COM-Based Technologies

CHAPTER

23

IN THIS CHAPTER

Robust support for COM-based technologies is one of the marquee features of Delphi. The term *COM-based technologies* refers to a number of sundry technologies that rely on COM as their foundation. These technologies include COM servers and clients, ActiveX controls, Object Linking and Embedding (OLE), Automation, and Microsoft Transaction Server (MTS). However, all this new technology at your fingertips can be a bit perplexing, if not daunting. This chapter is designed to give you a complete overview of the technologies that make up COM, ActiveX, and OLE and help you leverage these technologies in your own applications. In earlier days, this topic referred primarily to OLE, which provides a method for sharing data among different applications, dealing primarily with linking or embedding data associated with one type of application to data associated with another application (such as embedding a spreadsheet into a word processor document). However, there is a lot more to COM than just OLE-based word processor tricks!

In this chapter, you will first get a solid background in the basics of COM-based technologies in general and extensions to Object Pascal and VCL added to support COM. You will learn how to apply this knowledge in order to control Automation servers from your Delphi applications and write Automation servers of your own. You will also learn about more sophisticated COM topics, such as advanced Automation techniques and MTS. Finally, this chapter covers VCL's `TOleContainer` class, which encapsulates ActiveX containers. This chapter does not teach you everything there is to know about OLE and ActiveX—that could take volumes—but it does cover all the important features of OLE and ActiveX, particularly as they apply to Delphi.

COM Basics

First things first. Before we jump into the topic at hand, it is important that you understand the basic concepts and terminology associated with the technology. This section introduces you to basic ideas and terms behind the COM-based technologies.

COM: The Component Object Model

The *Component Object Model* (COM) forms the foundation upon which OLE and ActiveX technology is built. COM defines an API and a binary standard for communication between objects that is independent of any particular programming language or (in theory) platform. COM objects are similar to the VCL objects you are familiar with—except they have only methods and properties associated with them, not data fields.

A COM object consists of one or more *interfaces* (described in detail later in this chapter), which are essentially tables of functions associated with that object. You can call an interface's methods just like the methods of a Delphi object.

The component objects you use can be implemented from any EXE or DLL, although the implementation is transparent to you as a user of the object because of a service provided by

COM called *marshaling*. The COM marshaling mechanism handles all the intricacies of calling functions across process—and even machine—boundaries, which makes it possible to use a 32-bit object from a 16-bit application or access an object located on machine A from an application running on machine B. This intermachine communication is known as Distributed COM (DCOM) and is described in greater detail later in this chapter.

COM Versus ActiveX Versus OLE

"So, what's the difference between COM, OLE, and ActiveX, anyway?" That's one of the most common (and reasonable) questions developers ask as they get into this technology. It's a reasonable question because it seems that the purveyor of this technology, Microsoft, does little to clarify the matter. You've already learned that COM is the API and binary standard that forms the building blocks of the other technologies. In the old days (like 1995), *OLE* was the blanket term used to describe the entire suite of technologies built on the COM architecture. These days, OLE refers only to those technologies associated specifically with linking and embedding, such as containers, servers, in-place activation, drag-and-drop, and menu merging. In 1996, Microsoft embarked on an aggressive marketing campaign in an attempt to create brand recognition for the term *ActiveX,* which became the blanket term used to describe non-OLE technologies built on top of COM. ActiveX technologies include Automation (formerly called *OLE Automation*) controls, documents, containers, scripting, and several Internet technologies. Because of the confusion created by using the term *ActiveX* to describe everything short of the family pet, Microsoft has backed off a bit and now sometimes refers to non-OLE COM technologies simply as *COM-based technologies.*

Those with a more cynical view of the industry might say that the term *OLE* became associated with adjectives such as "slow" and "bloated," and marketing-savvy Microsoft needed a new term for those APIs on which it planned to base its future operating system and Internet technologies. Also amusing is the fact that Microsoft now claims OLE no longer stands for *Object Linking and Embedding*—it's just a word that is pronounced *Oh-lay.*

Terminology

COM technologies bring with them a great deal of new terminology, so some terms are presented here before going any deeper into the guts of ActiveX and OLE.

Although an instance of a COM object is usually referred to simply as an *object*, the type that identifies that object is usually referred to as a *component class* or *coclass.* Therefore, to create an instance of a COM *object*, you must pass the CLSID of the COM *class* you want to create.

The chunk of data that is shared between applications is referred to as an *OLE object*. Applications that have the capability to contain OLE objects are referred to as *OLE containers*. Applications that have the capability to have their data contained within an OLE container are called *OLE servers*.

A document that contains one or more OLE objects is usually referred to as a *compound document*. Although OLE objects can be contained within a particular document, full-scale applications that can be hosted within the context of another document are known as *ActiveX documents.*

As the name implies, an OLE object can be *linked* or *embedded* into a compound document. Linked objects are stored in a file on disk. With object linking, multiple containers—or even the server application—can link to the same OLE object on disk. When one application modifies the linked object, the modification is reflected in all the other applications maintaining a link to that object. Embedded objects are stored by the OLE container application. Only the container application is able to edit the OLE object. Embedding prevents other applications from accessing (and therefore modifying or corrupting) your data, but it does put the burden of managing the data on the container.

Another facet of ActiveX that you'll learn more about in this chapter is *Automation,* which is a means to allow applications (called *Automation controllers*) to manipulate objects associated with other applications or libraries (called an *Automation server*). Automation enables you to manipulate objects in another application and, conversely, to expose elements of your application to other developers.

What's So Great About ActiveX?

The coolest thing about ActiveX is that it enables you to easily build the capability to manipulate many types of data into your applications. You might snicker at the word *easily,* but it's true. It is much easier, for example, to give your application the capability to contain ActiveX objects than it is to build word processing, spreadsheet, or graphics-manipulation capabilities into your application.

ActiveX fits very well with Delphi's tradition of maximum code reuse. You don't have to write code to manipulate a particular kind of data if you already have an OLE server application that does the job. As complicated as OLE can be, it often makes more sense than the alternatives.

It also is no secret that Microsoft has a large investment in ActiveX technology, and serious developers for Windows 95, NT, and other upcoming operating systems will have to become familiar with using ActiveX in their applications. So, like it or not, COM is here for a while, and it behooves you, as a developer, to become comfortable with it.

OLE 1 Versus OLE 2

One of the primary differences between OLE objects associated with 16-bit OLE version 1 servers and those associated with OLE version 2 servers is in how they activate themselves. When you activate an object created with an OLE 1 server, the server application starts up and receives focus, and then the OLE object appears in the server application, ready for editing.

When you activate an OLE 2 object, the OLE 2 server application becomes active "inside" your container application. This is known as *in-place activation* or *visual editing.*

When an OLE 2 object is activated, the menus and toolbars of the server application replace or merge with those of the client application, and a portion of the client application's window essentially becomes the window of the server application. This process is demonstrated in the sample application shown later in this chapter.

Structured Storage

OLE 2 defines a system for storing information on disk known as *structured storage*. This system basically does on a file level what DOS does on a disk level. A storage object is one physical file on a disk, but it equates with the DOS concept of a directory, and it is made up of multiple storages and streams. A storage equates to a subdirectory, and a stream equates to a DOS file. You will often hear this implementation referred to as *compound files*.

Uniform Data Transfer

OLE 2 also has the concept of a *data object*, which is the basic object used to exchange data under the rules of uniform data transfer. *Uniform data transfer* (UDT) governs data transfers through the Clipboard, drag-and-drop, DDE, and OLE. Data objects allow for a greater degree of description about the kind of data they contain than previously was practical given the limitations of those transfer media. In fact, UDT is destined to replace DDE. A data object can be aware of its important properties, such as size, color, and even what device it is designed to be rendered on. Try doing that on the Windows Clipboard!

Threading Models

Every COM object operates in a particular threading model that dictates how an object can be manipulated in a multithreaded environment. When a COM server is registered, each of the COM objects contained in that server should register the threading model they support. For COM objects written in Delphi, the threading model chosen in the Automation, ActiveX control, or COM object wizards dictates how a control is registered. The COM threading models include the following:

- *Single.* The entire COM server runs on a single thread.
- *Apartment.* Also known as *single-threaded apartment* (STA). Each COM object executes within the context of its own thread, and multiple instances of the same type of COM object can execute within separate threads. Because of this, any data that is shared between object instances (such as global variables) must be protected by thread synchronization objects when appropriate.

- *Free.* Also known as multithreaded apartment (MTA). A client can call a method of an object on any thread at any time. This means that the COM object must protect even its own instance data from simultaneous access by multiple threads.
- *Both.* Both the apartment and free threading models are supported.

Keep in mind that merely selecting the desired threading model in the wizard doesn't guarantee that your COM object will be safe for that threading model. You must write the code to ensure that your COM servers operate correctly for the threading model you wish to support. This most often includes using thread synchronization objects to protect access to global or instance data in your COM objects. For more information on multithreaded development in Delphi, see Chapter 11, "Writing Multithreaded Applications."

