

Inventory Manager: Client/Server Development

CHAPTER

33

IN THIS CHAPTER

- **Designing the Back End 1732**
- **Centralizing Database Access: The Business Rules 1741**
- **Designing the User Interface 1758**
- **Summary 1784**

This chapter illustrates how to design a database application using the concepts discussed in Chapter 29, “Developing Client/Server Applications.” Here, we illustrate techniques for developing a two-tier client/server application. In this application we have divided up the application logic, or *business rules*, between both the client and the server. We also illustrate how to centralize data access in a data module, thus allowing us to completely separate the user interface from the database logic.

Back in Chapter 4, “Application Frameworks and Design Concepts,” we introduced you to a framework for forms that could be created independently or as child windows to another control. In this chapter, we use that framework for our user interface.

The database back end used is Local InterBase. The application is designed around a typical auto-parts business model. This business model requires the application to keep track of three primary sets of data:

- *Product inventory.* This includes the quantities of each item in the inventory and how much each item is worth.
- *Sales.* This set contains information on items sold and to which customer these items were sold.
- *Customer.* This set contains information such as name and address.

This is by no means a full-blown inventory manager application. The purpose of this chapter is to focus on the techniques of client/server development. We have provided a complete working application to illustrate that focus.

The chapter is divided into three parts. The first part, “Designing the Back End,” discusses the design of the back end. This includes the database objects you learned about in Chapter 29. The second part, “Centralizing Database Access: The Business Rules,” discusses how to use Delphi’s `TDataModule` to centralize database access. Finally, the third part, “Designing the User Interface,” discusses the design of the actual user interface for the inventory application.

Designing the Back End

We use the Local InterBase Server by InterBase Software Corporation as the back end for the Inventory Manager application. This gives us the capability to design the database entirely through SQL. It also offers the flexibility of being able to move some of the data processing to the server side of the equation through the use of triggers, generators, and stored procedures—which also helps to ensure better data integrity. Another more tangible benefit of the SQL back end is that it can be scaled to a true client/server environment.

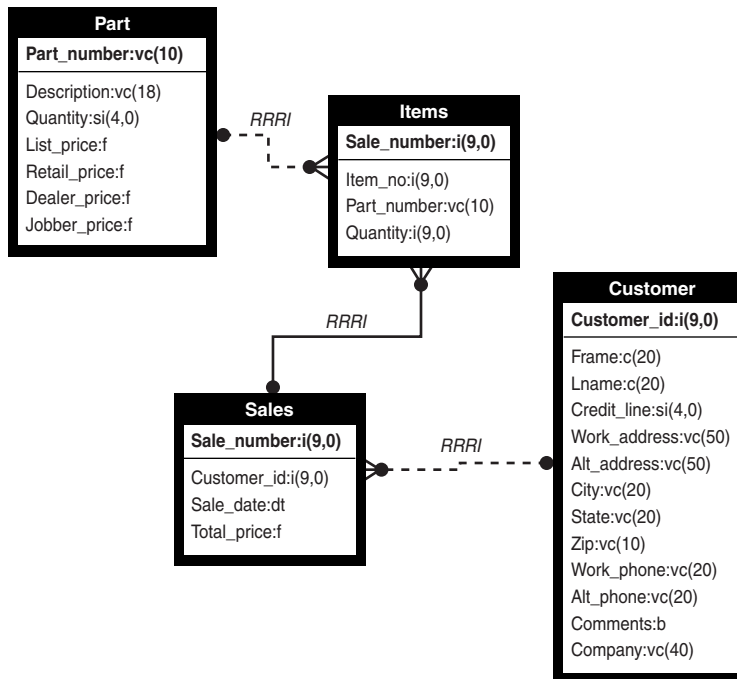
NOTE

Some of the topics in this chapter are specific to InterBase and may not apply to other SQL RDBMSs such as Oracle and Microsoft SQL. The concepts discussed, however, still apply and may just be implemented differently.

As discussed in Chapter 29, we will use SQL to create the various database objects required for the Inventory Manager application. This will include objects such as domains, tables, generators, triggers, stored procedures, and permissions.

There are several ways to create the back end using various *data-modeling* tools. Data-modeling tools such as xCase, RoboCase, Erwin, and SQL-Designer are but a few of the tools that greatly simplify the data-modeling process. All basically allow you to visually model your data without having to type out the SQL code. After you get your basic data-model designed, you can make changes as needed.

Figure 33.1 depicts the data model for our sales application.

**FIGURE 33.1**

Sales application data model.

Defining Domains

Before defining any tables, triggers, and so forth, you define domains that you will use throughout the rest of the SQL code that makes up the *metadata*.

NOTE

Metadata is all the objects (tables, indexes, and so on) contained as part of a database definition.

Think of a *domain* as an entity similar to a user-defined type in Object Pascal. Domains enable you to define special data types with more structure than the built-in data types.

Domains help simplify data and constraint declarations by enabling you to create shorthand names for types that are common throughout your database. Note that you cannot alter a domain after table columns have used it.

The following are some of the domains used in the sales metadata:

- `CREATE DOMAIN DCUSTOMERID AS INTEGER;`

This is a straightforward domain. It defines a new domain called DCUSTOMERID as a type identical to that of a standard, run-of-the-mill integer.

- `CREATE DOMAIN DCREDITLINE AS SMALLINT default 0 CHECK (VALUE BETWEEN 0 AND 3000);`

This defines a new smallint-type domain, but it applies the additional constraint that the value must lie between 0 and 3000.

- `CREATE DOMAIN DNAME AS CHAR(20);`

This defines a domain called DNAME that is a fixed-length string of exactly 20 characters.

- `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);`

This defines several domains as variable-length strings of up to 50, 20, 20, 10, and 20 characters, respectively.

- `CREATE DOMAIN DPRICE AS NUMERIC(15, 2) default 0.00;`

This creates a domain representing a decimal number. The first number, 15, specifies the digits of precision to store. The second number, 2, specifies the number of decimal places to store. The default value for columns of this domain is 0.00.

NOTE

The CHAR(*n*) data type always stores *n* characters to the database. If the string contained in a particular field is less than *n* characters, unused characters will be padded with spaces.

The VARCHAR(*n*) data type stores the exact size of the string, up to a maximum of *n*. Its advantage over CHAR is that it is more space efficient, but operations on VARCHARs tend to be slightly slower.

You can refer to the InterBase Corp. “InterBase Language Reference Guide” or to the IB32.H1p help file for further information on domains.

Defining the Tables

Using the defined domains, you can create tables. Each table is created by using the CREATE TABLE SQL statement, followed by the enumeration of table fields and data types or domains.

The CUSTOMER Table

The CUSTOMER table represents the customer data object, and it is defined as follows:

```
/* Table: CUSTOMER, Owner: 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));
```

The fields defined with the NOT NULL specifier indicate that the user must enter a value for those fields before a record can be posted to the table. In other words, those fields cannot be left blank.

