов 964

Программирование в Windows 964

Объектно-ориентированное программирование 964

Проектирование программного обеспечения и разработка пользовательского интерфейса 964

COM/ActiveX/OLE 965

Программирование на Delphi

- John Ayres, David Bowden, Larry Diehl, Phil Dorcas, Kenneth Harrison, Rod Mathes, Ovias Reza and Mike Tobin. *The Tomes of Delphi 3: Win32 Graphical API.* — Wordware Publishing, Inc., 1998.
- John Ayres, David Bowden, Larry Diehl, Phil Dorcas, Kenneth Harrison, Rod Mathes, Ovias Reza, and Mike Tobin. *The Tomes of Delphi 3: Win32 Core API.* — Wordware Publishing, Inc., 1997.
- Charlie Calvert. *Charlie Calvert's Delphi 4 Unleashed.* — Sams Publishing, 1998.
- Marco Cantu. *Mastering Delphi 5*.— Sybex, 1999.
- Marco Cantu, Tim Gooch, and John F. Lam. *Delphi Developer's Handbook.* — Sybex, 1997.
- Ray Lischner. *Hidden Paths of Delphi 3*. — Informant Communications Group, 1997.
- Ray Lischner. *Secrets of Delphi 2.* — Waite Group Press, 1996.

Разработка компонентов

- Ray Konopka. *Developing Custom Delphi 3 Components*. — Coriolis Group Books, 1997.
- Danny Thorpe. *Delphi Component Design*. — Addison-Wesley, 1997.

Программирование в Windows

- Jeffrey Richter. *Advanced Windows*, *3rd ed*. — Microsoft Press, 1997.

Объектно-ориентированное программирование

- Grady Booch. *Object-Oriented Analysis and Design with Applications, 2nd ed.* — Addison-Wesley, 1994.
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software* — Addison-Wesley, 1995.

Проектирование программного обеспечения и разработка пользовательского интерфейса

- Alan Cooper. *About Face: The Essentials of User Interface Design.* — IDG Books, 1995.
- Steve McConnell. *Rapid Development.* — Microsoft Press, 1996.

- Steve McConnell. *Software Project Survival Guide.* — Microsoft Press, 1998.
- Steve McConnell. *Code Complete.* — Microsoft Press, 1993.

COM/ActiveX/OLE

- Don Box. *Essential COM.* — Addison-Wesley, 1998.
- Kraig Brockschmidt. *Inside OLE, 2nd ed.* — Microsoft Press, 1995.

Приложение

Г

Описание содержимого прилагаемого компакт-диска

Что содержится на прилагаемом компакт-диске 967

Инструкции по установке программ 967

Описание продуктов от независимых поставщиков 967

Что содержится на прилагаемом компакт-диске

На прилагаемом ко второму тому компакт-диске содержится полный набор исходных текстов всех примеров программ и приложений, обсуждавшихся в различных главах этой книги. Здесь же имеются необходимые данные и ресурсы, нестандартные компоненты и утилиты.

Кроме того, в отдельном каталоге содержится некоторые программные продукты от независимых разработчиков, краткое описание каждого из которых будет дано чуть ниже.

Инструкции по установке программ

Вашему вниманию предлагаются краткие инструкции по установке поставляемого на компакт-диске программного обеспечения в среде Windows 95, Windows 98 и Windows NT 4.

1. Вставьте компакт-диск в устройство.
2. Дважды щелкните на пиктограмме My Computer (Мой компьютер), расположенной на рабочем столе Windows.
3. В окне My Computer дважды щелкните на пиктограмме устройства чтения компакт-дисков.
4. В раскрывшемся окне с содержимым компакт-диска дважды щелкните на пиктограмме `START.EXE`. Это вызовет запуск программы установки программного обеспечения, поставляемого на компакт-диске.
5. Для завершения процесса установки выполните все инструкции, выведенные программой установки на экран компьютера.

Описание продуктов от независимых поставщиков

1stClass

от компании Woll2Woll Software

Программный продукт *1stClass* представляет собой набор визуальных компонентов, разработанных специально для того, чтобы предоставить в распоряжение профессиональных разработчиков практически неограниченные возможности создания в среде Delphi и C++Builder приложений, которые никого не оставят равнодушным. В наборе компонентов содержатся превосходные формы и кнопки в виде графических изображений, мощные и изысканные иерархические элементы управления, как для работы с данными, так и общего назначения, а также управляющий элемент-контейнер Office 97 Outlook Bar Style. Познакомившись со всеми этими возможностями, вы в полной мере ощутите высокое качество библиотеки 1stClass.

С помощью элемента управления `Imager` к растровым изображениям можно применять различные графические спецэффекты. В группах переключателей и флажков растровое изображение можно использовать в виде фона. Кроме того, в распоряжение разработчика предоставляются интегрированные текстовые спецэффекты, а также разнообразные возможно-

сти создания строки состояния с многочисленными встроенными стилями и разделением на пропорциональные панели. Существует возможность применять достаточно сложные элементы управления, позволяющие выбрать шрифт, цвет и имеющие иерархическую структуру или графическое представление, которое может быть встроено непосредственно в сетку InfoPower. Не имеет значения, какой тип приложения вы создаете, — при использовании компонентов библиотеки 1stClass большой выигрыш будет достигнут в любом случае.

Для того чтобы приобрести, зарегистрировать или обновить этот программный продукт, можно посетить Web-страницу компании Woll2Woll Software по адресу `www.woll2woll.com`, отправить электронное сообщение по адресу `sales@woll2woll.com`, позвонить по телефону (925) 371-16-63 или отправить факс по номеру (925) 371-16-64.

Abbrevia (пробная версия)

от компании TurboPower Software

Abbrevia — высокопроизводительная библиотека, позволяющая применять промышленные стандарты сжатия данных в приложениях Delphi и C++Builder. С помощью этой библиотеки можно создавать и обрабатывать PKZIP-совместимые архивные файлы. Кроме сжатия и извлечения данных, библиотекой Abbrevia поддерживаются расширенные возможности утилиты сжатия PKZIP, например, такие как создание самораспаковывающихся архивных файлов, добавление к файлам и архивам комментариев, а также размещение больших архивных файлов на нескольких гибких дисках.

В комплект поставки библиотеки Abbrevia входит весь исходный текст (написанный в среде разработки Delphi), а также полный комплект печатной документации. Кроме того, примеры и файлы справочной системы имеются как в формате Delphi, так и в формате C++Builder. Библиотека совместима со всеми версиями Delphi и C++Builder компании Borland. При создании программ с использованием библиотеки Abbrevia не требуется никакой оплаты лицензии.

Если вы хотите приобрести, зарегистрировать или обновить этот программный продукт, посетите Web-страницу компании TurboPower Software по адресу `www.turbopower.com`, отправьте электронное сообщение по адресу `sales@turbopower.com` или позвоните по телефону 800/333-41-60 (за пределами США — по телефону 719/260-91-36).

Adobe Acrobat Reader 4.0

от компании Adobe Systems Incorporated

Adobe® Acrobat® Reader — это свободно распространяемый программный продукт, позволяющий просматривать и печатать файлы в формате PDF (Portable Document Format — Переносимый формат документов). Кроме того, с помощью Acrobat Reader можно заполнять и предоставлять PDF-формы, а также загрузить из Web зашифрованную информацию и разблокировать ее с помощью инструмента Web Buy.

Для загрузки обновленной версии Acrobat Reader или других программных продуктов компании Adobe посетите ее Web-страницу по адресу `www.adobe.com`.

Advantage Database Server 5.5 для NT или NetWare

от компании Extended Systems, Inc.