COM+

As a part of the Windows 2000 release, Microsoft has provided the most significant update to COM in recent memory with the release of a new iteration called *COM+*. The goal of COM+ is the simplification of the COM development process through the integration of several satellite technologies, most notably MTS (described later in this chapter) and Microsoft Message Queue (MSMQ). The integration of these technologies into the standard COM+ runtime means that all COM+ developers will be able to take advantage of features such as transaction control, security, administration, queued components, and publish and subscribe event services. Because COM+ consists mostly of off-the-shelf parts, this means complete backward compatibility, such that all existing COM and MTS applications automatically become COM+ applications.

COM Meets Object Pascal

Now that you understand the basic concepts and terms behind COM, ActiveX, and OLE, it's time to discuss how the concepts are implemented in Delphi. This section goes into more detail on COM and gives you a look at how it fits into the Object Pascal language and VCL.

Interfaces

COM defines a standard map for how an object's functions are laid out in memory. Functions are arranged in virtual tables (called *vtables)*—tables of function addresses identical to Delphi class *virtual method tables* (VMTs). The programming language description of each vtable is referred to as an *interface.*

Think of an interface as a facet of a particular class. Each facet represents a specific set of functions or procedures that you can use to manipulate the class. For example, a COM object that represents a bitmap image might support two interfaces: one containing methods that

enable the bitmap to render itself to the screen or printer and another interface to manage storing and retrieving the bitmap to and from a file on disk.

An interface really has two parts: The first part is the interface definition, which consists of a collection of one or more function declarations in a specific order. The interface definition is shared between the object and the user of the object. The second part is the interface implementation, which is the actual implementation of the functions described in the interface declaration. The interface definition is like a contract between the COM object and a client of that object—a guarantee to the client that the object will implement specific methods in a specific order.

Introduced in Delphi 3, the `interface` keyword in Object Pascal enables you to easily define COM interfaces. An interface declaration is semantically similar to a class declaration, with a few exceptions. Interfaces can consist only of properties and methods—no data. Because interfaces cannot contain data, their properties must write and read to and from methods. Most important, interfaces have no implementation because they only define a contract.

IUnknown

Just as all Object Pascal classes implicitly descend from `TObject`, all COM interfaces (and therefore all Object Pascal interfaces) implicitly derive from `IUnknown`, which is defined in the `System` unit as follows:

```
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;
```

Aside from the use of the `interface` keyword, another obvious difference between an interface and class declaration that you will notice from the preceding code is the presence of a *globally unique identifier* (GUID).

Globally Unique Identifiers (GUIDs)

A GUID (pronounced *goo-id)* is a 128-bit integer used in COM to uniquely identify an interface, coclass, or other entity. Because of their large size and the hairy algorithm used to generate these numbers, GUIDs are almost guaranteed to be globally unique (hence the name). GUIDs are generated using the `CoCreateGUID()` API function, and the algorithm employed by this function to generate new GUIDs combines information such as the current date and time, CPU clock sequence, network card number,

continues

and the balance of Bill Gates's bank accounts (okay, so we made up the last one). If you have a network card installed on a particular machine, a GUID generated on that machine is guaranteed to be unique because every network card has an internal ID that is globally unique. If you don't have a network card, it will synthesize a close approximation using other hardware information.

Because there is no language type that holds something as large as 128 bits in size, GUIDs are represented by the `TGUID` record, which is defined as follows in the `System` unit:

```
type
  PGUID = ^TGUID;
  TGUID = record
    D1: LongWord;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
  end;
```

Because it can be a pain to assign GUID values to variables and constants in this record format, Object Pascal also allows a `TGUID` to be represented as a string with the following format:

```
'{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}'
```

Thanks to this, the following declarations are equivalent as far as the Delphi compiler is concerned:

```
MyGuid: TGUID = (
  D1:$12345678;D2:$1234;D3:$1234;D4:($01,$02,$03,$04,$05,$06,$07,$08));

MyGuid: TGUID = '{12345678-1234-1234-12345678}';
```

In COM, every interface or class has an accompanying GUID that uniquely defines that interface. In this way, two interfaces or classes having the same name defined by two different people will never conflict because their respective GUIDs will be different. When used to represent an interface, a GUID is normally referred to as an *interface ID* (IID). When used to represent a class, a GUID is referred to as a *class ID* (CLSID).

TIP

You can generate a new GUID in the Delphi IDE using the Ctrl+Shift+G keystroke in the Code Editor.

In addition to its IID, `IUnknown` declares three methods: `QueryInterface()`, `_AddRef()`, and `_Release()`. Because `IUnknown` is the base interface for COM, all interfaces must implement `IUnknown` and its methods. The `_AddRef()` method should be called when a client obtains and wants to use a pointer to a given interface, and a call to `_AddRef()` must have an accompanying call to `_Release()` when the client is finished using the interface. In this way, the object that implements the interfaces can maintain a count of clients that are keeping a reference to the object, or *reference count*. When the reference count reaches zero, the object should free itself from memory. The `QueryInterface()` function is used to query whether an object supports a given interface and, if so, to return a pointer to that interface. For example, suppose that object `O` supports two interfaces, `I1` and `I2`, and you have a pointer to `O`'s `I1` interface. To obtain a pointer to `O`'s `I2` interface, you would call `I1.QueryInterface()`.

NOTE

If you're an experienced COM developer, you may have noticed that the underscore in front of the `_AddRef()` and `_Release()` methods is not consistent with other COM programming languages or even with Microsoft's COM documentation. Because Object Pascal is "`IUnknown` aware," you won't normally call these methods directly (more on this in a moment), so the underscores exist primarily to make you think before calling these methods.

Because every interface in Delphi implicitly descends from `IUnknown`, every Delphi class that implements interfaces must also implement the three `IUnknown` methods. You can do this yourself manually, or you can let VCL do the dirty work for you by descending your class from `TInterfacedObject`, which implements `IUnknown` for you.

Using Interfaces

Chapter 2, "The Object Pascal Language," and Delphi's own "Object Pascal Language Guide" documentation cover the semantics of using interface instances, so we won't rehash that material here. Instead, we'll discuss how `IUnknown` is seamlessly integrated into the rules of Object Pascal.

When an interface variable is assigned a value, the compiler automatically generates a call to the interface's `_AddRef()` method so that the reference count of the object is incremented. When an interface variable falls out of scope or is assigned the value `nil`, the compiler automatically generates a call to the interface's `_Release()` method. Consider the following piece of code:

```
var
  I: ISomeInteface;
begin
```

```
  I := FunctionThatReturnsAnInterface;
  I.SomeMethod;
end;
```

Now take a look at the following code snippet, which shows the code you would type (in bold) and an approximate Pascal version of the code the compiler generates (in normal font):

```
var
  I: ISomeInterface;
begin
  // interface is automatically initialized to nil
  I := nil;
  try
    // your code goes here
    I := FunctionThatReturnsAnInterface;
    // _AddRef() is called implicitly when I is assigned
    I._AddRef;
    I.SomeMethod;
  finally
    // implicit finally block ensures that the reference to the
    // interface is released
    if I <> nil I._Release;
  end;
end;
```

The Delphi compiler is also smart enough to know when to call `_AddRef()` and `_Release()` as interfaces are reassigned to other interface instances or assigned the value `nil`. For example, consider the following code block:

```
var
  I: ISomeInteface;
begin
  // assign I
  I := FunctionThatReturnsAnInterface;
  I.SomeMethod;
  // reassign I
  I := OtherFunctionThatReturnsAnInterface;
  I.OtherMethod;
  // set I to nil
  I := nil;
end;
```

Again, here is a composite of the user-written (bold) code and the approximate compiler-generated (normal) code:

```
var
  I: ISomeInterface;
begin
  // interface is automatically initialized to nil
```

```
  I := nil;
  try
    // your code goes here
    // assign I
    I := FunctionThatReturnsAnInterface;
    // _AddRef() is called implicitly when I is assigned
    I._AddRef;
    I.SomeMethod;
    // reassign I
    I._Release;
    I := OtherFunctionThatReturnsAnInterface;
    I._AddRef;
    I.OtherMethod;
    // set I to nil
    I._Release;
    I := nil;
  finally
    // implicit finally block ensures that the reference to the
    // interface is released
    if I <> nil I._Release;
  end;
end;
```


The preceding code example also helps to illustrate why Delphi prepends the underscore to the `_AddRef()` and `_Release()` methods. Forgetting to increment or decrement the reference of an interface was one of the classic COM programming bugs in the pre-`interface` days. Delphi's interface support is designed to alleviate these problems by handling the housekeeping details for you, so there's rarely ever a reason to call these methods directly.

Because the compiler knows how to generate calls to `_AddRef()` and `_Release()`, wouldn't it make sense if the compiler had some inherent knowledge of the third `IUnknown` method, `QueryInterface()`? It would, and it does. Given an interface pointer for an object, you can use the `as` operator to "typecast" the interface to another interface supported by the COM object. We say *typecast* because this application of the `as` operator isn't really a typecast in the strict sense but rather an internal call to the `QueryInterface()` method. The following sample code demonstrates this:

```
var
  I1: ISomeInterface;
  I2: ISomeOtherInterface;
begin
  // assign to I1
  I1 := FunctionThatReturnsAnInterface;
  // QueryInterface I1 for an I2 interface
  I2 := I1 as ISomeOtherInterface;
end;
```

In the preceding example, if the object referenced by `I1` doesn't support the `ISomeOtherInterface` interface, an exception will be raised by the `as` operator.