The COMMENTS field requires a bit of explanation. This field is of type BLOB (Binary Large Object), which means that any type of free-form data can be stored there. The SUB TYPE of TEXT, however, means that the data contained within the BLOB is ASCII text and therefore is compatible with the Delphi TDBMemo component.

The `CONSTRAINT` statement creates a primary key on the `CUSTOMER_ID` field, which ensures that each record's value for this field will be unique. This also is the first step to ensuring referential integrity throughout the database; the `PRIMARY KEY` field acts as a lookup field for the `FOREIGN KEY` field defined in another table, as you will see later.

The PART Table

The `PART` table is the shop inventory. This table's definition is fairly straightforward:

```
/* Table: PART, Owner: 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));
```

Each record represents the inventory of one unique part, holding description, quantity, and pricing information. Notice that this table also has a primary key—this time, on the `PART_NUMBER` field.

The SALES Table

The `SALES` table is the table that contains records for every sale to a customer. This table is defined as follows:

```
/* Table: SALES, Owner: 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);
```

Notice the `ALTER TABLE` statement, which adds a foreign key to the `SALES` table. A *foreign key* is a column or set of columns in one table that correspond in exact order to a column or set of columns defined as the primary key in another table. The foreign keys complete the referential integrity with the `SALES` table by ensuring that no entries are made for the `CUSTOMER_ID` field unless an entry with the same customer ID exists in the `CUSTOMER` table.

The ITEMS Table

The `ITEMS` table holds the items, or parts, for a particular sale. The `SALES` table has a one-to-many relationship with the `ITEMS` table and is linked by the `SALE_NUMBER` and `SALE_NO` fields in each table. The `ITEMS` table is defined as follows:

```
/* Table: ITEMS, Owner: 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);
```

Like the SALES table, the ITEMS table has a foreign key that ensures that no record is entered where the part number is nonexistent in the PART table.

Defining Generators

Think of a generator as a mechanism that automatically generates sequential numbers to be inserted into a table. Generators are often used to create unique numbers to be inserted into a table's keyed field. The SALES database will use generators to automatically generate new customer IDs for the CUSTOMER, SALES, and ITEMS tables. These generators are defined as follows:

```
CREATE GENERATOR GEN_CUSTID;
CREATE GENERATOR GEN_ITEMNO;
CREATE GENERATOR GEN_SALENO;
```

NOTE

After you add a generator to a database, it cannot be easily removed. The simplest technique is to remove or modify the trigger or stored procedure so that GEN ID() is not called. You also can remove your generator from the RDB\$GENERATORS systems table.

Defining Triggers

A *trigger* is a routine that automatically performs some action whenever a record in a table is inserted, updated, or deleted. Triggers enable you to let the database perform repetitive tasks as records are committed to tables, thereby freeing the application(s) used to access and modify the data from doing so.

NOTE

Triggers and generators are features specific to InterBase. Although most major SQL vendors also offer these facilities, it is possible that other SQL server vendors use

continues

different syntax and semantics in their implementations. Although they are very nice features, you should keep in mind that using generators and triggers can be a sticky point in migrating the application to a non-InterBase SQL server.

For starters, you need triggers that add new customers and sales numbers to their respective tables using the generators created earlier. The trigger to insert a new, unique customer ID would be as follows:

```
CREATE TRIGGER TCUSTOMER_ID FOR CUSTOMER
ACTIVE BEFORE INSERT POSITION 0
as begin
    new.customer_id = gen_id(gen_custid, 1);
end
```

The following trigger also works on the ITEMS table:

```
CREATE TRIGGER TITEM_NO FOR ITEMS
ACTIVE BEFORE INSERT POSITION 0
as begin
    new.item_no = gen_id(gen_itemno, 1);
end
```

NOTE

There are several additional triggers in this database that convert a two-letter state abbreviation to a full state name. You can find these triggers in `Sales.dd1` on the CD-ROM in the directory for this chapter.

Defining Stored Procedures

A *stored procedure* is a standalone routine that is located on the server as part of a database's metadata.

You can invoke a stored procedure and have it return a dataset just like a normal query. The advantages of stored procedures are that they reduce the amount of processing required at the client end, they reduce the network traffic, and they centralize some particular functionality. Stored procedures also can improve performance because they are precompiled SQL code executed on the server instead of across a network. The general functionality of stored procedures is discussed in greater length in Chapter 29, "Developing Client/Server Applications."

The SALES database employs two stored procedures. The first, `INSERT_SALE`, is used to insert a sale record into the SALES table. This stored procedure takes three input parameters: the

customer ID, the sale data, and the total cost of the sale. This procedure returns the sale identifier generated from within the stored procedure. The client application passes the value returned to another stored procedure where it will be used as a foreign key for the ITEMS table. INSERT_SALE is shown in Listing 33.1.

LISTING 33.1 The INSERT_SALE Stored Procedure

```
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
    /* First obtain a new Sale identifier from the          */
    /* GEN_SALENO generator. This value is being stored in */
    /* the rSale parameter which is defined as a return   */
    /* value and will therefore be returned to the calling */
    /* client.                                             */
    rSALE_NUMBER = gen_id(GEN_SALENO, 1);
    /* Now insert the record into the SALES table */
    INSERT INTO SALES(
        SALE_NUMBER,
        CUSTOMER_ID,
        SALE_DATE,
        TOTAL_PRICE)
    VALUES(
        :rSALE_NUMBER,
        :iCUSTOMER_ID,
        :iSALE_DATE,
        :iTOTAL_PRICE);
END
```

This stored procedure executes some very basic SQL code. It first retrieves a new ID for the sale record from the GEN_SALENO generator. It then performs a simple INSERT INTO SQL statement to insert the data passed to it through parameters.

The second stored procedure used by the application is slightly more complex. This stored procedure is named INSERT_SALE_ITEM and is used to insert individual items of a sale in the ITEMS table. More than likely, this stored procedure will be called several times for a single sale. Therefore, the client will first call the INSERT_SALE stored procedure to insert a sale record. It would have also gotten a sale ID from the call to INSERT_SALE. Then, the client would call

INSERT_SALE_ITEM for each item being sold. For every call, it must pass the specific item information and the sale ID previously obtained.

INSERT_SALE_ITEM takes three parameters: the sale ID, the part number, and the quantity of the item specified being sold. This stored procedure performs a few data-integrity operations. First, it makes sure that there is at least the number of items requested in the PART table. If not, an exception is raised. If the quantity of parts exists, the value of the Qty parameter is subtracted from the quantity in the PART table for the specified part. Finally, the item is added to the ITEMS table.

INSERT_SALE_ITEM is shown in Listing 33.2.