На компакт-диске продукт Advantage Database Server 5.5 представлен пробной версией для двух пользователей, работающих в среде Windows NT или NetWare, и, кроме того, он включает клиентский набор для Delphi. Advantage Database Server является масштабируемой, высокопроизводительной системой управления базами данных “клиент/сервер”, с помощью

которой можно создавать локальные, мобильные и сетевые приложения, а также программы для Web. С помощью компонентов, производных от класса TDataSet, предназначенных для использования совместно с компонентами TTable и TQuery, система Advantage Database Server обеспечивает простую интеграцию с приложениями Delphi. Использование операторов SQL как на удаленном, так и на локальном сервере позволило системе Advantage объединить эффективность обработки, свойственную языку SQL, с высокой скоростью перемещения по наборам данных и управления ими.

Проектирование приложений в Advantage Database Server базируется на простой стратегии, основной целью которой является высокая производительность и устойчивость программ. Благодаря такому подходу при создании программных продуктов удается избежать чрезмерных затрат и трудностей, что позволяет разработчикам быстро создавать эффективно функционирующие приложения. Более того, поскольку такой уникальный подход к проектированию одновременно упрощает эксплуатацию, использование разработанных приложений не потребует чрезмерных усилий со стороны администратора баз данных, за что заказчик, безусловно, будет очень благодарен разработчикам.

Преимущества.

- Использование операторов SQL как на удаленном, так и на локальном сервере.
- Полная масштабируемость и возможность использования на любых платформах.
- Простая интеграция с помощью компонентов TDataSet, дублирующих компоненты TTable и TQuery.
- Обеспечение оптимизации фильтров, используемых в промышленности.
- При распространении в локальных и совместно используемых средах не требуется оплата лицензии.
- Отсутствие затрат на повседневное администрирование базы данных.
- Быстрая и простая установка в существующей сетевой файловой системе и на имеющемся аппаратном обеспечении.

Ограничения: два клиента, только для некоммерческого использования.

Для получения информации об условиях приобретения звоните по телефону (800) 235-7576, дополнительный номер 5030, или обращайтесь к руководству разработчика Delphi 5. Чтобы получить информацию о других программных продуктах и о службе технической поддержки, посетите Web-страницу www.AdvantageDatabase.com.

Async Professional (пробная версия)

от компании TurboPower Software

Async Professional — одна из лучших в мире библиотек, предназначенная для установки последовательных соединений, передачи факсимильных сообщений и обеспечения коммерческой телефонной связи. За счет использования этой библиотеки к разработанным в среде Delphi и C++Builder приложениям можно добавить полнофункциональные средства поддержки связи.

В библиотеке Async Professional есть все: надежное управление взаимодействием портов, высокоскоростные протоколы передачи данных, простые в использовании пакеты управления проверкой входных данных; широкие возможности управления факс-модемом, и все, что требуется для телефонии. Благодаря предоставляемым возможностям можно реализовывать сложные коммерческие коммуникационные решения, в том числе голосовую связь, системы с автоответчиком, системы обратной передачи факсов и многое другое.

В комплект поставки библиотеки Async Professional входит полный исходный текст (написанный в среде разработки Delphi) и более чем 1000 страниц печатной документации. Примеры и файлы справочной системы имеются как в формате Delphi, так и в формате C++Builder. Библиотека совместима со всеми версиями Delphi и C++Builder компании Borland. При создании программ с использованием библиотеки Async Professional не требуется никакой оплаты лицензии.

Если вы хотите приобрести, зарегистрировать или обновить этот программный продукт, посетите Web-страницу компании TurboPower Software по адресу: `www.turbopower.com`, отправьте электронное сообщение по адресу: `sales@turbopower.com` или позвоните по телефону 800/333-41-60 (за пределами США — по телефону 719/260-91-36).

Communicator 4.7

от компании Netscape Communications Corp.

В комплект Netscape Communicator 4.7 входят приложения Navigator, Messenger, Composer, AOL Instant Messenger, PalmPilot Synch. (только для системы Windows), а также надстройки мультимедиа.

Если вы хотите обновить этот программный продукт, посетите Web-страницу компании Netscape Communications по адресу: `www.netscape.com`.

Dalis-SQL 1.5

от компании StarMan Group, Inc.

Dalis-SQL является программой генерации запросов к базе данных. Эта программа принимает запросы на английском языке и генерирует соответствующие операторы SQL.

Если вы хотите приобрести, зарегистрировать или обновить этот программный продукт, посетите Web-страницу компании StarMan Group Software по адресу: `www.starman.com`.

EarthLink TotalAccess 2.3.2

от компании EarthLink Network, Inc.

EarthLink TotalAccess™ — это пакет программного обеспечения Internet и регистрации, особо выделяемый прессой и пользователями за простоту его использования. И это не зря! Пакет включает все программное обеспечение, которое необходимо для получения доступа к Internet, а также содержит ссылку на Web-страницу компании EarthLink. С помощью этой страницы новые пользователи могут получить доступ к сотням важнейших интерактивных ресурсов и любой информации, необходимой для наиболее эффективного использования их учетной записи Internet, предоставляемой компанией EarthLink Network.

Примечание: если броузер у вас уже установлен и настройка сети выполнена, то текущая конфигурация будет перезаписана программой установки EarthLink. Убедитесь в правильности выполняемых действий! Если броузер Internet в системе ранее установлен не был, то подобной проблемы не возникнет.

Ограничения: бесплатная пробная учетная запись Internet в течение 15 дней.

С вопросами о приобретении этого программного продукта обратитесь к Web-странице компании EarthLink Network по адресу `www.earthlink.net`.

Essentials (пробная версия)

от компании TurboPower Software

Essentials представляет собой недорогую библиотеку, в которой содержится несколько десятков визуальных элементов управления, чаще всего применяемых программистами в системе Delphi и C++Builder. При использовании этой библиотеки всего за несколько минут программе можно придать профессиональный вид.

В библиотеке Essentials имеются компоненты, представляющие собой поля для ввода даты и числовой информации, снабженные раскрывающимися календарями и калькуляторами, объемные элементы управления текстом, элементы прокрутки, кнопки с контекстным меню, элементы управления для добавления к формам копий растрового изображения в качестве фона и элементы управления для выбора цвета. В этой библиотеке имеется даже раскрывающийся список, предназначенный для выбора цвета, а также автономный калькулятор и календарь, который можно поместить непосредственно в саму программу. Кроме того, Essentials является единственной библиотекой, содержащей элементы управления типа “свитков”. С их помощью легко создавать формы, которые могут плавно сворачиваться в свиток, когда они не находятся в непосредственном использовании!

В комплект поставки библиотеки Essentials входит полный исходный текст компонентов (написанный в среде разработки Delphi) и более чем 1000 страниц печатной документации. Примеры и файлы справочной системы имеются как в формате Delphi, так и в формате C++Builder. Библиотека совместима со всеми версиями Delphi и C++Builder компании Borland. При создании программ с использованием библиотеки Essentials не требуется оплаты лицензии.

Если вы хотите приобрести, зарегистрировать или обновить этот программный продукт, посетите Web-страницу компании TurboPower Software по адресу: `www.turbopower.com`, отправьте электронное сообщение по адресу: `sales@turbopower.com` или позвоните по телефону 800/333-41-60 (за пределами США — по телефону 719/260-91-36).

Hawk Eye 4.0 (пробная версия)

от компании Agni Software (P) Ltd.

Hawk Eye является первым завершенным отладчиком компонентов Delphi. Традиционно разработка компонентов в Delphi выполнялась не визуально. В любой среде ускоренной разработки (Rapid Application Development — RAD), подобной Delphi, невизуальная разработка всегда была связана с множеством затруднений. И даже после установки разработанных компонентов непосредственно в среду разработки Delphi их по-прежнему невозможно было эффективно протестировать и отладить как во время разработки, так и во время выполнения. И лишь с появлением отладчика Hawk Eye все это стало возможным: компоненты можно протестировать точно так же, как и любые другие приложения Delphi.