One additional language rule pertaining to interfaces is that an interface variable is assignment compatible with an Object Pascal class that implements that interface. For example, consider the following interface and class declarations:

```
type
  IFoo = interface
    // definition of IFoo
  end;

  IBar = interface(IFoo)
    // definition of IBar
  end;

  TBarClass = class(TObject, IBar)
    // definition of TBarClass
  end;
```

Given the preceding declarations, the following code is correct:

```
var
  IB: IBar;
  TB: TBarClass;
begin
  TB := TBarClass.Create;
  try
    // obtain TB's IBar interface pointer:
    IB := TB;
    // use TB and IB
  finally
    IB := nil;  // explicitly release IB
    TB.Free;
  end;
end;
```

Although this feature seems to violate traditional Pascal assignment-compatibility rules, it does make interfaces feel more natural and easier to work with.

An important but nonobvious corollary to this rule is that interfaces are only assignment compatible with classes that explicitly support the interface. For example, the `TBarClass` class defined earlier declares explicit support for the `IBar` interface. Because `IBar` descends from `IFoo`, conventional wisdom might indicate that `TBarClass` also directly supports `IFoo`. This is not the case, however, as the following sample code illustrates:

```
var
  IF: IFoo;
```

```
  TB: TBarClass;
begin
  TB := TBarClass.Create;
  try
    // compiler error raised on the next line because TBarClass
    // doesn't explicitly support IFoo.
    IF := TB;
    // use TB and IF
  finally
    IF := nil;  // expicitly release IF
    TB.Free;
  end;
end;
```

Interfaces and IIDs

Because the interface ID is declared as a part of an interface declaration, the Object Pascal compiler knows how to obtain the IID from an interface. Therefore, you can pass an interface type to a procedure or function that requires a `TIID` or `TGUID` as a parameter. For example, suppose you have a function like this:

```
procedure TakesIID(const IID: TIID);
```

The following code is syntactically correct:

```
TakesIID(IUnknown);
```

This capability obviates the need for `IID_`*`InterfaceType`* constants defined for each interface type that you might be familiar with if you've done COM development in C++.

Method Aliasing

A problem that occasionally arises when you implement multiple interfaces in a single class is that there can be a collision of method names in two or more interfaces. For example, consider the following interfaces:

```
type
  IIntf1 = interface
    procedure AProc;
  end;

  IIntf2 = interface
    procedure AProc;
  end;
```

Given that each of the interfaces contains a method called `AProc()`, how can you declare a class that implements both interfaces? The answer is *method aliasing*. Method aliasing enables you to map a particular interface method to a method of a different name in a class. The following code example demonstrates how to declare a class that implements `IIntf1` and `IIntf2`:

```
type
  TNewClass = class(TInterfacedObject, IIntf1, IIntf2)
  protected
    procedure IIntf2.AProc = AProc2;
    procedure AProc;   // binds to IIntf1.AProc
    procedure AProc2;  // binds to IIntf2.AProc
  end;
```

In this declaration, the `AProc()` method of `IIntf2` is mapped to a method with the name `AProc()`. Creating aliases in this way enables you to implement any interface on any class without fear of method name collisions.

The HResult Return Type

You might notice that the `QueryInterface()` method of `IUnknown` returns a result of type `HResult`. `HResult` is a very common return type for many ActiveX and OLE interface methods and COM API functions. `HResult` is defined in the `System` unit as a `type LongWord`. Possible `HResult` values are listed in the `Windows` unit (if you have the VCL source code, you can find them under the heading `{ HRESULT value definitions }`). An `HResult` value of `S_OK` or `NOERROR` (`0`) indicates success, whereas if the high bit of the `HResult` value is set, it indicates failure or some type of error condition. Two functions in the Windows unit, `Succeeded()` and `Failed()`, take an `HResult` as a parameter and return a `BOOL`, indicating success or failure. Here's the syntax for calling these methods:

```
if Succeeded(FunctionThatReturnsHResult) then
  \\ continue as normal

if Failed(FunctionThatReturnsHResult) then
  \\ error condition code
```

Of course, checking the return value of every single function call can become tedious. Also, dealing with errors returned by functions undermines Delphi's exception-handling methods for error detection and recovery. For these reasons, the `ComObj` unit defines a procedure called `OleCheck()` that converts `HResult` errors to exceptions. The syntax for calling this method is

```
OleCheck(FunctionThatReturnsHResult);
```

This procedure can be quite handy, and it will clean up your ActiveX code considerably.

COM Objects and Class Factories

In addition to supporting one or more interfaces that descend from `IUnknown` and implementing reference counting for lifetime management, COM objects also have another special feature: They are created through special objects called *class factories*. Each COM class has an associated class factory responsible for creating instances of that COM class. Class factories are

special COM objects that support the `IClassFactory` interface. This interface is defined in the `ActiveX` unit as follows:

```
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;
```

The `CreateInstance()` method is called to create an instance of the class factory's associated COM object. The `unkOuter` parameter of this method references the controlling `IUnknown` if the object is being created as a part of an aggregate (aggregation is explained a bit later). The `iid` parameter contains the IID of the interface by which you want to manipulate the object. Upon return, the `obj` parameter will hold a pointer to the interface indicated by `iid`.

The `LockServer()` method is called to keep a COM server in memory, even though no clients may be referencing the server. The `fLock` parameter, when `True`, should increment the server's lock count. When `False`, `fLock` should decrement the server's lock count. When the server's lock count is `0` and there are no clients referencing the server, COM will unload the server.

TComObject and TComObjectFactory

Delphi provides two classes that encapsulate COM objects and class factories: `TComObject` and `TComObjectFactory`, respectively. `TComObject` contains the necessary infrastructure for supporting `IUnknown` and creation via `TComObjectFactory`. Likewise, `TComObjectFactory` supports `IClassFactory` and has the capability to create `TComObject` objects. You can easily generate a COM object using the COM Object Wizard found on the ActiveX page of the New Items dialog. Listing 23.1 shows pseudocode for the unit generated by this wizard, which illustrates the relationship between these classes.

LISTING 23.1 COM Server Unit Pseudocode

```
unit ComDemo;

interface

uses ComObj;

type
  TSomeComObject = class(TComObject, interfaces supported)
    class and interface methods declared here
  end;
```

continues

LISTING 23.1 Continued

```
implementation

uses ComServ;

TSomeComObject implementation here

initialization
  TComObjectFactory.Create(ComServer, TSomeComObject,
    CLSID_TSomeComObject, 'ClassName', 'Description');
end;
```

The `TComServer` descendant is declared and implemented like most VCL classes. What binds it to its corresponding `TComObjectFactory` object is the parameters passed to `TComObjectFactory`'s constructor `Create()`. The first constructor parameter is a `TComServer` object. You almost always will pass the global `ComServer` object declared in the `ComServ` unit in this parameter. The second parameter is the `TComObject` class you want to bind to the class factory. The third parameter is the CLSID of the `TComObject`'s COM class. The fourth and fifth parameters are the class name and description strings used to describe the COM class in the System Registry.

The `TComObjectFactory` instance is created in the initialization of the unit in order to ensure that the class factory will be available to create instances of the COM object as soon as the COM server is loaded. Exactly how the COM server is loaded depends on whether the COM server is an in-process server (a DLL) or an out-of-process server (an application).

In-Process COM Servers

In-process (or *in-proc*, for short) COM servers are DLLs that can create COM objects for use by the host application. This type of COM server is called *in process* because, as a DLL, it resides in the same process as the calling application. An in-proc server must export four standard entry-point functions:

```
function DllRegisterServer: HResult; stdcall;
function DllUnregisterServer: HResult; stdcall;
function DllGetClassObject (const CLSID, IID: TGUID; var Obj): HResult;
  stdcall;
function DllCanUnloadNow: HResult; stdcall;
```

Each of these functions is already implemented by the `ComServ` unit, so the only work to be done for your Delphi COM servers is to ensure that these functions are added to an `exports` clause in your project.

NOTE

A good example of a real-world application of in-process COM servers can be found in Chapter 24, "Extending the Windows Shell," which demonstrates how to create shell extensions.

DllRegisterServer()

The `DllRegisterServer()` function is called to register a COM server DLL with the System Registry. If you simply export this method from your Delphi application, as described earlier, VCL will iterate over all the COM objects in your application and register them with the System Registry. When a COM server is registered, it will make a key entry in the System Registry under

`HKEY_CLASSES_ROOT\CLSID\{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx}`

for each COM class, where the X's denote the CLSID of the COM class. For in-proc servers, an additional entry is created as a subkey of the preceding key called `InProcServer32`. The default value for this key is the full path to the in-proc server DLL. Figure 23.1 shows a COM server registered with the System Registry.