LISTING 33.2 The INSERT_SALE_ITEM Stored Procedure

```
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
    /* CHECK IF IQTY ITEMS EXISTS IN THE PARTS TABLE */
    SELECT QUANTITY FROM PART
        WHERE PART_NUMBER = :iPART_NO
        INTO Actual_Qty;
    IF (Actual_Qty < IQTY) THEN
        EXCEPTION EXP_EXCESS_ORDER;
    ELSE BEGIN
        /* First remove the quantity of parts from the PART table */
        UPDATE PART
        SET QUANTITY = (:Actual_Qty - :iQty)
        WHERE PART_NUMBER = :iPART_NO;
        /* Now Insert the new order */
        INSERT INTO ITEMS(
            SALE_NUMBER,
            PART_NO,
            QTY)
        VALUES(
            :iSALE_NUMBER,
            :iPART_NO,
            :iQTY);
    END
END
```

NOTE

If you are using the ISQL tool to enter database metadata, you need to change the terminating character. Because all statements within a procedure must be terminated by a semicolon (;)—which is also the SQL terminating character—you must set the SQL terminating character to some other symbol to avoid conflicts. Do this by using the `SET TERM` command.

In `SALES`, you will use the caret symbol as the terminating character. This line of SQL code will invoke the following change:

```
SET TERM ^ ;
```

Granting Permissions

The final step in defining a database is granting permission to the tables and stored procedures to particular users. For simplicity, you can grant all users `SELECT` and `UPDATE` rights on the `CUSTOMER` table with the following statement:

```
GRANT SELECT, UPDATE ON CUSTOMER TO PUBLIC WITH GRANT OPTION;
```

Alternatively, you can grant all rights to the `SALES` table with the following statement:

```
GRANT ALL ON SALE TO PUBLIC WITH GRANT OPTION;
```

The `GRANT OPTION` clause means that those who are granted access to tables also are allowed to grant others access to the data. The `GRANT` statements used on the Inventory Manager's tables and stored procedures are as follows:

```
/* Grant permissions for this database */
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;
```

The next section discusses how to connect to the database objects.

Centralizing Database Access: The Business Rules

This section illustrates how to separate database access and business logic from the user interface. This serves several purposes. By placing the business logic within one data module, you make it easier to maintain that same business logic because it is not scattered throughout the

application. This technique also makes it possible to port your two-tier model to a three-tier model by adding the appropriate components to the data module that already contains the business logic. We do not do that here, but we mention this because it is something that merits serious consideration when developing two-tier systems.

You can use `TDataModule` to encompass as much of the database side of things as you see fit. We will show how we do this for the Inventory Manager application.

In our demo application, we use a single `TDataModule` component. For small applications, this approach is sufficient. For larger applications, you might consider separating the disparate pieces among several `TDataModule` components where it logically makes sense.

Listing 33.3 shows the source code for `TDDGSalesDataModule`, which is defined in `SalesDM.pas`.

LISTING 33.3 `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

    // Connection methods

    procedure Logout;
    function Login: Boolean;
    function Connect: Boolean;
    procedure Disconnect;

    // Customer methods
    procedure FirstCustomer;
    procedure LastCustomer;
    procedure NextCustomer;
    procedure PrevCustomer;
    procedure EditCustomer;
    procedure NewCustomer;

```

LISTING 33.3 Continued

```
procedure AcceptCustomer;
procedure CancelCustomer;
procedure DeleteCustomer;
function IsFirstCustomer: Boolean;
function IsLastCustomer: Boolean;
function GetCustomerName: String;
function SearchForCustomer: Boolean;

// Parts methods
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;

// Sales methods

procedure AddItemToSale;
procedure SaveSale;
procedure CancelSale;
function SaleItemsTotalPrice: double;
procedure OpenTempItems;
procedure CloseTempItems;

// Surfaced properties
property AfterTempItemsChange: TDataSetNotifyEvent
    read GetAfterTempItemsChange
    write SetAfterTempItemsChange;

end;

var
    DDGSalesDataModule: TDDGSalesDataModule;

implementation

uses CustomerSrchFrm, LoginFrm;
```

```
{SR *.DFM}

procedure TDDGSalesDataModule.SetAfterTempItemsChange(Value:
    TDataSetNotifyEvent);
begin
    { This writer method adds the Value parameter to both the AfterPost and
      AfterDelete events of the temporary items table. This ensures that whenever
      the data changes, the event handler will get called. }
    tblTempItems.AfterPost := Value;
    tblTempItems.AfterDelete := Value;
end;

function TDDGSalesDataModule.GetAfterTempItemsChange: TDataSetNotifyEvent;
begin
    Result := tblTempItems.AfterPost;
end;

// Login methods.

procedure TDDGSalesDataModule.dbSalesLogin(Database: TDatabase;
    LoginParams: TStrings);
begin
    { Calls method below to populate the LoginParams strings list
      with the user's login information. GetLoginParams is defined in
      LoginFrm.pas. }
    GetLoginParams(LoginParams);
end;

procedure TDDGSalesDataModule.Logout;
begin
    Disconnect;
end;

function TDDGSalesDataModule.Login: Boolean;
begin
    Result := Connect;
end;

function TDDGSalesDataModule.Connect: Boolean;
begin
    { Connects the user to the database. When dbSales is set to True, its OnLogon
      event handler will be invoked which will invoke our customer login dialog
      defined in LoginFrm.pas. }
    try
        dbSales.Connected := True;
    end;
end;
```

LISTING 33.3 Continued

```
    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);
    dbSales.Connected := False;
    Result := False;
end;
end;

procedure TDDGSalesDataModule.Disconnect;
begin
    // Disconnect from the database.
    dbSales.Connected := False;
end;

// Customer methods

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
  { Normally, return the company name. If there is not a company name, return
  the customer's name. }
  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
```

LISTING 33.3 Continued

```
// Assume failure.
Result := False;
{ Invoke the SearchCustomer function which is defined in CustomerSrchFrm.pas.
  this function returns the query string that is added to the
  qryCustomerSearch
  TQuery component }
SearchQry := SearchCustomer;
if SearchQry <> EmptyStr then
begin
  Screen.Cursor := crSQLWait;
  try
    qryCustomerSearch.Close;
    qryCustomerSearch.SQL.Clear;
    qryCustomerSearch.SQL.Add(SearchQry);
    qryCustomerSearch.Open;
  try

    // If a record was not found, exit this method.
    if qryCustomerSearch.FieldName('CUSTOMER_ID').IsNull then
    begin
      Screen.Cursor := crDefault;
      Exit;
    end;

    { If a record is found, get the customer's id that is used to
      locate the record in the actual qryCustomer, TQuery component. This
      will position the cursor to the location of the record. }
    CustID := qryCustomerSearch.FieldName('CUSTOMER_ID').AsInteger;

    { If the record is not found in qryCustomer, there is an
    inconsistency
      in the database, raise an error. }
    if not qryCustomer.Locate('CUSTOMER_ID', CustID, []) then
      raise Exception.Create('Inconsistency in database.')
    else
      Result := True;
  finally
    qryCustomerSearch.Close;
  end;
finally
  Screen.Cursor := crDefault;
end;
end
else
  Result := False;
end;
```

```
// Parts Methods

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;
```

LISTING 33.3 Continued