Отладчик Hawk Eye предоставляет среду, напоминающую среду разработки Delphi. Однако она разработана специально для облегчения тестирования компонентов. При отладке компонентов Hawk Eye обеспечивает ряд полезных возможностей, в том числе журнал событий. При тестировании компонента во время разработки в его код можно также помещать точки останова. Отладчик Hawk Eye поддерживает редакторы свойств и компонентов, так что компоненты и их редакторы можно эффективно отлаживать как во время разработки, так и во время выполнения. Одним словом, отладчик Hawk Eye — это лучшее и самое простое средство тестирования и отладки компонентов.

По вопросам приобретения, регистрации или обновления этого программного продукта обращайтесь к Web-странице компании Agni по адресу: `www.agnisoft.com`.

InfoPower 2000

от компании Woll2Woll Software

InfoPower 2000 является одной из наиболее популярных библиотек компонентов, при использовании которой профессиональные разработчики баз данных в Delphi и C++Builder получают практически неограниченные возможности. В этой библиотеке содержится превосходная сетка Database, представляющая собой новый мощный компонент отображения данных. С помощью этого компонента можно иерархически просматривать данные из нескольких наборов, а также использовать их по своему усмотрению, подобно инспектору объектов Delphi. В библиотеке InfoPower 2000 имеются также элементы управления просмотром записей, мощный текстовый процессор RichEdit, улучшенные компоненты просмотра нескольких полей и ускоренного поиска. Кроме того, в этой библиотеке содержится множество прочих элементов управления, таких как бегунки, раскрывающиеся списки и элементы управления датой/временем. Библиотека InfoPower предоставляет также гибкий и выразительный язык проверки изображений, расширяемый навигатор в базе данных, мощные средства визуальной фильтрации данных, поддержку QBH, создание пользовательских окон, возможность использования эффекта прозрачности и многое другое.

Если вы хотите приобрести, зарегистрировать или обновить этот программный продукт, посетите Web-страницу компании Woll2Woll Software по адресу: www.woll2woll.com, отправьте электронное сообщение по адресу: sales@woll2woll.com, позвоните по телефону (925) 371-16-63 или отправьте факс по номеру (925) 371-16-64.

Internet Explorer 5.0

от компании Microsoft Corporation

Internet Explorer 5.0 является последней версией Web-броузера компании Microsoft. В IE 5 поддерживается динамический язык HTML, язык Java и формат CDF.

По вопросам обновления этого программного продукта обращайтесь на Web-страницу компании Microsoft по адресу: www.microsoft.com.

IntraBob v3.01

от Боба Сварта (Bob Swart)

Программный продукт IntraBob v3.01 предназначен для тестирования приложений CGI и WinCGI, а также для тестирования и отладки библиотек DLL, поддерживающих интерфейс ISAPI. Кроме того, этим продуктом поддерживаются также модули Web.

Последнюю версию IntraBob можно найти в Internet по адресу: www.drbob42.com (или более точно, www.drbob42.com/ftp/intrabob.zip).

OnGuard (пробная версия)

от компании TurboPower Software

Лучший способ продажи разработанного программного обеспечения — предоставить потенциальным покупателям возможность его попробовать. Именно такой подход и применяется системой OnGuard. Этот программный продукт представляет собой средство установки защиты, с помощью которого можно создавать защищенные версии приложений Delphi и C++Builder, распространяемые по принципу “попробуй, а затем приобретай” (“try-before-you-buy”).

Если программный продукт был создан с использованием системы OnGuard, то перед его приобретением покупатели имеют возможность познакомиться с его полнофункциональной версией. После приобретения программы для разблокирования пробной версии и ее дальнейшего использования в полном объеме можно воспользоваться технологией Unlocking Code Generation, также предоставляемой системой OnGuard. С помощью этой технологии можно увеличить период апробирования, получить доступ к конкретным возможностям или разблокировать всю программу.

Система OnGuard предоставляет также полный набор стратегий защиты, который можно использовать без изменений или настроить по своему усмотрению, повысив степень защищенности программного обеспечения. С помощью OnGuard легко можно ограничить возможность запуска программы в заданном диапазоне даты, в течение определенного периода времени или заданным количеством запусков. Можно даже заблокировать программу на конкретном компьютере, чтобы пользователь не смог ее скопировать на другое рабочее место. Кроме того, система OnGuard обеспечивает защиту Network Metering, так что использовать программу в локальной сети одновременно может только допустимое количество лицензированных пользователей.

В комплект поставки системы OnGuard входит весь исходный код (написанный в среде разработки Delphi), а также полный комплект печатной документации. Кроме того, примеры и файлы справочной системы имеются как в формате Delphi, так и в формате C++Builder. Система совместима со всеми версиями Delphi и C++Builder начиная с версии 3. При создании программ с использованием системы OnGuard не требуется никакой оплаты лицензии.

Если вы хотите приобрести, зарегистрировать или обновить этот программный продукт, посетите Web-страницу компании TurboPower Software по адресу: `www.turbopower.com`, отправьте электронное сообщение по адресу: `sales@turbopower.com` или позвоните по телефону 800/333-41-60 (за пределами США — по телефону 719/260-91-36).

Orpheus (пробная версия)

от компании TurboPower Software

В пакете Orpheus содержится более 100 визуальных элементов управления, при использовании которых разрабатываемые программы будут лучше выглядеть и функционировать. Эти проверенные временем элементы управления применяются профессиональными разработчиками ежедневно, благодаря чему их проекты становятся более быстрыми и надежными.

В пакете Orpheus содержатся мощные элементы управления вводом даты с проверкой, текстовые редакторы, блокноты, бегунки, счетчики, календари, сетки, ограничительные рамки и многое другое. В состав Orpheus включены как элементы управления общего назначения, так и элементы управления, предназначенные для работы с данными.

В комплект поставки пакета Orpheus 3 входит весь исходный код (написанный в среде разработки Delphi), а также более 1000 страниц печатной документации. Кроме того, примеры и файлы справочной системы имеются как в формате Delphi, так и в формате C++Builder. Пакет Orpheus 3 совместим со всеми версиями Delphi и C++Builder компании Borland. При создании программ с использованием Orpheus не требуется никакой оплаты лицензии.

Если вы хотите приобрести, зарегистрировать или обновить этот программный продукт, посетите Web-страницу компании TurboPower Software по адресу: `www.turbopower.com`, отправьте электронное сообщение по адресу: `sales@turbopower.com` или позвоните по телефону 800/333-41-60 (за пределами США — по телефону 719/260-91-36).

PowerTCP Internet Toolkit Evaluation

от компании DART Communications

С помощью этого пакета любое приложение Windows можно быстро адаптировать к использованию в Internet. Комплект PowerTCP позволяет создавать поддерживающие протокол TCP/IP приложения, которые могут использоваться в любой среде разработки вместе с элементами управления ActiveX (OCX), VBX, библиотеками DLL, компонентами Delphi и библиотеками C++. Пакет PowerTCP поддерживает компоненты TCP, TELNET, FTP, SMTP, POP3, средства эмуляции VT320, SNMP, TFTP, UDP, HTTP, FINGER, REXEC, RLOGIN, RSHELL, TIME и WHOIS. Для каждого компонента имеется пример приложения, демонстрирующего его функции, методы и свойства.

Для получения информации об условиях приобретения, регистрации или обновления этого программного продукта посетите Web-страницу компании DART Communications по адресу: `www.dart.com`.

ReportBuilder Pro 4.21 (пробная версия)

от компании Digital Metaphors Corporation

Системы ReportBuilder и ReportBuilder Pro представляют собой средства генерации отчетов, которые могут использоваться как разработчиками, так и конечными пользователями. Эти системы имеют профессиональный пользовательский интерфейс, подобный Office 97, а также полностью расширяемую объектно-ориентированную архитектуру и обеспечивает все возможности, необходимые для создания программного обеспечения в рамках предприятия. В системе ReportBuilder разнообразие функций сочетается с высоким качеством.