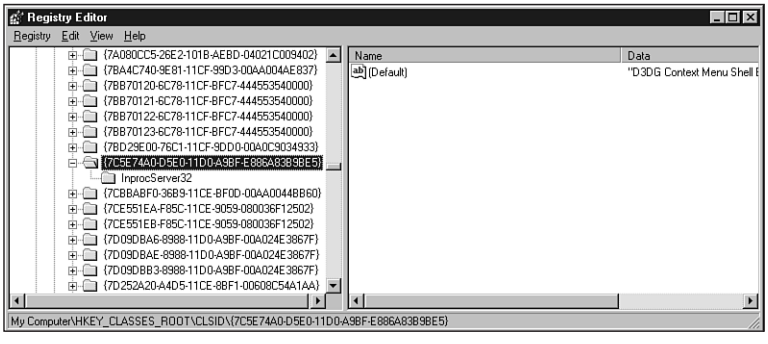

FIGURE 23.1
A COM server as shown in the Registry Editor.

DllUnregisterServer()

The `DllUnregisterServer()` function's job is simply to undo what is done by the `DllRegisterServer()` function. When called, it should remove all the entries in the System Registry made by `DllRegisterServer()`.

DllGetClassObject()

`DllGetClassObject()` is called by the COM engine in order to retrieve a class factory for a particular COM class. The `CLSID` parameter of this method is the CLSID of the type of COM class you want to create. The `IID` parameter holds the IID of the interface instance pointer you want to obtain for the class factory object (usually, `IClassFactory`'s interface ID is passed here). Upon successful return, the `Obj` parameter contains a pointer to the class factory interface denoted by `IID` that is capable of creating COM objects of the class type denoted by `CLSID`.

DllCanUnloadNow()

`DllCanUnloadNow()` is called by the COM engine to determine whether the COM server DLL is capable of being unloaded from memory. If there are references to any COM object within the DLL, this function should return `S_FALSE`, indicating that the DLL should not be unloaded. If none of the DLL's COM objects are in use, this method should return `S_TRUE`.

TIP

Even after all references to an in-proc server's COM objects have been freed, COM may not necessarily call `DllCanUnloadNow()` to begin the process of releasing the in-proc server DLL from memory. If you want to ensure that all unused COM server DLLs have been released from memory, call the `CoFreeUnusedLibraries()` API function, which is defined in the ActiveX units as follows:

```
procedure CoFreeUnusedLibraries; stdcall;
```

Creating an Instance of an In-Proc COM Server

To create an instance of a COM server in Delphi, use the `CreateComObject()` function, which is defined in the `ComObj` unit as follows:

```
function CreateComObject(const ClassID: TGUID): IUnknown;
```

The `ClassID` parameter holds the CLSID, which identifies the type of COM object you want to create. The return value of this function is the `IUnknown` interface of the requested COM object, or the function raises an exception if the COM object cannot be created.

`CreateComObject()` is a wrapper around the `CoCreateInstance()` COM API function. Internally, `CoCreateInstance()` calls the `CoGetClassObject()` API function to obtain an `IClassFactory` for the specified COM object. `CoCreateInstance()` does this by looking in the Registry for the COM class's `InProcServer32` entry in order to find the path to the in-proc server DLL, calling `LoadLibrary()` on the in-proc server DLL, and then calling the DLL's `DllGetClassObject()` function. After obtaining the `IClassFactory` interface pointer, `CoCreateInstance()` calls `IClassFactory.CreateInstance()` to create an instance of the specified COM class.

TIP

`CreateComObject()` can be inefficient if you need to create multiple objects from a class factory because it disposes of the `IClassFactory` interface pointer obtained by `CoGetClassObject()` after creating the requested COM object. In cases where you need to create multiple instances of the same COM object, you should call `CoGetClassObject()` directly and use `IClassFactory.CreateInstance()` to create multiple instances of the COM object.

NOTE

Before you can use any COM or OLE API functions, you must initialize the COM library using the `CoInitialize()` function. The single parameter to this function must be `nil`. To properly shut down the COM library, you should call the `CoUninitialize()` function as the last call to the OLE library. Calls are cumulative, so each call to `CoInitialize()` in your application must have a corresponding call to `CoUninitialize()`.

For applications, `CoInitialize()` is called automatically from `Application.Initialize()`, and `CoUninitialize()` is called automatically from the finalization of `ComObj`.

It's not necessary to call these functions from in-process libraries because their client applications are required to perform the initialization and uninitialization for the process.

Out-of-Process COM Servers

Out-of-process servers are executables that can create COM objects for use by other applications. The name comes from the fact that they do not execute from within the same process of the client but instead are executables that operate within the context of their own processes.

Registration

Like their in-proc cousins, out-of-process servers must also be registered with the System Registry. Out-of-process servers must make an entry under

`HKEY_CLASSES_ROOT\CLSID\{`*`xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx`*`}`

called `LocalServer32`, which identifies the full pathname of the out-of-process server executable.

Delphi applications' COM servers are registered in the `Application.Initialize()` method, which is usually the first line of code in an application's project file. If the `/regserver` command-line switch is passed to your application, `Application.Initialize()` will register

the COM classes with the System Registry and immediately terminate the application. Likewise, if the `/unregserver` command-line switch is passed, `Application.Initialize()` will unregister the COM classes with the System Registry and immediately terminate the application. If neither of these switches are passed, `Application.Initialize()` will register the COM classes with the System Registry and continue to run the application normally.

Creating an Instance of an Out-of-Process COM Server

On the surface, the method for creating instances of COM objects from out-of-process servers is the same as for in-proc servers: Just call `ComObj`'s `CreateComObject()` function. Behind the scenes, however, the process is quite different. In this case, `CoGetClassObject()` looks for the `LocalServer32` entry in the System Registry and invokes the associated application using the `CreateProcess()` API function. When the out-of-proc server application is invoked, the server must register its class factories using the `CoRegisterClassObject()` COM API function. This function adds an `IClassFactory` pointer to COM's internal table of active registered class objects. `CoGetClassObject()` can then obtain the requested COM class's `IClassFactory` pointer from this table to create an instance of the COM object.

Aggregation

You know now that interfaces are the basic building blocks of COM as well as that inheritance is possible with interfaces, but interfaces are entities without implementation. What happens, then, when you want to recycle the implementation of one COM object within another? COM's answer to this question is a concept called *aggregation*. Aggregation means that the containing (outer) object creates the contained (inner) object as part of its creation process, and the interfaces of the inner object are exposed by the outer. An object has to allow itself to operate as an aggregate by providing a means to forward all calls to its `IUnknown` methods to the containing object. For an example of aggregation within the context of VCL COM objects, you should take a look at the `TAggregatedObject` class in the `AxCtrls` unit.

Distributed COM

Introduced with Windows NT 4, Distributed COM (or *DCOM*) provides a means for accessing COM objects located on other machines on a network. In addition to remote object creation, DCOM also provides security facilities that allow servers to specify which clients have rights to create instances of which servers and what operations they may perform. Windows NT 4 and Windows 98 have built-in DCOM capability, but Windows 95 requires an add-on available on Microsoft's Web site (`http://www.microsoft.com`) to serve as a DCOM client.

You can create remote COM objects using the `CreateRemoteComObject()` function, which is declared in the `ComObj` unit as follows:

```
function CreateRemoteComObject(const MachineName: WideString;
  const ClassID: TGUID): IUnknown;
```

The first parameter, `MachineName`, to this function is a string representing the network name of the machine containing the COM class. The `ClassID` parameter specifies the CLSID of the COM class to be created. The return value for this function is the `IUnknown` interface pointer for the COM object specified in `CLSID`. An exception will be raised if the object cannot be created.

`CreateRemoteComObject()` is a wrapper around the `CoCreateInstanceEx()` COM API function, which is an extended version of `CoCreateInstance()` that knows how to create objects remotely.

Automation

Automation (formerly known as *OLE Automation*) provides a means for applications or DLLs to expose programmable objects for use by other applications. Applications or DLLs that expose programmable objects are referred to as *Automation servers*. Applications that access and manipulate the programmable objects contained within Automation servers are known as *Automation controllers*. Automation controllers are able to program the Automation server using a macro-like language exposed by the server.

Among the chief advantages to using Automation in your applications is its language-independent nature. An Automation controller is able to manipulate a server regardless of the programming language used to develop either component. Additionally, because Automation is supported at the operating system level, the theory is that you'll be able to leverage future advancements in this technology by using Automation today. If these things sound good to you, then read on. What follows is information on creating Automation servers and controllers in Delphi.

CAUTION

If you have an Automation project from Delphi 2 that you want to migrate to the current version of Delphi, you should be forewarned that the techniques for Automation changed drastically starting with Delphi 3. In general, you shouldn't mix Delphi 2's Automation unit, `OleAuto`, with the newer `ComObj` or `ComServ` units. If you want to compile a Delphi 2 Automation project in Delphi 5, the `OleAuto` unit remains in the `\Delphi5\lib\Delphi2` subdirectory for backward compatibility.