```
procedure TDDGSalesDataModule.NextPart;
begin
    qryParts.Next;
end;

procedure TDDGSalesDataModule.PrevPart;
begin
    qryParts.Prior;
end;

function TDDGSalesDataModule.SearchForPart: Boolean;
{ This method searches for a part based on the part id specified by the
  user. }
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;

// Sales methods

procedure TDDGSalesDataModule.AddItemToSale;
begin
    { The tblTempItems is a temporary table used to hold the
      items that are being added to a sale. If the user saves
      the sale, these records will be used in the stored procedure
      calls that actually store the sale on the database. }
    if not tblTempItems.Locate('PART_NUMBER',
        qryParts.FieldName('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;
end;
```

```

end
else
  MessageDlg('Item already in list', mtWarning, [mbok], 0);
end;

procedure TDDGSalesDataModule.CancelSale;
begin
  { If the user cancels the sale, the items that were added to the tblTempItems
    table will have to be cleared. }
  tblTempItems.Close;
  tblTempItems.EmptyTable;
  tblTempItems.Open;
end;

procedure TDDGSalesDataModule.SaveSale;
var
  SaleNo: Integer;
begin
  { If the user saves the sale, first create a sale record which will return
    a sale key to SaleNo. This is used as the link for sale items which are
    added next. The sale items are gotten from the temporary table
    tblTempItems. }
  dbSales.StartTransaction;
  try
    { First create the sale record. }
    with spInsertSale do
      begin
        ParamByName('iCUSTOMER_ID').AsInteger := qryCustomer['CUSTOMER_ID'];
        ParamByName('iSALE_DATE').AsDateTime := Now;
        ParamByName('iTOTAL_PRICE').AsFloat := SaleItemsTotalPrice;
        ExecProc;
        // Get the key value in SaleNo.
        SaleNo := ParamByName('rSALE_NUMBER').AsInteger;
      end;

      // Now add all records in tblTempItems to the sale specified by 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;
          end;
        end;
      end;
    end;
  end;
end;

```

LISTING 33.3 Continued

```
        tblTempItems.Next;
    end;

    dbSales.Commit;

    // Refresh modified tables.
    qryParts.Close;
    qryParts.Open;

    tblTempItems.Close;
    tblTempItems.EmptyTable;
    tblTempItems.Open;

except
    dbSales.Rollback;
end;
end;

function TDDGSalesDataModule.SaleItemsTotalPrice: double;
begin
    { qryTotalPrice retrieves the total price for all records added to the
      tblTempItems table. This method may be called from any form using this
      data module. }
    qryTotalPrice.Close;
    qryTotalPrice.Open;
    try
        Result := qryTotalPrice.FieldByName('SUM OF TOTAL_PRICE').AsFloat;
    finally
        qryTotalPrice.Close;
    end;
end;

procedure TDDGSalesDataModule.tblTempItemsBeforePost(DataSet: TDataSet);
begin
    { Before posting a record to the temporary table, calculate the total price
      for the TOTAL_PRICE field based on the number of items that the user is
      adding. }
    tblTempItemsTOTAL_PRICE.ReadOnly := False;
    try
        tblTempItems['TOTAL_PRICE'] := tblTempItems['RETAIL_PRICE'] *
            tblTempItems['QUANTITY'];
    finally
        tblTempItemsTOTAL_PRICE.ReadOnly := True;
    end;
end;
end;
```

```

procedure TDDGSalesDataModule.OpenTempItems;
begin
    tblTempItems.Close;
    tblTempItems.EmptyTable;
    tblTempItems.Open;
end;

procedure TDDGSalesDataModule.CloseTempItems;
begin
    tblTempItems.Active := False;
end;

end.

```

TDDGSalesDataModule has a TDatabase component, dbSales, and the various TQuery, TUpdatesSQL, and TStoredProc components necessary for our sales inventory application.

DbSales is the main connection to the SQL back end that exists in Sales.gdb. This connection is made through the alias DDGSALES, which we set up using the DBExplorer program. DBSales establishes the application-level alias DDGSalesDB. Initially, its Connected property is set to False so all tables belonging to it will also be closed when the application is first run. DbSales has an OnLogin event handler that we will discuss momentarily.

You will notice that we functionally grouped the TDDGSalesDataModule's method definitions. These functional groups are as follows:

<i>Method Group</i>	<i>Definition</i>
Connection methods	Methods that allow the user to log on and log off the application
Customer methods	Methods that manipulate customer data specifically
Parts methods	Methods that manipulate the parts data specifically
Sales methods	Methods that create and manage sales

Refer to the listing's commentary for an explanation of the various methods. In particular, examine the SaveSale() method, which is the method that uses the TStoredProc component to create a new sale and adds sale items to that sale. The stored procedures in this method are hooked to the stored procedures shown in Listings 33.1 and 33.2.

Login/Logout Methods

The methods for logging in and logging out are appropriately named Login() and Logout(). Login() invokes the Connect() method, which establishes a connection to the database through dbSales. It does this by setting the dbSales.Connected property to True. When this

happens, the `dbSales.OnLogin` event handler is invoked if one exists. The event handler `dbSalesLogin()` invokes the `GetLoginParams()` method defined in `LoginFrm.pas`, which populates the user's login information by displaying a custom login dialog. This method is shown in Listing 33.4.

LISTING 33.4 TLoginForm: The Custom Login Form

```
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
    end;
end;
```



```
    LoginForm.Free;  
end;  
end;  
  
end.
```

The `Logout()` method simply closes `dbSales`, which in turn closes all the `TQuery/TTable` connections.

Customer Table Methods

`DDGSalesDataModule` contains several methods to manipulate the `CUSTOMER` table: `NewCustomer()`, `AcceptCustomer()`, `EditCustomer()`, `DeleteCustomer()`, and `CancelCustomer()`. All are straightforward in that they just call the appropriate `TQuery` methods to invoke the action. The remaining methods require a bit more explanation.

`GetCustomerName()` is a function that retrieves the company name of a customer. If a company name does not exist, the method returns the first and last name of a customer whose customer ID is that specified by the `CustID` parameter.

`SearchForCustomer()` allows the user to perform a search on the customer table for a certain customer. The search is based on the fields specified by the user from a customer search form. This form builds a query string that gets passed to the server. We will discuss the functionality of this form later. For now, just assume that it builds a query string that gets assigned to `qryCustomerSearch.SQL`. If the customer specified is found, that customer record is made the active one.

Part Table Methods

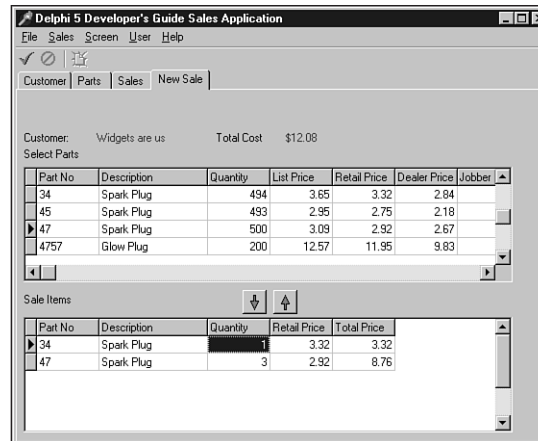
The part methods are similar to the customer methods. The `NewPart()`, `EditPart()`, `AcceptPart()`, `DeletePart()`, and `CancelPart()` methods are simple methods that call the appropriate `TQuery` methods to perform the specific operation.

`SearchForPart()` is not quite as complex as `SearchForCustomer()`. It retrieves a part number by using the `InputQuery()` function and then performs a `Locate()` operation to find the part.

Sales Methods

The sales methods are where things get a bit more interesting. These methods represent more what you would be doing to perform various operations against a client/server database—in particular, the `SaveSale()` method.

`AddItemToSale()` allows the user to specify the items to add to a new sale (see Figure 33.2).

**FIGURE 33.2**

Adding items to a sale.

`CancelSale()` terminates an “insert sale” operation.

`SaveSale()` is `DDGSalesDataModule`'s most complex method. This method uses the transaction capabilities of `dbSales` to add a sale to the database. This involves starting the transaction, adding the sale record, adding x number of items being sold, and then committing or rolling back the entire process (transaction).

The sale record is added by using the `spInsertSale` stored procedure. Notice how the sale number that is generated inside the actual stored procedure is returned to the client with the following statement:

```
SaleNo := ParamByName('rSALE_NUMBER').AsInteger;
```

This value is then used for each record added to the `ITEMS` table through the `TStoredProc` component `spInsertSaleItems`. This is how you link the items being sold with a sale.

Temporary Table Methods

The `TempPartsTable` methods perform operations on the temporary table used to hold items for a sale. Table 33.1 shows the definition of this table.

TABLE 33.1 TEMPPART.DB Table Fields

<i>Field Name</i>	<i>Type</i>	<i>Size</i>	<i>Meaning</i>
PART_NO	A	10	Part number for this item
DESCRIPTION	A	18	Description of this part
QUANTITY	S		Number of parts being sold
RETAIL_PRICE	N	50	Retail price for the item being sold
RETAIL_PRICE	N	50	Total price for the number of parts being sold

The `AddItemToSale()` method is responsible for adding parts to the sale.

The `SaleItemsTotalPrice()` method returns the total price in items existing in `tblTempItems`. This method uses the `qryTotalPrice` component to run a query against the Paradox table to calculate the total price. The SQL statement that is executed is

```
select SUM(RETAIL_PRICE) from temppart.db
```

This statement returns the sum of the numeric values for the specified column—in this case the `RETAIL_PRICE` column.

The `tblTempItemsBeforePost()` method is the event handler for the `tblTempParts.BeforePost` event. This event handler ensures that the record being posted reflects the correct price based on the quantity of items being sold. This is possible because the `BeforePost` event occurs before the record is actually posted to the table.

Surfacing Data-Access Component Events to Users of the TDataModule

One of the problems with centralizing database access is that the data-access components each have their own event that you might want the user interface to know about. Usually, you do this because you want something to happen on the UI side as a result of a data-access component's event. Because the components reside on the `TDataModule`, there is no automatic way for forms using the `TDataModule` to hook into these events. Keep in mind that the `TDataModule` may be made accessible in the form of a compiled unit.

One way of surfacing certain events is to give the `TDataModule` its own event to which any forms using it can attach an event handler. This `TDataModule` event can be invoked as a result of a specific component's event. This is how you surface the `AfterPost` and `AfterDelete` events for the `tblTempParts` table through one property—`AfterTempItemsChange`. This property has both reader and writer methods that directly access the `tblTempParts` actual properties.

Designing the User Interface

With the centralized data access defined, you can build the user interface around the methods, properties, and events of the `TDataModule` object. In the next few sections, we are going to talk about the various forms in the application.

This application uses the framework discussed in Chapter 4, “Application Frameworks and Design Concepts,” where a form can become a child window of another window.

Our application uses the model shown in Figure 33.3.

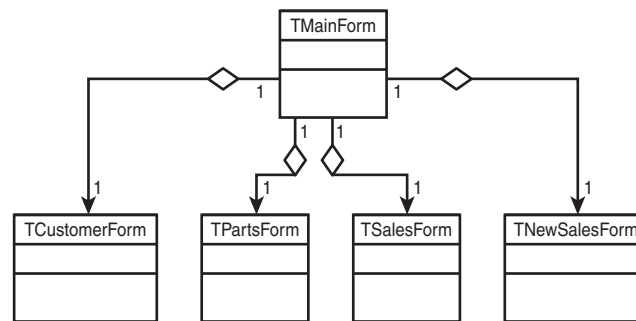


FIGURE 33.3

Inventory application layout.

This main form can contain four child forms:

- *Customer form.* Used to add, edit, and browse customers in the system
- *Parts form.* Used to add, edit, and browse the inventory of parts
- *Sales form.* Used to browse sales
- *New Sales form.* Used to add a new sale

There are some other supportive forms that are not invoked as child forms of the main form. We will discuss these forms momentarily. For now, we will focus primarily on the main form and each of the child forms.

TMainForm: The Application’s Main Form

The main form of the application contains a `TTabControl` component, which serves as the parent component to the child forms. The user changes the child form by selecting the desired screen either from the main menu or by selecting a tab of `tcMain`. The coding logic ensures that the menu items and tab controls remain in sync. Most of the main form logic focuses on ensuring that only one child form is created and visible and that others are properly freed.

Listing 33.5 shows the source code for the main form, TMainForm.

LISTING 33.5 The Inventory Application's Main Form: TMainForm

```
unit MainFrm;

interface

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

type

  { There are four types of child forms that can be displayed in this
    application. The TActiveScreenType is declared to allow us to know
    which of the four types of forms are active. }

  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
    // ActiveScreenType stores the type of form that is active.
    ActiveScreenType: TActiveScreenType;
    // ActiveScreen is a reference to the active child form.
```

LISTING 33.5 Continued

```
    ActiveScreen: TChildForm;
    procedure SetActiveScreen;
public
    { Public declarations }
end;

var
    MainForm: TMainForm;

implementation

uses CustomerFrm, PartsFrm, NewSalesFrm, SalesFrm, SalesDM;

{$R *.DFM}

procedure TMainForm.FormCreate(Sender: TObject);
begin
    // Set the alignment for the main tab control.
    tcMain.Align := alClient;
end;

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

procedure TMainForm.ScreenClick(Sender: TObject);
begin
    { This method is invoked when the user has chosen to change the screen via
      the main menu.
      This method determines if it is possible to change to another child form. It
      does this by making sure that each child form's CanChange() method returns
      True. If so, it changes the global ActiveScreenType value and invokes the
      SetActiveScreen() method to actually perform the change logic. }
    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;