Ограничения.

- При каждом запуске ReportBuilder отображается “nag”-экран.
- На печать выводятся отчеты размером до пяти страниц.
- При выводе на печать в верхней части каждой страницы помещается название компании Digital Metaphors и ее номер телефона.

Этот программный продукт можно заказать, обратившись на защищенную интерактивную страницу компании Digital Metaphors, позвонив по телефону 972/931-19-41 или отправив факс по номеру 972/931-78-65.

Rubicon 2.07 (пробная версия)

от компании Tamarack Associates

Встречалось ли вам когда-нибудь в действительности быстрое средство поиска? Хотите ли вы, чтобы в ваших приложениях был интерфейс поиска, который имеется на всех популярных поисковых серверах Internet? Приходится ли вам выполнять поиск большого количества ресурсов? Если да, то вам подойдет именно Rubicon — средство текстового поиска. Достаточно позволить системе Rubicon проиндексировать данные, и пользователи смогут выполнять поиск, введя лишь несколько слов. Условие поиска может иметь вид выражения, содержащего логические операции И, ИЛИ, НЕ и т.д. Естественно, условие поиска может также содержать целые фразы и символы заполнения.

Большинство операций поиска выполняется системой Rubicon очень быстро: как правило, операции с локальными данными занимают менее 0.1 секунды, тогда как общепринятые приемы поиска работают в несколько тысяч раз дольше. Как это происходит? После выпол-

нения предварительной индексации всех слов в той части данных, где планируется проводить поиск, системе Rubicon впоследствии в большинстве случаев уже не требуется считывать данные. Вместо этого для поиска требуемой информации она просто использует индексы.

Для получения информации об условиях приобретения, регистрации или получения обновленных версий этого программного продукта посетите Web-страницу компании Tamarack Associates по адресу: `www.tamaracka.com`.

Пробная версия StarTeam Workstation

от компании StarBase Corporation

StarTeam Workstation — полностью интегрированный набор средств управления работой группы разработчиков и настройки программного обеспечения. Этот набор средств коренным образом изменяет принципы работы группы разработчиков в локальной сети и позволяет создавать лучшие программные продукты гораздо быстрее и с более низкой стоимостью, чем раньше. Комплект StarTeam Workstation представляет собой версию файлового сервера, которая может использоваться небольшими группами разработчиков.

Для получения информации об условиях приобретения, регистрации или получения обновленных версий этого программного продукта посетите Web-страницу компании StarBase по адресу: `www.starbase.com`.

SysTools (пробная версия)

от компании TurboPower Software

В библиотеке SysTools 2 имеется более 800 оптимизированных подпрограмм для операций со строками, выполнения арифметических операций над значениями даты и времени, сортировки, математических вычислений с высокой точностью, надежного доступа к системному реестру и многих других потребностей.

В специализированной библиотеке Real Business Finance and Statistics имеется много мощных функций Microsoft Excel, которые можно поместить прямо в создаваемый проект. Кроме того, средство Bar Code Builder позволяет генерировать широкий диапазон стандартных промышленных штриховых кодов.

Подпрограммы библиотеки SysTools способны обрабатывать даже вложения электронных сообщений, проверять целостность данных путем подсчета контрольной суммы, а также управлять данными в многократно используемых контейнерных классах, таких как деревья, хэш-таблицы и отсортированные множества.

Воспользуйтесь элементами управления библиотеки SysTools, чтобы получить быстрый доступ к диалоговым окнам Windows, таким как окно просмотра или форматирования. Кроме того, очень просто выполнять файловые операции с использованием всех возможностей анимации Windows. Можно даже создать программы, которые будут сворачивать панель задач Windows!

Состоящая из отдельных модулей, библиотека SysTools позволяет добавлять в приложение только необходимый код. Это гарантирует, что код программы никогда не окажется неоправданно большим.

В комплект поставки пакета библиотеки SysTools 2 входит весь исходный текст (написанный в среде разработки Delphi), а также большое количество печатной документации. Кроме того, примеры и файлы справочной системы имеются как в формате Delphi, так и в формате C++Builder. Библиотека Systools 2 совместима со всеми версиями Delphi и C++Builder компании Borland. При создании программ с использованием SysTools не требуется никакой оплаты лицензии.

Если вы хотите приобрести, зарегистрировать или обновить этот программный продукт, посетите Web-страницу компании TurboPower Software по адресу: `www.turbopower.com`, отправьте электронное сообщение по адресу: `sales@turbopower.com` или позвоните по телефону 800/333-41-60 (за пределами США — по телефону 719/260-91-36).

WinZip 7.0 (SR-1)

от компании Nico Mak Computing, Inc.

Программа WinZip позволяет эффективно использовать в Windows файлы `.zip` и другие архивные файлы и форматы сжатия. Дополнительный мастер делает процесс восстановления данных из архива гораздо проще, чем когда-либо. К отличительным особенностям системы WinZip относится встроенная поддержка CAB-файлов и файлов таких популярных форматов Internet, как TAR, gzip, UUencode, BinHex и MIME. Через отдельные внешние программы поддерживаются также утилиты сжатия ARJ, LZH и ARC. Система WinZip позволяет взаимодействовать с большинством антивирусных программ-сканеров.

Ограничения: 30-дневная условно-бесплатная пробная версия.

Для получения информации об условиях приобретения, регистрации или получения обновленных версий этого программного продукта посетите Web-страницу компании Nico Mak Computing по адресу: `www.winzip.com`.

Предметный указатель

$

$DENYPACKAGEUNIT, 121
$DESIGNONLY, 121
$G, 121
$I, 905
$IMPLICITBUILD, 121
$IMPORTEDDATA, 121
$Q, 905
$RUNONLY, 121
$WEAKPACKAGEUNIT, 121

A

ActiveForm, *389*; *432*; *439*
ActiveX, *198*; *199*; *301*; *388*
  ActiveForm, *389*; *432*; *439*
  автоматизация, *200*
  документы, *199*
  лицензии, *391*
  пересылающие методы, *392*
  серверы, *389*
  среда разработки, *417*
  страницы свойств, *389*; *420*
  флаги OLEMISC, *417*
  элементы управления, *388*
ADO, *506*; *596*
AppBar, *329*

B

BDE, *506*; *651*
  API
    `DbiCopyTable()`, *667*
    `DbiGetSeqNo()`, *657*
    `DbiOpenDatabase()`, *659*
    `DbiOpenUserList()`, *668*
    `dbiPackTable()`, *656*
    `DbiSetProp()`, *654*; *664*
    `dbiUndeleteRecord()`, *656*
  дескрипторы, *652*
  курсоры, *652*
    синхронизация, *652*
  модуль определения, *651*

C

CGI, *702*
Clipboard, *201*
CLSID, *199*; *204*
COM, *198*; *199*; *202*; *491*; *495*
  API
    `CoCreateInstance()`, *213*
    `CoCreateInstanceEx()`, *215*
    `CoFreeUnusedLibraries()`, *213*
    `CoGetClassObject()`, *213*; *214*
    `CoInitialize()`, *213*; *214*
    `CoRegisterClassObject()`, *214*
    `CoUninitialize()`, *213*; *214*
    `GetActiveObject()`, *256*