IDispatch

Automation objects are essentially COM objects that implement the `IDispatch` interface. `IDispatch` is defined in the `System` unit as shown here:

```
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;
```

The first thing you should know is that you don't have to understand the ins and outs of the `IDispatch` interface to take advantage of Automation in Delphi, so don't let this complicated interface alarm you. You generally don't have to interact with this interface directly because Delphi provides an elegant encapsulation of Automation, but the description of `IDispatch` in this section should provide you with a good foundation for understanding Automation.

Central to the function of `IDispatch` is the `Invoke()` method, so we'll start there. When a client obtains an `IDispatch` pointer for an Automation server, it can call the `Invoke()` method to execute a particular method on the server. The `DispID` parameter of this method holds a number, called a *dispatch ID,* that indicates which method on the server should be invoked. The `IID` parameter is unused. The `LocaleID` parameter contains language information. The `Flags` parameter describes what kind of method is to be invoked and whether it's a normal method or a put or get method for a property. The `Params` property contains a pointer to an array of `TDispParams`, which holds the parameters passed to the method. The `VarResult` parameter is a pointer to an `OleVariant`, which will hold the return value of the method that is invoked. `ExcepInfo` is a pointer to a `TExcepInfo` record that will contain error information if `Invoke()` returns `DISP_E_EXCEPTION`. Finally, if `Invoke()` returns `DISP_E_TYPEMISMATCH` or `DISP_E_PARAMNOTFOUND`, the `ArgError` parameter is a pointer to an integer that will contain the index of the offending parameter in the `Params` array.

The `GetIDsOfName()` method of `IDispatch` is called to obtain the dispatch ID of one or more method names given strings identifying those methods. The `IID` parameter of this method is unused. The `Names` parameter points to an array of `PWideChar` method names. The `NameCount` parameter holds the number of strings in the `Names` array. `LocaleID` contains language information. The last parameter, `DispIDs`, is a pointer to an array of `NameCount` integers, which `GetIDsOfName()` will fill in with the dispatch IDs for the methods listed in the `Names` parameter.

`GetTypeInfo()` retrieves the type information (type information is described next) for the Automation object. The `Index` parameter represents the type of information to obtain and should normally be `0`. The `LCID` parameter holds language information. Upon successful return, the `TypeInfo` parameter will hold an `ITypeInfo` pointer for the Automation object's type information.

The `GetTypeInfoCount()` method retrieves the number of type information interfaces supported by the Automation object in the `Count` parameter. Currently, `Count` will only contain two possible values: `0`, meaning the Automation object doesn't support type information, and `1`, meaning the Automation object does support type information.

Type Information

After you have spent a great deal of time carefully crafting an Automation server, it would be a shame if potential users of your server couldn't exploit its capabilities to the fullest because of lack of documentation on the methods and properties provided. Fortunately, Automation provides a means for helping avoid this problem by allowing developers to associate type information with Automation objects. This type information is stored in something called a *type library*, and an Automation server's type library can be linked to the server application or library as a resource or stored in an external file. Type libraries contain information about classes, interfaces, types, and other entities in a server. This information provides clients of the Automation server with the information needed to create instances of each of its classes and properly call methods on each interface.

Delphi generates type libraries for you when you add Automation objects to applications and libraries. Additionally, Delphi knows how to translate type library information into Object Pascal so that you can easily control Automation servers from your Delphi applications.

Late Versus Early Binding

The elements of Automation that you've learned about so far in this chapter deal with what's called *late binding*. Late binding is a fancy way to say that a method is called through `IDispatch`'s `Invoke()` method. It's called *late binding* because the method call isn't resolved until runtime. At compile time, an Automation method call resolves into a call to `IDispatch.Invoke()` with the proper parameters, and at runtime, `Invoke()` executes the Automation method. When you call an Automation method via a Delphi `Variant` or `OleVariant` type, you're using late binding because Delphi must call `IDispatch.GetIDsOfNames()` to convert the method name into a `DispID`, and then it can invoke the method by calling `IDispatch.Invoke()` with the `DispID`.

A common optimization of early binding is to resolve the `DispID`s of methods at compile time and therefore avoid the runtime calls to `GetIDsOfNames()` in order to invoke a method. This

optimization is often referred to as *ID binding,* and it is the convention used when you invoke methods via a Delphi `dispinterface` type.

Early binding occurs when the Automation object exposes methods by means of a custom interface descending from `IDispatch`. This way, controllers can call Automation objects directly through the vtable without going through `IDispatch.Invoke()`. Because the call is direct, a call to such as method will generally occur faster than a call through late binding. Early binding is used you when call a method using a Delphi `interface` type.

An Automation object that allows methods to be called both from `Invoke()` and directly from an `IDispatch` descendant interface is said to support a *dual interface*. Delphi-generated Automation objects always support a dual interface, and Delphi controllers allow methods to be called both through `Invoke()` and directly through an interface.

Registration

Automation objects must make all the same Registry entries as regular COM objects, but Automation servers typically also make an additional entry under

```
HKEY_CLASSES_ROOT\CLSID\{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx}
```

called `ProgID`, which provides a string identifier for the Automation class. Yet another Registry entry under `HKEY_CLASSES_ROOT\(ProgID string)` is made, which contains the CLSID of the Automation class in order to cross-reference back to the first Registry entry under `CLSID`.

Creating Automation Servers

Delphi makes it a fairly simple chore to create both out-of-process and in-process Automation servers. The process for creating an Automation server can be boiled down into four steps:

1. Create the application or DLL you want to automate. You can even use one of your existing applications as a starting point in order to spice it up with some automation. This is the only step where you'll see a real difference between creating in-process and out-of-process servers.
2. Create the Automation object and add it to your project. Delphi provides an Automation Object Expert to help this step go smoothly.
3. Add properties and methods to the Automation object by means of the type library. These are the properties and methods that will be exposed to Automation controllers.
4. Implement the methods generated by Delphi from your type library in your source code.

Creating an Out-of-Process Automation Server

This section walks you through the creation of a simple out-of-process Automation server. Start by creating a new project and placing a `TShape` and a `TEdit` component on the main form, as shown in Figure 23.2. Save this project as `Srv.dpr`.

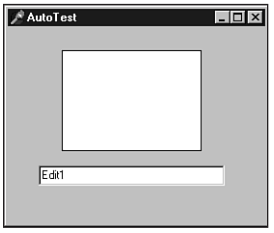

FIGURE 23.2
The main form of the Srv project.

Now add an Automation object to the project by selecting File, New from the main menu and choosing Automation Object from the ActiveX page of the New Items dialog, as shown in Figure 23.3. This will invoke the Automation Object Wizard shown in Figure 23.4.

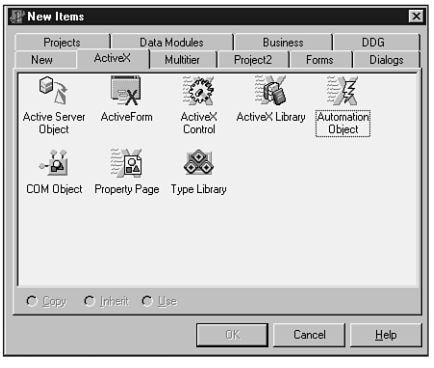

FIGURE 23.3
Adding a new Automation object.

FIGURE 23.4
The Automation Object Wizard.

In the Class Name field of the Automation Object Wizard dialog, you should enter the name you want to give the COM class for this Automation object. The wizard will automatically prepend a *T* to the class name when creating the Object Pascal class for the Automation object and an *I* to the class name when creating the primary interface for the Automation object. The Instancing combo box in the wizard can hold any one of these three values:

Value	*Description*
Internal	This OLE object will be used internal to the application only, and it will not be registered with the System Registry. External processes cannot access internal instanced Automation servers.
Single Instance	Each instance of the server can export only one instance of the OLE object. If a controller application requests another instance of the OLE object, Windows will start a new instance of the server application.
Multiple Instance	Each server instance can create and export multiple instances of the OLE object. In-process servers are always multiple instance.

When you complete the wizard's dialog, Delphi will create a new type library for your project (if one doesn't already exist) and add an interface and a coclass to the type library. Additionally, the wizard will generate a new unit in your project that contains the implementation of the Automation interface added to the type library. Figure 23.5 shows the type library editor immediately after the wizard's dialog is dismissed, and Listing 23.2 shows the implementation unit for the Automation object.

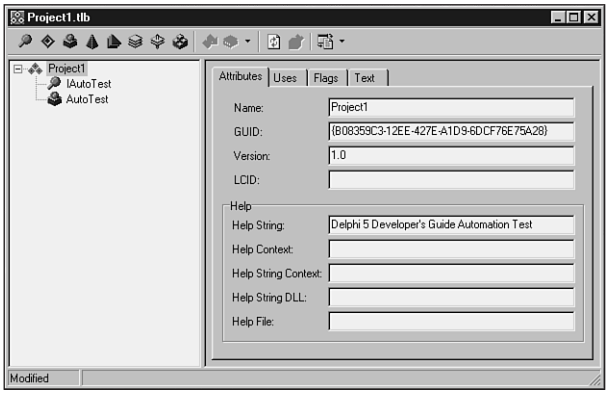

FIGURE 23.5
A new Automation project as shown in the type library editor.