    // Ensure the TTabControl is in-sync with the clicked item on the menu.
    tcMain.TabIndex := ord(ActiveScreenType);
    SetActiveScreen;
    end
end;
end;
end;

procedure TMainForm.tcMainChange(Sender: TObject);
begin
{ This method changes the screen when the user has switched tabs. It
synchronizes the settings for the main menu and the tab control. This method
also calls the SetActiveScreen() method to actually change the active
screen.}
    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;
{ This method changes the active screen to one of the four child forms. Each
child form becomes a child of the TTabControl tcMain. }
var
    TempScreen: TChildForm;
begin
{ Determine if we have an instantiated child form yet. If so, unmerge its
menu and free the child form. }

    TempScreen := ActiveScreen;

    // Unmerge the menu.
    if Assigned(ActiveScreen) then

```

LISTING 33.5 Continued

```
begin
  if ActiveScreen.GetFormMenu <> nil then
    mmSales.UnMerge(ActiveScreen.GetFormMenu);

end;

{ Determine which active screen (child form) to create and set its toolbar
  to have the main form as the parent if appropriate. }
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;

// Merge the menu of the child form with the menu of the main form.
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
```



```
// Change only if the child form is in the mode that allows changing.
AllowChange := ActiveScreen.CanChange;
end;

procedure TMainForm.mmiLogonClick(Sender: TObject);
begin
  // Log the user onto the system
  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
  // Log the user off the system.
  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.
```

Refer to the commentary within the main form's listing (see Listing 33.5) for specifics of each method. The bulk of this application's code exists in `DDGSalesDataModule` (already discussed). The child forms contain most of the logic in regards to the user interface. We will discuss these next.

TCustomerForm: Customer Entry

TCustomerForm is where the user can add, edit, and delete customers from the database. This form is shown in Figure 33.4. Because much of the user interface logic exists in TCustomerForm's ancestor classes, this form's source code is pleasingly thin and simple to understand. Listing 33.6 is the source for TCustomerForm.

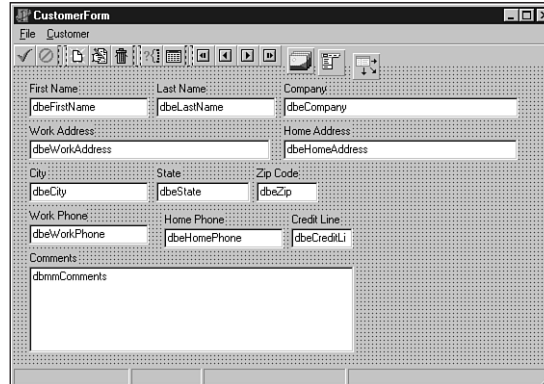


FIGURE 33.4

The customer data-entry form.

LISTING 33.6 Customer Entry Form: TCustomerForm

```
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;
    dbfFirstName: TDBEdit;
    lblLastName: TLabel;
    dbfLastName: TDBEdit;
    lblCreditLine: TLabel;
    dbfCreditLine: 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;
```

LISTING 33.6 Continued

```
{SR *.DFM}

procedure TCustomerForm.SetNavButtons;
begin
    // Ensure that the navigational buttons are set according to the form's mode.
    sbFirst.Enabled := not DDGSalesDataModule.IsFirstCustomer;
    sbLast.Enabled  := not DDGSalesDataModule.IsLastCustomer;
    sbPrev.Enabled  := not DDGSalesDataModule.IsFirstCustomer;
    sbNext.Enabled  := not DDGSalesDataModule.IsLastCustomer;

    // synchronize the navigational menu items with the speedbuttons.
    mmiFirst.Enabled := sbFirst.Enabled;
    mmiLast.Enabled  := sbLast.Enabled;
    mmiPrevious.Enabled := sbPrev.Enabled;
    mmiNext.Enabled  := sbNext.Enabled;
end;

procedure TCustomerForm.sbFirstClick(Sender: TObject);
begin
    // Go to the first record in the result set.
    inherited;
    DDGSalesDataModule.FirstCustomer;
    SetNavButtons;
end;

procedure TCustomerForm.sbPrevClick(Sender: TObject);
begin
    // Go to the previous record in the result set.
    inherited;
    DDGSalesDataModule.PrevCustomer;
    SetNavButtons;
end;

procedure TCustomerForm.sbNextClick(Sender: TObject);
begin
    // Go to the next record in the result set.
    inherited;
    DDGSalesDataModule.NextCustomer;
    SetNavButtons;
end;

procedure TCustomerForm.sbLastClick(Sender: TObject);
begin
    // Go to the last record in the result set.
```

```
    inherited;
    DDGSalesDataModule.LastCustomer;
    SetNavButtons;
end;

procedure TCustomerForm.sbInsertClick(Sender: TObject);
begin
    // Insert a new customer.
    inherited;
    DDGSalesDataModule.NewCustomer;
end;

procedure TCustomerForm.sbEditClick(Sender: TObject);
begin
    // Edit the current customer.
    inherited;
    DDGSalesDataModule.EditCustomer;
end;

procedure TCustomerForm.sbDeleteClick(Sender: TObject);
begin
    // Delete the current customer.
    inherited;
    DDGSalesDataModule.DeleteCustomer;
end;

procedure TCustomerForm.sbCancelClick(Sender: TObject);
begin
    // Cancel the Edit or Add operation.
    inherited;
    DDGSalesDataModule.CancelCustomer;
end;

procedure TCustomerForm.sbAcceptClick(Sender: TObject);
begin
    // Accept Add or Edit changes.
    inherited;
    DDGSalesDataModule.AcceptCustomer;
end;

procedure TCustomerForm.FormShow(Sender: TObject);
begin
    // Initialize menus and buttons accordingly.
    inherited;
    SetNavButtons;
end;
```

LISTING 33.6 Continued

```
function TCustomerForm.CanChange: Boolean;
begin
    // Allow the user to change forms only when browsing record.
    Result := FormMode = fmBrowse;
end;

function TCustomerForm.GetFormMenu: TMainMenu;
begin
    { Return the main menu. This is required by the main form for
      menu merging. }
    Result := mmFormMenu;
end;

procedure TCustomerForm.sbFindClick(Sender: TObject);
begin
    // Search for a specific customer by invoking the customer search form.
    inherited;
    DDGSalesDataModule.SearchForCustomer;
end;