`RegisterActiveObject()`, *253*
CLSID, *199*
DCOM, *199*; *215*
GUID, *203*
IID, *208*
MTS, *275*
автоматизация, *215*
агрегирование, *215*
виртуальные таблицы, *202*
внешний сервер, *214*
внутренний сервер, *211*
диспинтерфейсы, *274*
заглушки, *266*
идентификатор интерфейса, *204*
идентификатор класса, *204*
интерфейсы, *202*
компонентный класс, *199*
маршалинг, *198*; *266*; *491*
методы
`Item()`, *258*
многопоточное разделение, *201*
однопоточное разделение, *201*
подсчет ссылок, *204*
потоковые модели, *201*
псевдоним метода, *208*
терминология, *199*
фабрика класса, *210*; *417*
функции
`CoCreateInstance()`, *344*
`CoInitialize()`, *344*
`CoUninitialize()`, *344*

cookies, *720; 887*

CORBA, *491*; *615*; *740*
DII, *525*
ORB-брокер, *491*
Smart Agent, *522*
VisiBroker, *493*
заглушки, *492*
интерфейсы, *491*
`ICorbaObject`, *496*
`ISkeletonObject`, *496*
`IStubObject`, *496*
каркасы, *492*
классы, *496*
`TBOA`, *497*
`TCorbaDispatchStub`, *495*; *497*; *531*
`TCorbaFactory`, *497*
`TCorbaImplementation`, *495*; *497; 503*
`TCorbaListManager`, *497*
`TCorbaObjectFactory`, *497*
`TCorbaSkeleton`, *497*
`TCorbaStub`, *497*; *531*
`TORB`, *497*
`TQueryServerCorbaFactory`, *524*
клиенты, *522*
методы
`CorbaBind()`, *526*
полномочия, *492*
представители, *492*
связывание
позднее, *523*; *525*
раннее, *523*
хранилище интерфейсов, *493*
язык IDL, *492*

D

DAO, *597*

DCOM, *199*; *215*

DDL, *599*; *615*

DLL, *703*

G

GUID, *495*

H

Handle, *25*

HTML, *701*; *756*
Cookies, *720, 887*
перенаправления, *724*
потоки данных, *727*

таблицы, *714*
формы, *725*
HTML-таблицы, *714*
HTTP, *701*; *740*; *742*

I

IDL, *492*
последовательность, *508*
IID, *204*; *495*
IIS, *703*
InterBase, *615*
interface, *202*
Internet
приложения, *851*
InternetExpress, *759*
ISAPI, *703*

M

MIDAS, *611*; *733*; *755*; *829*
InternetExpress, *759*
архитектура приложений, *736*
компоненты
TClientDataSet, *744*; *749*; *763*; *829*; *839*
TCORBAConnection, *740*
TDataSetProvider, *742*; *764*; *829*
TDCOMConnection, *740*; *743; 760*; *839*
TSocketConnection, *740*; *742*; *744*
TWebConnection, *740*; *744*
TClientDataSet, *741*
методы
TClientDataSet.Refresh(), *752*
модель портфеля, *829*
модуль RDM, *736*
провайдеры, *743*
события
DataSetProvider.OnUpdateError, *747*
TClientDataSet.OnFilterRecord, *842*
TClientDataSet.OnReconcileError, *747*; *840*
TDataSetProvider.BeforeUpdateRecord, *746*
тонкие клиенты, *734*
уровень, *733*
MIME, *728*
MSMQ, *202*
MTA, *201*
MTS, *202*; *275*
диспетчер ресурсов, *280*
заместители, *276*
интерфейсы
ICreateHomeLoan, *279*
IObjectContext, *278*; *283*
IObjectControl, *282*
ISharedProperty, *285*
ISharedPropertyGroup, *285*
ISharedPropertyGroupManager, *285*
класс TMtsAutoObject, *282*
методы
GetObjectContext(), *278*
IObjectContext.IsCallerInRole(), *279*
IObjectContext.SetAbort(), *278*
IObjectContext.SetComplete(), *278*
пакеты, *278*
распределитель ресурсов, *280*; *285*
ресурсы, *280*
роли, *279*
транзакции, *279*

N

NSAPI, *703*

O

Object Windows Library, *23*
ODBC, *590*; *597*
OLE, *198*; *199*; *200*; *301*
активизация по месту, *200*

внедрение, *200*
контейнеры, *199*
объект данных, *201*
связывание, *200*
серверы, *199*
составной документ, *199*
структурированное хранилище, *201*
OLE DB, *597*
OMG, *491*
Open Database Connectivity, *590*
Open Tools API, *455*
интерфейсы
`IOTAActionServices`, *475*
`IOTACreator`, *482*
`IOTAFormWizard`, *482*
`IOTAMenuWizard`, *458*
`IOTAModuleCreator`, *482*
`IOTANotifier`, *457*
`IOTARepositoryWizard`, *482*
`IOTAWizard`, *457*; *458*; *459*; *482*
классы
`TComponentEditor`, *456*
`TFormDesigner`, *456*
`TIAddInNotifier`, *456*
`TIComponentInterface`, *456*
`TIEditInterface`, *456*
`TIEditReader`, *456*
`TIEditView`, *456*
`TIEditWriter`, *456*
`TIExpert`, *456*
`TIFileStream`, *456*
`TIFormInterface`, *456*
`TIMemoryStream`, *456*
`TIMenuItemIntf`, *456*
`TIModuleInterface`, *456*
`TIModuleNotifier`, *456*
`TInterface`, *456*
`TIStream`, *456*
`TIToolServices`, *456*
`TIVCSClient`, *456*
`TIVirtualFileSystem`, *456*
`TIVirtualStream`, *456*
`TNotifierObject`, *458*
`TPropertyEditor`, *456*
`TStream`, *456*
классы `TIResourceEntry`, *456*
классы `TIResourceFile`, *456*
мастера форм, *482*
ORB-брокер, *492*
VisiBroker, *493*
OWL, *23*

R

RAD, *701*
RDO, *597*
RTL, *24*
RTTI, *23*; *40*

S

`safecall`, *232*
SQL, *586*; *597*; *611*; *751*; *779*
InterBase, *615*
Windows ISQL, *615*
внешний ключ, *618; 783*
вычисляемые столбцы, 617
генераторы, *784*
динамические запросы, *639*
домены, *780*
запрос, *586*
значения по умолчанию, *618*
индексы, *617*
исключительные ситуации, *623*
обобщающие функции, *665*
объединения, *753*
права доступа, *625*
представления, *619*
создание таблицы, *616*
таблицы, *780*; *782*
транзакции, *634*

триггеры, *624*; *780*; *784*
хранимые процедуры, *586*; *620*; *780*; *785*
STA, *201*

T

TCP/IP, *742*
TIMainMenuIntf, *456*
TTypeData, 42
TTypeInfo, 41

U

UDT, *201*

V

VCL, *23*
VisiBroker, *493*; *494*
OAD, *493*
Smart Agent, *493*
инструменты администрирования, *494*

W

WebBroker, *878*
Web-брокер, *760*
Web-серверы, *702*
Win32 API, *904*
BitBlt(), *135*; *137*
CoCreateGUID(), *203*
CombineRgn(), *134*
CopyFile(), *667*
CreateEllipticRgn(), *133*
CreateEllipticRgnIndirect(), *133*
CreatePolygonRgn(), *133*
CreatePolyPolygonRgn(), *133*
CreateProcess(), *97*; *102*; *214*
CreateRectRgn(), *133*
CreateRectRgnIndirect(), *134*
CreateRoundRectRgn(), *133*; *134*
CreateWindowEx(), *91*
DrawText(), *136*
ExtCreateRegion(), *134*
ExtractIcon(), *103*
FreeCurrentRegion(), *134*
GetDoubleClickTime(), *319*
GetLastError(), *910*
GetTabbedTextExtent(), *93*
GetTextMetrics(), *135*
InvalidateRect(), *138*
LISTBOX, *91*
LoadBitmap(), *730*
LoadLibrary(), *213*; *379*
LoadLibraryEx(), *379*
MessageBox(), *365*
PlaySound(), *561*; *674*
RegisterWindowMessage(), *316*; *317*; *331*
SetWindowRgn(), *133*
SHAppBarMessage(), *330*
Shell_NotifyIcon(), *315*
ShellExecute(), *97*
SHFileOperation, *364*
SHGetSpecialFolderPath(), *345*
ShowWindow(), *321*
коды ошибок, *910*
Windows
контекстные меню, *368*
оболочка, *315*
обработчики перемещений, *363*
панели инструментов рабочего стола, *329*
панель задач, *315*
пиктограммы, *318*; *378*
подсказки, *318*
расширения оболочки, 361
ярлыки, *343*