LISTING 23.2 Automation Object Implementation Unit

```
unit TestImpl;

interface

uses
  ComObj, ActiveX, Srv_TLB;

type
  TAutoTest = class(TAutoObject, IAutoTest)
  protected
    { Protected declarations }
  end;

implementation

uses ComServ;

initialization
  TAutoObjectFactory.Create(ComServer, TAutoTest, Class_AutoTest,
    ciMultiInstance, tmApartment);
end.
```


The Automation object, `TAutoTest`, is a class that descends from `TAutoObject`. `TAutoObject` is the base class for all Automation servers. As you add methods to your interface by using the type library editor, new method skeletons will be generated in this unit that you will implement, thus forming the innards of your Automation object.

CAUTION

Again, be careful not to confuse Delphi 2's `TAutoObject` (from the `OleAuto` unit) with Delphi 5's `TAutoObject` (from the `ComObj` unit). The two are not compatible.

Similarly, the `automated` visibility specifier introduced in Delphi 2 is now mostly obsolete.

When the Automation object has been added to the project, you must add one or more properties or methods to the primary interface using the type library editor. For this project, the type library will contain properties to get and set the shape, color, and type as well as the edit control's text. For good measure, you'll also add a method that displays the current status of these properties in a dialog. Figure 23.6 shows the completed type library for the Srv project. Note especially the enumeration added to the type library (whose values are shown in the right pane) to support the `ShapeType` property.

NOTE

As you add properties and methods to Automation objects in the type library, keep in mind that the parameters and return values used for these properties and methods must be of Automation-compatible types. Types compatible with Automation include `Byte`, `SmallInt`, `Integer`, `Single`, `Double`, `Currency`, `TDateTime`, `WideString`, `WordBool`, `PSafeArray`, `TDecimal`, `OleVariant`, `IUnknown`, and `IDispatch`.

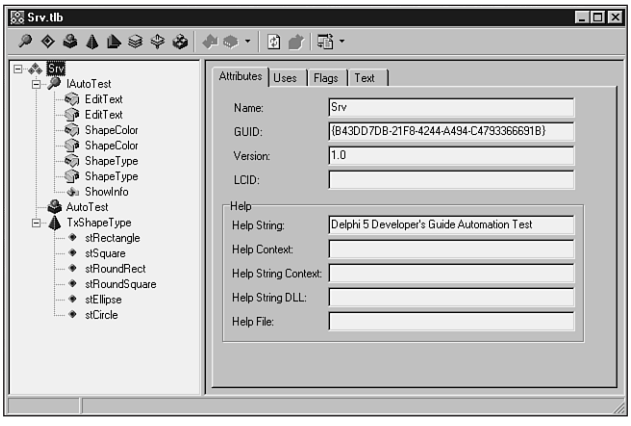

FIGURE 23.6
The completed type library.

When the type library has been completed, all that is left to do is fill in the implementation for each of the method stubs created by the type library editor. This unit is shown in Listing 23.3.

LISTING 23.3 The Completed Implementation Unit

```
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."';
```

continues

LISTING 23.3 Continued

```
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.
```

The `uses` clause for this unit contains a unit called `Srv_TLB`. This unit is the Object Pascal translation of the project type library, and it is shown in Listing 23.4.

LISTING 23.4 `Srv_TLB`: The Type Library File

```
unit Srv_TLB;

// ************************************************************************ //
// WARNING
// -------
// The types declared in this file were generated from data read from a
// Type Library. If this type library is explicitly or indirectly (via
// another type library referring to this type library) re-imported, or
// the 'Refresh' command of the Type Library Editor activated while
// editing the Type Library, the contents of this file will be regenerated
// and all manual modifications will be lost.
// ************************************************************************ //

// PASTLWTR : $Revision:   1.88  $
// File generated on 10/28/99 1:55:17 PM from Type Library described below

// ************************************************************************ //
// NOTE:
// Items guarded by $IFDEF_LIVE_SERVER_AT_DESIGN_TIME are used by
// properties which return objects that may need to be explicitly created
// via a function call prior to any access via the property. These items
// have been disabled in order to prevent accidental use from within the
// object inspector. You may enable them by defining
// LIVE_SERVER_AT_DESIGN_TIME or by selectively removing them from the
// $IFDEF blocks. However, such items must still be programmatically
// created via a method of the appropriate CoClass before they can be used
// ************************************************************************ //
// 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} // Unit must be compiled without type-checked pointers
interface

uses Windows, ActiveX, Classes, Graphics, OleServer, OleCtrls, StdVCL;

// ************************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// ************************************************************************//
const
  // TypeLibrary Major and minor versions
  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}';

// ************************************************************************//
// Declaration of Enumerations defined in Type Library
// ************************************************************************//
// Constants for enum TxShapeType
type
  TxShapeType = TOleEnum;
const
  stRectangle = $00000000;
  stSquare = $00000001;
  stRoundRect = $00000002;
  stRoundSquare = $00000003;
  stEllipse = $00000004;
  stCircle = $00000005;

type

// ************************************************************************//
// Forward declaration of types defined in TypeLibrary
```

continues

LISTING 23.4 Continued

```
// ************************************************************************//
  IAutoTest = interface;
  IAutoTestDisp = dispinterface;

// ************************************************************************//
// Declaration of CoClasses defined in Type Library
// (NOTE: Here we map each CoClass to its Default Interface)
// ************************************************************************//
  AutoTest = IAutoTest;


// ************************************************************************//
// Interface: IAutoTest
// Flags:     (4416) 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(Value: 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;
```

```
// ********************************************************************//
// The Class CoAutoTest provides a Create and CreateRemote method to
// create instances of the default interface IAutoTest exposed by
// the CoClass AutoTest. The functions are intended to be used by
// clients wishing to automate the CoClass objects exposed by the
// server of this typelibrary.
// ********************************************************************//
  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);
```

continues

LISTING 23.4 Continued

```
    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;

{$IFDEF LIVE_SERVER_AT_DESIGN_TIME}
// ********************************************************************//
// OLE Server Properties Proxy Class
// Server Object    : TAutoTest
// (This object is used by the IDE's Property Inspector to allow editing
//  of the properties of this server)
// ********************************************************************//
 TAutoTestProperties = class(TPersistent)
  private
    FServer:    TAutoTest;
    function    GetDefaultInterface: IAutoTest;
    constructor Create(AServer: TAutoTest);
  protected
    function  Get_EditText: WideString;
    procedure Set_EditText(const Value: WideString);
    function  Get_ShapeColor: OLE_COLOR;
    procedure Set_ShapeColor(Value: OLE_COLOR);
    function  Get_ShapeType: TxShapeType;
    procedure Set_ShapeType(Value: TxShapeType);
  public
    property DefaultInterface: IAutoTest read GetDefaultInterface;
  published
    property EditText: WideString read Get_EditText write Set_EditText;
    property ShapeColor: OLE_COLOR read Get_ShapeColor write
      Set_ShapeColor;
    property ShapeType: TxShapeType read Get_ShapeType write
      Set_ShapeType;
  end;
{$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
```

continues

LISTING 23.4 Continued

```
  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;
```

continues

LISTING 23.4 Continued

```
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.
```

Looking at this unit from the top down, you will notice that the type library version is specified first and then the GUID for the type library, `LIBID_Srv`, is declared. This GUID will be used when the type library is registered with the System Registry. Next, the values for the `TxShapeType` enumeration are listed. What's interesting about the enumeration is that the values are declared as constants rather than as an Object Pascal enumerated type. This is because type library enums are like C/C++ enums (and unlike Object Pascal) in that they don't have to start at the ordinal value zero or be sequential in value.

Next, in the Srv_TLB unit the IAutoTest interface is declared. In this interface declaration you'll see the properties and methods you created in the type library editor. Additionally, you'll see the Get_*XXX* and Set_*XXX* methods generated as the read and write methods for each of the properties.

Safecall

Safecall is the default calling convention for methods entered into the type library editor, as you can see from the IAutoTest declaration earlier. Safecall is actually more than a calling convention because it implies two things: First, it means that the method will be called using the safecall calling convention. Second, it means that the method will be encapsulated so that it returns an HResult value to the caller. For example, suppose you have a method that looks like this in Object Pascal:

```
function Foo(W: WideString): Integer; safecall;
```

This method actually compiles to code that looks something like this:

```
function Foo(W: WideString; out RetVal: Integer): HResult; stdcall;
```

The advantage of safecall is that it catches all exceptions before they flow back into the caller. When an unhandled exception is raised in a safecall method, the exception is handled by the implicit wrapper and converted into an HResult, which is returned to the caller.

Next in Srv_TLB is the dispinterface declaration for the Automation object: IAutoTestDisp. A dispinterface signals to the caller that Automation methods may be executed by Invoke() but does not imply a custom interface through which methods can be executed. Although the IAutoTest interface can be used by development tools that support early-binding Automation, IAutoTestDisp's dispinterface can be used by tools that support late binding.

The Srv_TLB unit then declares a class called CoAutoTest, which makes creation of the Automation object easy; just call CoAutoTest.Create() to create an instance of the Automation object.