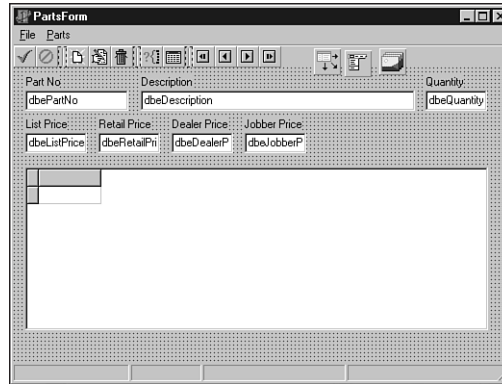
procedure TCustomerForm.sbBrowseClick(Sender: TObject);
begin
    // Set the form to browse mode. This will cancel an edit or add operation.
    inherited;
    if not (FormMode = fmBrowse) then
        DDGSalesDataModule.CancelCustomer;
end;

end.
```

Refer to the listing commentary for explanations of the specific methods. The small amount of code required for this form is possible because most of the database logic exists in `TDDGSalesDataModule`, not to mention how much is handled for you by the VCL. The remaining forms are equally lean.

TPartsForm: Inventory Entry

The parts entry form, `TPartsForm`, is shown in Figure 33.5. Listing 33.7 shows its source code.

**FIGURE 33.5**

The parts data-entry form.

LISTING 33.7 Parts Entry Form: TPartsForm

```
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;
```

33

INVENTORY MANAGER:
CLIENT/SERVER
DEVELOPMENT

continues

LISTING 33.7 Continued

```
    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
    // Ensure that the navigational buttons are set according to the form's mode.
    sbFirst.Enabled := not DDGSalesDataModule.IsFirstPart;
    sbLast.Enabled := not DDGSalesDataModule.IsLastPart;
    sbPrev.Enabled := not DDGSalesDataModule.IsFirstPart;
    sbNext.Enabled := not DDGSalesDataModule.IsLastPart;

    // synchronize the navigational menu items with the speedbuttons.
    mmiFirst.Enabled := sbFirst.Enabled;
    mmiLast.Enabled := sbLast.Enabled;
    mmiPrevious.Enabled := sbPrev.Enabled;
    mmiNext.Enabled := sbNext.Enabled;

end;
```



```
procedure TPartsForm.sbAcceptClick(Sender: TObject);
begin
    // Accept add/edit changes to this part.
    inherited;
    DDGSalesDataModule.AcceptPart;
end;

procedure TPartsForm.sbCancelClick(Sender: TObject);
begin
    // Cancel add/Edit operation.
    inherited;
    DDGSalesDataModule.CancelPart;
end;

procedure TPartsForm.sbInsertClick(Sender: TObject);
begin
    // Insert a new part.
    inherited;
    DDGSalesDataModule.NewPart;
end;

procedure TPartsForm.sbEditClick(Sender: TObject);
begin
    // Edit the current part.
    inherited;
    DDGSalesDataModule.EditPart;
end;

procedure TPartsForm.sbDeleteClick(Sender: TObject);
begin
    // Delete the current part.
    inherited;
    DDGSalesDataModule.DeletePart;
end;

procedure TPartsForm.sbFirstClick(Sender: TObject);
begin
    // Go to the first record in the result set.
    inherited;
    DDGSalesDataModule.FirstPart;
    SetNavButtons;
end;

procedure TPartsForm.sbPrevClick(Sender: TObject);
begin
    // Go to the previous record in the result set.
```

LISTING 33.7 Continued

```
    inherited;
    DDGSalesDataModule.PrevPart;
    SetNavButtons;
end;

procedure TPartsForm.sbNextClick(Sender: TObject);
begin
    // Go to the next record in the result set.
    inherited;
    DDGSalesDataModule.NextPart;
    SetNavButtons;
end;

procedure TPartsForm.sbLastClick(Sender: TObject);
begin
    // Go to the last record in the result set.
    inherited;
    DDGSalesDataModule.LastPart;
    SetNavButtons;
end;

procedure TPartsForm.FormShow(Sender: TObject);
begin
    // Initialize the speedbuttons and menu items accordingly.
    inherited;
    SetNavButtons;
end;

function TPartsForm.CanChange: Boolean;
begin
    // Allow the user to change forms, only if not adding or editing a record.
    Result := FormMode = fmBrowse;
end;

function TPartsForm.GetFormMenu: TMainMenu;
begin
    // Return the main menu. This is used by the main form for menu merging of
    // child forms.
    Result := mmFormMenu;
end;

procedure TPartsForm.sbFindClick(Sender: TObject);
begin
    // Search for a part by the part number.
    inherited;
```

```

DDGSalesDataModule.SearchForPart;
end;

procedure TPartsForm.sbBrowseClick(Sender: TObject);
begin
    // Go into browse mode but only after canceling any changes made to the
    // current record.
    inherited;
    if not (FormMode = fmBrowse) then
        DDGSalesDataModule.CancelPart;
end;

end.

```

You will see from the listing that this is almost identical to `TCustomerForm`. This type of consistency is a desired attribute and one that makes code easier to understand.

TSalesForm: Sales Browsing

The sales form is used to browse existing sales (see Figure 33.6). Its source code contains only one method, `GetFormMenu()`, which had to be overridden to return `nil` so that the main form would not attempt to perform a menu-merging operation. We will not show the listing for this form because there is no specific code that we wrote. You will find its unit, `SalesFrm.pas`, on the CD-ROM in the directory for this chapter.

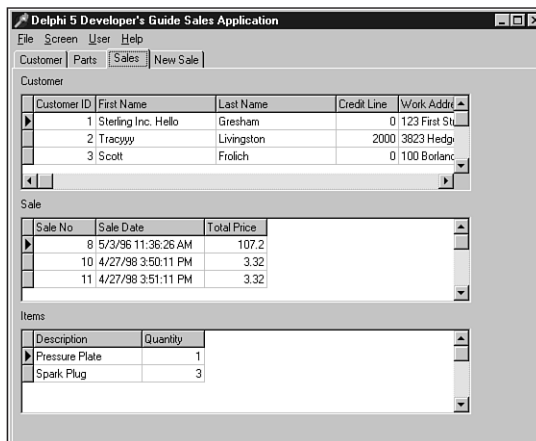


FIGURE 33.6

The sales-browsing form.

TNewSalesForm: Sales Entry

TNewSalesForm is the most complex of the four child forms. Its source is shown in Listing 33.8. Nevertheless, it is still a very simple form. The code commentary discusses the coding logic. In particular, note that we had to create a method to return its TToolBar component. This method already exists in the TDBNavStatForm component of which the other child forms were descendants. This form, however, is a descendant of TChildForm only. Therefore, we needed to create the method for it. Figure 33.7 shows TNewSalesForm.

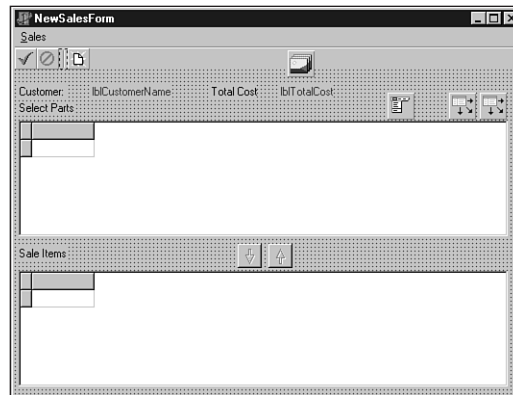


FIGURE 33.7

The new sales data-entry form.

LISTING 33.8 New Sales Entry Form: TNewSalesForm

```
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;

```

LISTING 33.8 Continued

```
begin
    Result := not AddingSale;
end;

procedure TNewSalesForm.FormCreate(Sender: TObject);
begin
    inherited;
    // The tblTempItems table on DDGSalesDataModule is required for this form.
    DDGSalesDataModule.OpenTempItems;
    AddingSale := False; // Initially we're not adding a sale.