Windows ISQL, ***615***

А

автоматизация, ***199*****;** ***200*****;** ***215*****;** ***391*****;** ***392***
- библиотеки типов, *217*
- двойной интерфейс, *218*
- диспетчерский идентификатор, *216*
- информация о типе, *217*
- коллекции, *256*
- контроллеры, *200*; *215*; *237*
- регистрация объектов, *218*
- связывание, *217*
- связывание идентификаторов, *241*
- серверы, *200*; *215*; *218*
- события
  - источник, *247*
  - исходящий интерфейс, *246*
  - сток, *247*
  - точка подключения, *246*
- события автоматизации, *245*

агрегирование, ***215***

анимация, ***135***

архитектура клиент/сервер, ***606***

Б

база данных, ***542***

библиотеки типов, ***217***

бизнес-правила, ***607*****;** ***619*****;** ***733*****;** ***788***

блокировка записей, ***612***

брокеры запросов объектов, ***491***

буфер обмена, ***201*****;** ***307***

В

виртуальные таблицы, ***218***

владение, ***30***

внедрение, ***200***

внешние ключи, ***618***

вычисляемые поля, ***556***

вычисляемые столбцы, ***617***

Д

демаршалинг, ***503***

дескриптор Win32, ***25***

дескрипторы
- баз данных, *652*
- курсоров, *652*

диапазоны фильтрации, ***570***

динамические запросы, ***639***

директивы компилятора, ***121*****;** ***470*****;** ***876***
- `$D`, *904*
- $DENYPACKAGEUNIT, *121*
- `$DESIGNONLY`, *121*
- $G, *121*
- $I, *905*
- `$IMPLICITBUILD`, *121*
- `$IMPORTEDDATA`, *121*
- $Q, *905*
- $R, *906*
- `$R *.DFM`, *909*
- `$RUNONLY`, *121*
- $WEAKPACKAGEUNIT, *121*

диспетчер ресурсов, ***280***

диспинтерфейсы, ***274***

домены, ***780*****;** ***781***

драйверы ODBC, ***590***

З

заглушка, ***492***

закладка, ***544***

запрос, ***541***

защита данных, ***608***

И

индексы, *542*; *617*

интерфейсы, *198*; *202*; *273*; *491*

- GUID, *203*
- IAmbientDispatch, *418*
- IBindStatusCallback, *440*
- IClassFactory, *210*
- IConnectionPointContainer, *246*; *252*
- IContextMenu, *361*; *369*; *370*
- ICopyHook, *361*; *363*
- IDataBroker, *422*
- IDataObject, *361*
- IDispatch, *216*; *217*; *250*; *257*; *272*; *281*; *417*; *431*; *494*; *495*
- IDropTarget, *361*
- IEnumConnections, *253*
- IEnumVARIANT, *257*
- IExtractIcon, *361*; *379*
- IFont, *416*
- IObjectContext, *283*
- IObjectControl, *282*
- IPersistFile, *347*; *361*; *379*
- IPicture, *416*
- IProvider, *422*
- ISharedProperty, *285*
- ISharedPropertyGroup, *285*
- ISharedPropertyGroupManager, *285*
- IShellExtInit, *361*; *369*
- IShellLink, *343*; *347*; *348*
- IShellPropSheetExt, *361*
- IStrings, *416*
- IUnknown, *203*; *204*; *250*; *266*; *494*; *495*
- диспинтерфейсы, *257*
- идентификаторы, *208*
- исходящие, *246*

информация о типе, *217*

исключения, *651*

- EAbort, *904*
- EAccessViolation, *905*
- EAssertionFailed, *905*
- EBitsError, *905*
- EComponentError, *905*
- EControlC, *905*
- EDbEditError, *905*
- EDdeError, *905*
- EExternalException, *905*
- EInOutError, *905*
- EIntError, *905*
  - EDivByZero, *905*
  - EIntOverflow, *905*
  - ERangeError, *906*
- EIntfCastError, *906*
- EInvalidCast, *906*
- EInvalidGraphic, *906*
- EInvalidGraphicOperation, *906*
- EInvalidOperation, *906*
- EInvalidPointer, *907*
- EListError, *907*
- EMathError, *907*
  - EInvalidOp, *907*
  - EOverflow, *907*
  - EUnderflow, *907*
  - EZeroDivide, *907*
- EMCIDeviceError, *907*
- EMenuError, *908*
- EOleCtrlError, *908*
- EOleError, *908*
- EOutlineError, *908*
- EOutOfMemory, *908*
- EPackageError, *908*
- EParserError, *908*
- EPrinter, *908*
- EPrivilege, *908*
- EPropertyError, *908*
- EReconcileError, *840*
- ERegistryException, *908*
- EReportError, *909*
- EResNotFound, *909*
- EStackOverflow, *908*
- EStreamError, *909*

EStringListError, *909*
EThread, *909*
ETreeViewError, *909*
EVariantError, *644*; *645*
EWin32Error, *909*
Exception, *904*

К

каркас, *492*
категории свойств, *174*
классы
DataSpace, *598*
TActiveXControl, *392*; *417*
TActiveXControlFactory, *417*
TADOCommand, *598*
TADOConnection, *598*
TADODataSet, *598*
TADOQuery, *598*
TADOStoredProc, *598*
TADOTable, *598*
TAutoObject, *221*; *282*; *417*
TBatchMove, *590*
TBDEDataSet, *542*; *675*
TBitmap, *135*; *730*
TBlobField, *559*
TBlobStream, *560*
TCanvas, *136*
TCharProperty, *150*
TClassProperty, *150*
TCollection, *179*
TCollectionItem, *179*
TColorProperty, *150*
TComObject, *210*; *363*; *369*
TComObjectFactory, *210*
TComponent, *24*; *33*; *34*; *65*
TComponentEditor, *159*
TComponentProperty, *150*
TControl, *34*
TCustomContentProducer, *711*
TCustomControl, *25*; *36*; *65*
TCustomForm, *482*
TDatabase, *542*; *736*
TDataLink, *670*
TDataModule, *482*; *574*; *704*; *788*
TDataset, *506*; *541*; *542*; *543*; *652*; *653*
TDataSetTableProducer, *711*; *714*
TDataSource, *390*; *549*; *575*
TDBDataSet, *542*; *739*
TDBEdit, *549*
TDBGrid, *545*
TDBLookupCombo, *549*
TDBNavigator, *103*
TDBNavStatForm, *876*
TDefaultEditor, *160*
TDispatchConnection, *739*; *741*
TEdit, *103*
TEnumProperty, *150*
TField, *549*; *550*; *681*
TFieldDataLink, *670*
TFiler, *164*
TFileStream, *729*
TFloatProperty, *150*
TFont, *416*
TFontNameProperty, *150*
TFontProperty, *150*
TFormDesigner, *159*
TGraphicControl, *36*; *65*
THintWindow, *131*
THTMLTableColumn, *717*
THTTPRequest, *721*
THTTPResponse, *721*
TIniFile, *65*
TIntegerProperty, *150*
TISAPIRequest, *708*
TISAPIResponse, *708*; *710*
TJPEGImage, *730*
TListBox, *87*; *91*
TMemo, *87*
TMethodProperty, *150*
TMtsAutoObject, *281*