Finally, Srv_TLB creates a class called TAutoTest that wraps the server into a component that can be placed on the palette. This feature, new to Delphi 5, is targeted more toward Automation servers that you import rather than new Automation servers that you create.

As mentioned earlier, you must run this application once to register it with the System Registry. Later in this chapter, you'll learn about the controller application used to manipulate this server.

Creating an In-Process Automation Server

Just as out-of-process servers start out as applications, in-process servers start out as DLLs. You can begin with an existing DLL or with a new DLL, which you can create by selecting DLL from the New Items dialog found under the File, New menu.

NOTE

If you're not familiar with DLLs, they are covered in depth in Chapter 9, "Dynamic Link Libraries." This chapter assumes that you have some knowledge of DLL programming.

As mentioned earlier, in order to serve as an in-process Automation server, a DLL must export four functions that are defined in the `ComServ` unit: `DllGetClassObject()`, `DllCanUnloadNow()`, `DllRegisterServer()`, and `DllUnregisterServer()`. Do this by adding these functions to the `exports` clause in your project file, as shown in the project file `IPS.dpr` in Listing 23.5.

LISTING 23.5 `IPS.dpr`—The Project File for an In-Process Server

```
library IPS;

uses
  ComServ;

exports
  DllRegisterServer,
  DllUnregisterServer,
  DllGetClassObject,
  DllCanUnloadNow;

begin
end.
```

The Automation object is added to the DLL project in the same manner as an executable project: through the Automation Object Wizard. For this project, you will add only one property and one method, as shown in the type library editor in Figure 23.7. The Object Pascal version of the type library, `IPS_TLB`, is shown in Listing 23.6.

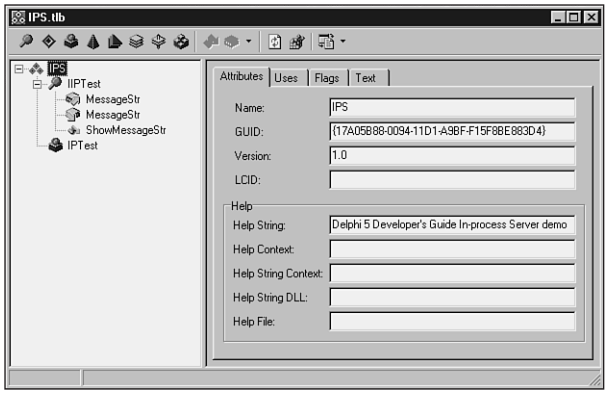

FIGURE 23.7
The IPS project in the type library editor.

LISTING 23.6 IPS_TLB.pas—The Type Library Import File for the In-Process Server Project

```
unit IPS_TLB;

// ************************************************************************ //
// WARNING
// -------
// The types declared in this file were generated from data read from a
// Type Library. If this type library is explicitly or indirectly (via
// another type library referring to this type library) re-imported, or the
// 'Refresh' command of the Type Library Editor activated while editing the
// Type Library, the contents of this file will be regenerated and all
// manual modifications will be lost.
// ************************************************************************ //

// PASTLWTR : $Revision:   1.79  $
// File generated on 8/14/99 11:37:16 PM from Type Library described below.

// ************************************************************************ //
// Type Lib: C:\work\d5dg\code\Ch23\Automate\IPS.tlb (1)
// 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
```

continues

LISTING 23.6 Continued

```
uses Windows, ActiveX, Classes, Graphics, OleServer, OleCtrls, StdVCL;

// ************************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// ************************************************************************//
const
  // TypeLibrary Major and minor versions
  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

// ************************************************************************//
// Forward declaration of types defined in TypeLibrary
// ************************************************************************//
  IIPTest = interface;
  IIPTestDisp = dispinterface;

// ************************************************************************//
// Declaration of CoClasses defined in Type Library
// (NOTE: Here we map each CoClass to its Default Interface)
// ************************************************************************//
  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;

// ************************************************************************//
// The Class CoIPTest provides a Create and CreateRemote method to
// create instances of the default interface IIPTest exposed by
// the CoClass IPTest. The functions are intended to be used by
// clients wishing to automate the CoClass objects exposed by the
// server of this typelibrary.
// ************************************************************************//
  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.
```


Clearly, this is a pretty simple Automation server, but it serves to illustrate the point. The `MessageStr` property can be set to a value and then shown with the `ShowMessageStr()` function. The implementation of the `IIPTest` interface resides in the unit `IPSMain.pas`, which is shown in Listing 23.7.

LISTING 23.7 `IPSMain.pas`—The Main Unit for the In-Process Server Project

```
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), 'Your string is...', 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,
    tmApartment);
end.
```

As you learned earlier in this chapter, in-process servers are registered differently than out-of-process servers; an in-process server's `DllRegisterServer()` function is called to register it

with the System Registry. The Delphi IDE makes this very easy: Select Run, Register ActiveX server from the main menu.

Creating Automation Controllers

Delphi makes it extremely easy to control Automation servers in your applications. Delphi also gives you a great amount of flexibility in how you want to control Automation servers, with options for early binding using interfaces or late binding using dispinterfaces or variants.

Controlling Out-of-Process Servers

The Control project is an Automation controller that demonstrates all three types of Automation (interfaces, dispinterface, and variants). Control is the controller for the Srv Automation server application from earlier in this chapter. The main form for this project is shown in Figure 23.8.

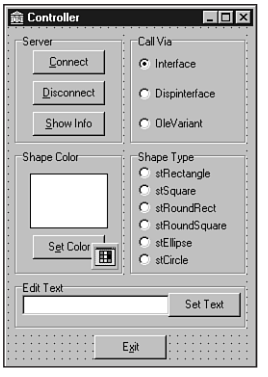

FIGURE 23.8
The main form for the Control project.

When the Connect button is clicked, the Control application connects to the server in several different ways with the following code:

```
FIntf := CoAutoTest.Create;
FDispintf := CreateComObject(Class_AutoTest) as IAutoTestDisp;
FVar := CreateOleObject('Srv.AutoTest');
```

This code shows `interface`, `dispinterface`, and `OleVariant` variables, each creating an instance of the Automation server in different ways. What's interesting about these different techniques is that they're almost totally interchangeable. For example, the following code is also correct:

```
FIntf := CreateComObject(Class_AutoTest) as IAutoTest;
FDispintf := CreateOleObject('Srv.AutoTest') as IAutoTestDisp;
FVar := CoAutoTest.Create;
```

Listing 23.8 shows the `Ctrl` unit, which contains the rest of the source code for the Automation controller. Notice that the application allows you to manipulate the server using either the `interface`, `dispinterface`, or `OleVariant`.

LISTING 23.8 `Ctrl.pas`—The Main Unit for the Controller Project for the Out-of-Process Server Project

```
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
    { Private declarations }
    FIntf: IAutoTest;
    FDispintf: IAutoTestDisp;
    FVar: OleVariant;
```

```
    procedure SetControls;
    procedure EnableControls(DoEnable: Boolean);
  public
    { Public declarations }
  end;

var
  ControlForm: TControlForm;

implementation

{$R *.DFM}

uses ComObj;

procedure TControlForm.SetControls;
// Initializes the controls to the current server values
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;
```

continues

LISTING 23.8 Continued

```
  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.
```


Another interesting thing this code illustrates is how easy it is to disconnect from an Automation server: Interfaces and dispinterfaces can be set to `nil`, and variants can be set to `Unassigned`. Of course, the Automation server will also be released when the `Control` application is closed, as a part of the normal finalization of these lifetime-managed types.

TIP

Interfaces will almost always perform better than dispinterfaces and variants, so you should always use interfaces to control Automation servers when available.

Variants rank last in terms of performance because, at runtime, an Automation call through a variant must call `GetIDsOfNames()` to convert a method name into a dispatch ID before it can execute the method with a call to `Invoke()`.

The performance of dispinterfaces is in between that of an interface and that of a variant. "But why," you might ask, "is the performance different if variants and dispinterfaces both use late binding?" The reason for this is that dispinterfaces take advantage of an optimization called *ID binding,* which means that the dispatch IDs of

continues

methods are known at compile time, so the compiler doesn't need to generate a runtime call to `GetIDsOfName()` prior to calling `Invoke()`. Another, perhaps more obvious, advantage of dispinterfaces over variants is that dispinterfaces allow for the use of CodeInsight for easier coding, whereas this is not possible using variants.

Figure 23.9 shows the Control application controlling the Srv server.

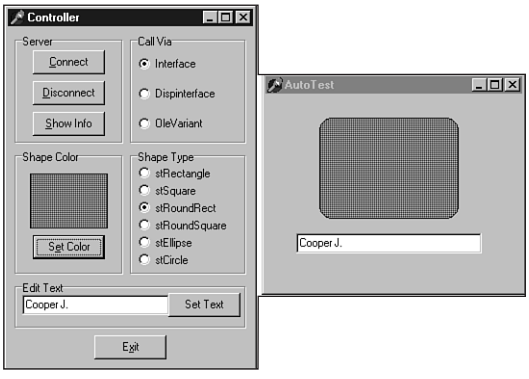

FIGURE 23.9
Automation controller and server.