    { Assign the TempItemsAfterChange event handler to the event handlers
      surfaced by DDGSalesDataModule. }
    DDGSalesDataModule.AfterTempItemsChange := TempItemsAfterChange;
    SetSaleMenus;
end;

procedure TNewSalesForm.FormDestroy(Sender: TObject);
begin
    // Close the DDGSalesDataModule.tblTempItems table.
    inherited;
    DDGSalesDataModule.CloseTempItems;
end;

procedure TNewSalesForm.FormShow(Sender: TObject);
begin
    // Retrieve the customer name for the current customer.
    inherited;
    lblCustomerName.Caption := DDGSalesDataModule.GetCustomerName;
    { The total should show a balance of zero since the form has just been
      invoked. }
    lblTotalCost.Caption := '$ 0.00';
end;

procedure TNewSalesForm.TempItemsAfterChange(DataSet: TDataSet);
begin
    // This is required in the AfterPost event of the
    // tblTempItems on the datamodule
    // because we must recalculate this everytime the user makes a change.
    lblTotalCost.Caption := FormatFloat('$#,##0.00',
        DDGSalesDataModule.SaleItemsTotalPrice);
end;

procedure TNewSalesForm.sbAddPartClick(Sender: TObject);
begin
```

```
// Add the selected item to the sale.
inherited;
DDGSalesDataModule.AddItemToSale;
end;

procedure TNewSalesForm.mmiNewClick(Sender: TObject);
begin
    // Set the form into a mode to represent adding a sale.
    inherited;
    AddingSale := True;
    SetSaleMenus;
end;

procedure TNewSalesForm.mmiCancelClick(Sender: TObject);
begin
    // Cancel the current sale.
    inherited;
    AddingSale := False;
    DDGSalesDataModule.CancelSale;
    SetSaleMenus;
end;

procedure TNewSalesForm.mmiSaveClick(Sender: TObject);
begin
    // Save the current sale.
    inherited;
    DDGSalesDataModule.SaveSale;
    AddingSale := False;
    SetSaleMenus;
    { Invoke the TempItemsAfterChange event handler to ensure that the form
      updates its controls accordingly. }
    TempItemsAfterChange(nil);
end;

procedure TNewSalesForm.SetSaleMenus;
begin
    // Set menu items and speed buttons to reflect the form's mode.
    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;
```

LISTING 33.8 Continued

```
end;

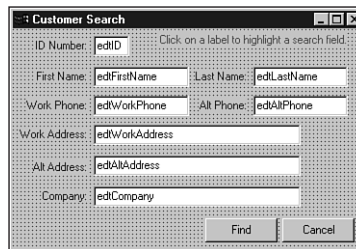
function TNewSalesForm.GetFormMenu: TMainMenu;
begin
    // Return the main menu to be used by the main form for menu merging.
    Result := mmFormMenu;
end;

procedure TNewSalesForm.SetToolBarParent(AParent: TWinControl);
begin
    { This form uses a toolbar, return its parent. We were required to
      create this method for this form as it is a descendant of TChildForm,
      not TDBNavStatForm which already contains this method. }
    tbSales.Parent := AParent;
end;

end.
```

The CustomerSearch Dialog

TCustomerSearchForm is used by DDGSalesDataModule to retrieve a query statement to be used to perform a search on the CUSTOMER table. This form is responsible for obtaining the field values from the user and building the query statement in SQL code. TCustomerSearchForm is shown in Figure 33.8.

**FIGURE 33.8**

The customer search form.

TCustomerSearchForm is not a child form like the previously mentioned forms. TCustomerSearchForm contains no data-aware controls. The user places values into the fields on which a search is to be performed. The user must then click the labels for the fields on which to search. This turns the label's color to c1Red. The logic of TCustomerSearchForm uses the values entered by the user and the TLabel colors to build a SQL query statement.

Listing 33.9 shows the source code for TCustomerSearchForm.

LISTING 33.9 Customer Search Form: TCustomerSearchForm

```
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;
```

LISTING 33.9 Continued

```
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;
{ This function builds an SQL query statement based on the search
  fields of a customer record as specified by the user. The search
  fields are indicated by the labels whose color is clRed. The user
  can select these labels by clicking on them. The user must enter a
  value into the edit field to which the labels refer. }
var
  Sep: String[3]; // Used as a separator.
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;
end;
```

```
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]);
```

continues

LISTING 33.9 Continued

```
end;

procedure TCustomerSearchForm.ClearEditFields;
{ This method clears all of the edit fields and sets their labels
  to clNavy in color. }
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;
  // Clear the edit fields.
  ClearEditFields;
end;

procedure TCustomerSearchForm.FindCustBtnClick(Sender: TObject);
begin
  FindPressed := True;
  // Make the QueryString available to the caller of this dialog.
  QueryString := BuildSQLStatement;
end;

procedure TCustomerSearchForm.CancelBtnClick(Sender: TObject);
begin
  ClearEditFields;
end;

procedure TCustomerSearchForm.lblIDNumberClick(Sender: TObject);
{ All labels are hooked to this OnClick event handler which changes
  the color of the labels. The clRed color is used to specify a label on
  which to perform a search operation. }
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
  { Before closing the form to perform a search operation, make sure
    the user has specified on which fields to perform the search. }
  if (QueryString = '') and FindPressed then
    begin
      MessageDlg('You must highlight a search field by'+
        ' clicking on a label.', mtInformation, [mbOk], 0);
      Action := caNone;
    end
  else begin
      Action := caHide;
      ClearEditFields;
    end;
end;

end.
```

The main method to examine here is the `BuildSQLStatement()` function, which returns a string representing the SQL query statement. This method looks at each of the labels and, if its color is `clRed`, uses its corresponding edit control to build a query statement by using a series of `Format()` statements.

`ClearEditFields()` is a simple method used to set all labels to `clNavy` and to clear the contents of the edit controls. This method is used when the form is created in the `FormCreate()` event handler.

The `FormClose()` event handler ensures that the user has specified fields on which to perform the search by ensuring that `QueryString` is not empty. Only if a field was selected will `QueryString` contain a valid SQL statement. Additionally, this method allows the form to close regardless of the user's specified fields if the user clicked the Cancel button. This is determined by the value of the `FindPressed` Boolean variable, which is set to `True` when the Find button is clicked.

If Find is clicked, the SQL statement is returned to the calling form.

Summary

This concludes the Inventory Application. This chapter illustrates how you would design a client/server, two-tier application. The two-tier model probably makes up the majority of client/server systems. Nevertheless, with the Internet and related technologies, the three-tiered model is becoming more popular and is the topic of later chapters.