TMultiReadExclusiveWriteSynchronizer, *739*
TNotifierObject, *482*
TObject, *65*
TOleContainer, *301*; *306*; *307*
TOrdinalProperty, *150*
TPageProducer, *711*
TPanel, *103*; *107*
TPersistent, *32*; *70*; *164*
TPropertyEditor, *149*
TQuery, *541*; *542*; *586*; *664*; *739*; *764*
TQueryTableProducer, *711*; *715*
TRDSConnection, *598*
TReader, *165*
TResourceStream, *730*
TSetElementProperty, *150*
TSetProperty, *150*
TSpeedButton, *103*
TStoredProc, *542*; *586*; *739*
TStringList, *38*; *705*
TStringProperty, *150*
TStrings, *37*; *416*
TTable, *541*; *542*; *549*; *568*; *575*; *739*
TTimer, *138*
TTreeView, *390*
TWebActionItem, *880*
TWebApplication, *707*
TWebDispatcher, *704*; *705*
TWebModule, *704*
TWebRequest, *705*; *708*
TWebResponse, *705*; *708*; *724*
TWinControl, *25*; *35*; *65*; *103*; *390*; *482*
TWriter, *165*
TObject, *31*
клиентские приложения, *606*
коллекции, *256*; *257*
компонентный класс, *199*
компоненты, *23*
ADOExpress, *596*
DBGrid, *753*
TBatchMove, *667*
TDatabase, *627*; *630*; *743*; *798*; *507*; *829*
TDataSet, *674*; *676*; *744*; *752*; *764*
TDataSetTableProducer, *879*
TDatasource, *764*; *853*
TDBDataSet, *744*
TDBGrid, *390*; *688*; *749*; *876*; *879*
TDispatchConnection, *743*
TFontDialog, *110*
TListView, *259*
TMidasPageProducer, *760*
TOLEnterpriseConnection, *740*
TOpenDialog, *110*
TPageProducer, *763*; *879*; *889*
TQuery, *507*; *637*; *638*; *644*; *648*; *664*; *667*; *743*; *744*; *751*; *798*; *829*
TRemoteDataModule, *829*
TSession, *829*; *879*
TStoredProc, *646*; *798*
TTable, *637*; *667*; *876*; *879*
TTimer, *319*
TUpdateSQL, *754*; *798*
TWebDispatcher, *879*; *881*
TXMLBroker, *760*
владение, *30*
графические, *25*
деструкторы, *81*
конструкторы, *81*
методы, *80*
наследование, *31*; *64*
невизуальные, *25*
оболочки, *112*
пользовательские, *24*
потоки данных, *30*
псевдовизуальные, *131*
разработка, *64*
регистрация, *83*
свойства-события, *76*
события, *28*; *76*
создание, *66*
создание свойств, *67*
стандартные, *24*
тестирование, *85*
компоненты VCL, *390*

контейнеры, ***199***
контекст, ***278***
контекстные меню, ***368***
контроллер автоматизации, ***200***

М

маршалинг, ***198*****;** ***266*****;** ***491*****;** ***503***
мастера форм, ***482***
масштабируемость, ***275*****;** ***607***
метаданные, ***780***
методы
Application.HandleException, *903*
Application.Initialize(), *214*
CdsCustomer.ApplyUpdates(), *746*
DefinePropertyPages(), *420*
GetPackageInfo(), *379*
GetValue(), *151*
IClassFactory.CreateInstance(), *210*; *213*
IClassFactory.LockServer(), *210*
IConnectionPoint.Advise, *247*
IConnectionPoint.GetConnectionInterface(), *247*
IConnectionPointContainer.EnumConnectionPoints(), *246*
IConnectionPointContainer.FindConnectionPoint(), *246*
IDispatch.GetIDsOfNames(), *217*; *272*
IDispatch.GetTypeInfo(), *217*
IDispatch.Invoke(), *216*; *217*; *272*
Invoke(), *241*
IOTAWizard.Execute(), *457*
IUnknown.AddRef(), *204*
IUnknown.QueryInterface(), *204*; *206*
IUnknown.Release(), *204*
Response.SendResponse(), *730*
SetValue(), *151*
TActiveXControlFactory.HasMashineLicense(), *419*
Tcanvas.TextHeight(), *136*
TComponent
DestroyComponents(), *34*
FindComponent(), *34*
GetParentComponent(), *34*
HasParent(), *34*
InsertComponent(), *34*
RemoveComponent(), *34*
TComponent.Create(), *34*
TComponent.Destroy(), *34*
TComponent.Destroying(), *34*
TDatabase.ApplyUpdates(), *629*
TDatabase.Close(), *629*
TDatabase.CloseDatasets(), *629*
TDatabase.Commit(), *629*; *635*
TDatabase.Create(), *629*
TDatabase.Destroy(), *629*
TDatabase.Execute(), *629*
TDatabase.Free(), *629*
TDatabase.Open(), *629*
TDatabase.RollBack(), *630*; *635*
TDatabase.StartTransaction(), *630*; *634*
TDataSet.AllocRecordBuffer(), *679*
TDataSet.Append(), *552*
TDataSet.Cancel(), *553*
TDataSet.CursorPosChanged(), *653*
TDataSet.Delete(), *553*
TDataSet.Edit(), *552*
TDataSet.FindFirst(), *567*
TDataSet.FindLast(), *567*
TDataSet.FindNext(), *567*
TDataSet.FindPrior(), *567*
TDataSet.First(), *543*
TDataSet.FreeRecordBuffer(), *679*
TDataSet.GetBookmarkData(), *682*
TDataSet.GetBookmarkFlag(), *682*
TDataSet.GetFieldData(), *681*
TDataSet.GetRecord(), *679*
TDataSet.GetRecordSize(), *681*
TDataSet.Insert(), *552*

TDataSet.InternalAddRecord(), *684*
TDataSet.InternalClose(), *686*
TDataSet.InternalDelete(), *684*
TDataSet.InternalFirst(), *683*
TDataSet.InternalGotoBookmark(), *683*
TDataSet.InternalHandleException(), *686*
TDataSet.InternalInitFieldDefs(), *686*
TDataSet.InternalInitRecord(), *679*
TDataSet.InternalLast(), *683*
TDataSet.InternalOpen(), *687*
TDataSet.InternalPost(), *685*
TDataSet.InternalSetToRecord(), *683*
TDataSet.IsCursorOpen(), *688*
TDataSet.Last(), *543*
TDataSet.MoveBy(), *543*
TDataSet.Next(), *543*
TDataSet.Prior(), *543*
TDataSet.Refresh(), *565*
TDataSet.SetBookmarkData(), *682*
TDataSet.SetBookmarkFlag(), *682*
TDataSet.SetFieldData(), *681*
TDataSet.UpdateCursorPos(), *652*
Tfiler.DefineBinaryProperty(), *164*
Tfiler.DefineProperty(), *164*
TGraphicControl
  Paint(), *36*
TISAPIResponse.SendRedirect(), *710*
TISAPIResponse.SendResponse(), *710*
TISAPIResponse.SendStream(), *710*
TJPEGImage.SaveToStream(), *730*
TObject.ClassInfo(), *41*
TObject.ClassName(), *41*
TObject.ClassParent(), *41*
TObject.ClassType(), *41*
TObject.Free(), *32*
TObject.InheritsFrom(), *41*
TObject.InstanceSize(), *41*
TOleContainer.CreateLinkToFile(), *304*
TOleContainer.CreateObjectFromFile(), *303*
TOleContainer.InsertObjectDialog(), *302*
TOleContainer.PasteSpecialDialog(), *307*
TPersistent
  DefineProperties(), *33*
TPersistent.Assign(), *32*
TPersistent.AssignTo(), *32*
TPersistent.DefineProperties(), *164*
TQuery.ExecSQL(), *638*
TQuery.FieldValues(), *644*
TQuery.ParamByName(), *641*
TQuery.Prepare(), *639*
TQuery.PrepareCursor(), *664*
TQuery.UnPrepare(), *640*
TStoredProc.ExecProc(), *647*
TStoredProc.Prepare(), *646*
TStrings
  LoadFromFile(), *39*
  SaveToFile(), *39*
Tstrings.Add(), *39*
Tstrings.AddObject(), *39*
Tstrings.AddStrings(), *39*
Tstrings.Assign(), *39*
Tstrings.Clear(), *39*
Tstrings.Delete(), *39*
Tstrings.Exchange(), *39*
Tstrings.IndexOf(), *39*
Tstrings.Insert(), *39*
Tstrings.Move(), *39*
TTable.ApplyRange(), *570*
TTable.BatchMove(), *667*
TTable.CancelRange(), *571*
Ttable.Close(), *543*
TTable.CreateTable(), *574*
TTable.FindKey(), *569*; *638*
TTable.FindNearest(), *569*; *638*
TTable.GotoKey(), *569*
TTable.GotoNearest(), *569*
TTable.OnFilterRecord, *567*
Ttable.Open(), *543*
TTable.SetKey(), *569*
TTable.SetRange(), *570*

`TTable.SetRangeEnd()`, *570*
`TTable.SetRangeStart()`, *570*
`TWebRequest.ExtractCookieFields()`, *722*
`TWebResponse.SendRedirect()`, *724*
`TWebResponse.SendStream()`, *727*
`TWebResponse.SetCookieField()`, *721*
`TWinControl.AlignControls()`, *36*
`TWinControl.CanFocus()`, *36*
`TWinControl.DisableAlign()`, *36*
`TWinControl.EnableAlign()`, *36*
`TWinControl.Focused()`, *36*
`TWinControl.ReAlign()`, *36*
`Tobject.Create()`, *31*
`Tobject.Destroy()`, *31*
`UpdateRegistry`, *742*
деструкторы, *81*
доступа к свойствам, *26*
конструкторы, *81*
пересылающие, *392*
создание, *80*
многопоточное подразделение, *201*
многоуровневые приложения, *733*
модели потоков, *738*
моделирование данных, *780*
модель портфеля, *735*
модули данных, *574*; *736*; *788*

Н

наборы данных, *541*
ADO, *596*
BLOB-поля, *559*
DDL, *615*
ODBC, *590*
Paradox, *864*
база данных, *542*
блокировка данных, *612*
вложенные, *764*
внешние ключи, *618*
вычисляемые поля, *556*
закладка, *544*
записи, *612*
запись, *541*
запрос, *541*
изменение состояния, *565*
индекс, *542*
индексы, *570*; *617*
модули данных, *574*
ограничение выборки, *664*
подстановочные поля, *557*
поиск, *568*
поля, *541*; *549*
столбец, *541*
страницы, *612*
строка, *541*
таблица, *541*
таблицы dBASE, *653*; *656*
таблицы Paradox, *657*
текстовые таблицы, *586*
транзакции, *613*
фильтр, *566*
фильтрация по диапазонам, *570*
надстройки

О

объект данных, *201*
объекты
`ADOExpress`, *597*
контекст, *278*
однопоточная модель, *201*
окно подсказки, *131*
операторы
`as`, *40*; *206*
`is`, *40*

П

пакеты, *113*

времени выполнения, *114*
контроль версий, *121*
пакеты надстроек, *122*
разработки, *114*
разработки и времени выполнения, *114*
редактор пакетов, *116*
слабые пакеты, *121*
установка, *115*
флаги, *378*
палитра компонентов, *110*
переменные
интерфейсы, *207*
пиктограммы, *378*
подстановочные поля, *557*
позднее связывание, *493*
порядковый номер записи, *657*
потоки данных, *164*
потоковые модели, *201*
права доступа, *625; 780*
представления, *619*
привилегии пользователя, *612*
приложения
MIDAS, *733*
клиент, *606*
клиент/сервер, *606*; *733*
сервер, *607*
протокол, *701*
процедуры
DispCallByID(), *275*
GetExtensionVersion(), *707*
HttpExtensionProc(), *707*
LoadPackage(), *379*
RegisterNonActiveX(), *390*
TerminateExtension(), *707*

Р

раздельная модель, *201*
распределитель ресурсов, *280*; *285*
расширения оболочки, *361*
редактор библиотек типов, *504*
редакторы компонентов, *159*
TDefaultEditor, *160*
редакторы свойств, *149*
текстовые, *150*
типы, *149*
роль, *279*

С

свободная модель, *201*
свойства
категории, *174*
редакторы свойств, *149*
связывание, *200*; *217*
сервер автоматизации, *200*
сервер транзакций, *275*
серверные приложения, *607*
серверы
OLE, *199*
события, *28*; *76*
OnDataChange, *670*
OnReconcileError, *747*
TControl.OnClick, *35*
TControl.OnDblClick, *35*
TControl.OnDragDrop, *35*
TControl.OnDragOver, *35*
TControl.OnEndDrag, *35*
TControl.OnMouseDown, *35*
TControl.OnMouseMove, *35*
TControl.OnMouseUp, *35*
TDatabase.OnLogin, *631*
TDataSource.OnDataChange, *549*
TDataSource.OnStateChange, *549*
TDataSource.OnUpdateData, *549*
TPageProducer.OnHTMLTag, *712*
TQueryTableProducer.OnCreateContent, *717*

TQueryTableProducer.OnFormatCell, *718*
TQueryTableProducer.OnGetTableCaption, *718*
TTable.OnCalcFields, *573*
TTable.OnNewRecord, *573*
TWebModule.OnCreate, *715*; *716*
TWinControl.OnEnter, *36*
TWinControl.OnExit, *36*
TWinControl.OnKeyDown, *36*
TWinControl.OnKeyPress, *36*
TWinControl.OnKeyUp, *36*
метод диспетчеризации, *76*
обработчик события, *76*
свойства-события, *77*
создание, *76*
события автоматизации, *245*
соглашения о вызовах
safecall, *232*
COM
RDM-модуль, *736*
автоматизация, *736*
страницы данных, *612*
страницы свойств, *389*; *420*
структурированное хранилище, *201*

Т

таблица, *541*
тестирование, *85*
типы времени выполнения, *40*
типы данных
hDBICur, *652*
hDBIDb, *652*
HResult, *209*
OleVariant, *217*; *524*
SafeArray, *267*
TAppBarData, *331*
TBookmarkStr, *544*
TExcepInfo, *217*
TFieldType, *551*
TGUID, *203*
Variant, *550*; *744*
WideString, *272*
варианты, *270*
множества, *69*
объекты, *70*
перечислимые типы, *68*
простые типы, *67*
толстый клиент, *607*
тонкие клиенты, *734*
транзакции, *613*; *634*
триггеры, *624*; *780*

У

упаковка таблиц
dBASE, *656*
Paradox, *658*

Ф

фабрика класса, *210*; *417*; *503*
фильтры, *566*
функции
CreateComObject(), *213*; *214*
CreateRemoteComObject(), *215*
DbiGetRecord(), *653*
DefineBinaryProperty(), *167*
DefineProperty(), *165*
DisableControls(), *655*
DllCanUnloadNow(), *213*
DllGetClassObject(), *212*; *213*
DllRegisterServer(), *212*; *237*
DllUnregisterServer(), *212*
DragQueryFile(), *369*
EnableControls(), *655*
EnumConnections(), *247*
HandleReconcileError(), *747*

HlinkGoBack(), *440*
HlinkGoForward(), *440*
InitializePackage(), *379*
InterfaceConnect(), *251*
QueryInterface, *246*
Register(), *118*
RegisterComponents(), *118*
RegisterNonActiveX(), *259*
RegisterPropertiesInCategory(), *174*
RegisterPropertyInCategory(), *174*
TApplication
Initialize(), *344*
URLDownloadToFile(), *440*

Х

хранилище объектов, *110*; *876*
хранимые процедуры, *620*; *780*

Ц

целостность данных, *609*

Э

элемент управления, *24*

Я

ярлыки, *345*