Controlling In-Process Servers

The technique for controlling an in-process server is no different than that for controlling its out-of-process counterpart. Just keep in mind that the Automation controller is now executing within your own process space. This means performance will be a bit better than with out-of-process servers, but it also means that a crash in the Automation server can take down your application.

Now you'll look at a controller application for the in-process Automation server created earlier in this chapter. In this case, we'll use only the interface for controlling the server. This is a pretty simple application, and Figure 23.10 shows the main form for the IPCtrl project. The code in Listing 23.9 is `IPCMain.pas`, the main unit for the IPCtrl project.

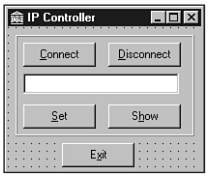

FIGURE 23.10
The IPCtrl project's main form.

LISTING 23.9 `IPCMain.pas`—The Main Unit for the Controller Project for the In-Process Server Project

```
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
    { Private declarations }
    IPTest: IIPTest;
    procedure EnableControls(DoEnable: Boolean);
  public
    { Public declarations }
  end;

var
  IPCForm: TIPCForm;
```

continues

LISTING 23.9 Continued

```
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.
```

Remember to ensure that the server has been registered prior to attempting to run IPCtrl. You can do this in several ways: Using Run, Register ActiveX Server from the main menu while the

IPS project is loaded, using the Windows `RegSvr32.exe` utility, and using the `TRegSvr.exe` tool that comes with Delphi. Figure 23.11 shows this project in action controlling the IPS server.

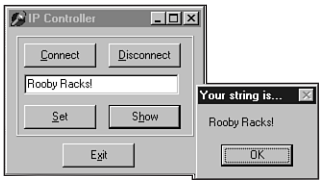

FIGURE 23.11
IPCtrl controlling the IPS server.

Advanced Automation Techniques

In this section, our goal is to get you up to speed on some of the more advanced features of Automation that the wizards never told you about. Topics such as Automation events, collections, type library gotchas, and low-level language support for COM are all covered. Rather than devote more time to talking about this stuff, let's jump right in and do it!

Automation Events

We Delphi programmers have long taken events for granted. You drop a button, you double-click `OnClick` in the Object Inspector, and you write some code. No big deal. Even from the control writer's point of view, events are a snap. You create a new method type, add a field and published property to your control, and you're good to go. For Delphi COM developers, however, events can be scary. Many Delphi COM developers avoid events altogether simply because they "don't have time to learn all that mumbo jumbo." If you fall into that group, you'll be happy to know that working with events is actually not very difficult thanks to some nice built-in support provided by Delphi. Although all the new terms associated with Automation events can add an air of complexity, in this section I hope to demystify events to the point where you think, "Oh, is that all they are?"

What Are Events?

Put simply, events provide a means for a server to call back into a client to provide some information. Under a traditional client/server model, the client calls the server to perform an action or obtain some data, the server executes the action or obtains the data, and control returns to the client. This model works fine for most things, but it breaks down when the event in which the client is interested is asynchronous in nature or is driven by a user interface entry. For example, if the client sends the server a request to download a file, the client probably doesn't want to sit around and wait for the thing to download before it can continue processing (especially

over a high-latency connection such as a modem). A better model would be for the client to issue the instruction to the server and continue to go about its business until the server notifies the client about the completion of the file download. Similarly, a user interface entry, such as a button click, is a good example of when the server needs to notify the client using an event mechanism. The client obviously can't call a method on the server that waits around until some button is clicked.

Generally speaking, the server is responsible for defining and firing events, whereas the client is normally responsible for connecting itself to and implementing events. Of course, given such a loose definition, there is room to haggle, and consequently Delphi and Automation provide two very different approaches to the idea of events. Drilling down into each of these models will help put things into perspective.

Events in Delphi

Delphi follows the KISS (keep it simple, stupid!) methodology when it comes to events. Events are implemented as method pointers—these pointers can be assigned to some method in the application and are executed when such a method is called via the method pointer. As an illustration, consider the everyday application-development scenario of an application that needs to handle an event on a component. If you look at the situation abstractly, the "server" in this case would be a component, which defines and fires the event. The "client" is the application that employs the component, because it connects to the event by assigning some specific method name to the event method pointer.

Although this simple event model is one of the things that make Delphi elegant and easy to use, it definitely sacrifices some power for the sake of usability. For example, there is no built-in way to allow multiple clients to listen for the same event (this is called *multicasting*). Also, there is no way to dynamically obtain a type description for an event without writing some RTTI code (which you probably shouldn't be using in an application anyway due to its version-specific nature).

Events in Automation

Whereas the Delphi event model is simple yet limited, the Automation event model is powerful but more complex. As a COM programmer, you may have guessed that events are implemented in Automation using interfaces. Rather than existing on a per-method basis, events exist only as part of an interface. This interface is often called an *events interface* or an *outgoing interface*. It's called *outgoing* because it is not implemented by the server like other interfaces but is instead implemented by clients of the server, and methods of the interface will be called outward from the server to the client. Like all interfaces, event interfaces have associated with them corresponding *interface identifications* (IIDs) that uniquely identify them. Also, the description of the events interface is found in the type library of an Automation object, tied to the Automation object's coclass like other interfaces.

Servers needing to surface event interfaces to clients must implement the `IConnectionPointContainer` interface. This interface is defined in the ActiveX unit as follows:

```
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;
```

In COM parlance, a *connection point* describes the entity that provides programmatic access to an outgoing interface. If a client needs to determine whether a server supports events, all it has to do is `QueryInterface` for the `IConnectionPointContainer` interface. If this interface is present, the server is capable of surfacing events. The `EnumConnectionPoints()` method of `IConnectionPointContainer` enables clients to iterate over all the outgoing interfaces supported by the server. Clients may use the `FindConnectionPoint()` method to obtain a specific outgoing interface.

You'll notice that `FindConnectionPoint()` provides an `IConnectionPoint` that represents an outbound interface. `IConnectionPoint` is also defined in the `ActiveX` unit, and it looks like this:

```
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;
```

The `GetConnectionInterface()` method of `IConnectionPoint` provides the IID of the outgoing interface supported by this connection point. The `GetConnectionPointContainer()` method provides the `IConnectionPointContainer` (described earlier), which manages this connection point. The `Advise` method is the interesting one. `Advise()` is the method that actually does the magic of hooking up the outgoing events on the server to the `events` interface implemented by the client. The first parameter to this method is the client's implementation of the `events` interface, and the second parameter will receive a cookie that identifies this particular connection. `Unadvise()` simply disconnects the client/server relationship established by

`Advise()`. `EnumConnections` enables the client to iterate over all currently active connections—that is, all connections that have called `Advise()`.

Because of the obvious confusion that can arise if we describe the participants in this relationship as simply *client* and *server*, Automation defines some different nomenclature that enables us to unambiguously describe who is who. The implementation of the outgoing interface contained within the client is called a *sink*, and the server object that fires events to the client is referred to as the *source*.

What is hopefully clear in all this is that Automation events have a couple of advantages over Delphi events. Namely, they can be multicast, because `IConnectionPoint.Advise()` can be called more than once. Also, Automation events are self-describing (via the type library and the enumeration methods), so they can be manipulated dynamically.

Automation Events in Delphi

Okay, all this technical stuff is well and good, but how do we actually make Automation events work in Delphi? Glad you asked. At this point, we will create an Automation server application that exposes an outgoing interface and a client that implements a sink for the interface. Bear in mind, too, that you don't need to be an expert in connection points, sinks, sources, and whatnot in order to get Delphi to do what you want. However, it does help you in the long run when you understand what goes on behind the wizard's curtain.

The Server

The first step in creating the server is to create a new application. For purposes of this demo, we will create a new application containing one form with a client-aligned `TMemo`, as shown in Figure 23.12.

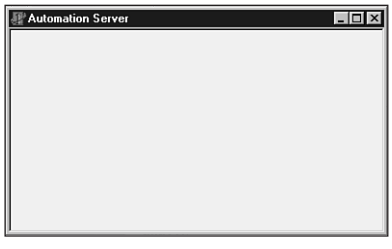

FIGURE 23.12
Automation Server with the Events main form.

Next, we will add an Automation object to this application by selecting File, New, ActiveX, Automation Object from the main menu. This invokes the Automation Object Wizard (refer to Figure 23.4).

Note the Generate Event Support Code option on the Automation Object Wizard. This box must be selected because it will generate the code necessary to expose an outgoing interface on the Automation object. It will also create the outgoing interface in the type library. After selecting OK in this dialog, we are presented with the Type Library Editor window. Both the Automation interface and the outgoing interface are already present in the type library (named `IServerWithEvents` and `IServerWithEventsEvents`, respectively). `AddText()` and `Clear()` methods have been added to the `IServerWithEvents` interface, and `OnTextChanged()` and `OnClear()` methods have been added to the `IServerWithEventsEvents` interface.

As you might guess, `Clear()` will clear the contents of the memo, and `AddText()` will add another line of text to the memo. The `OnTextChanged()` event will fire when the contents of the memo change, and the `OnClear()` event will fire when the memo is cleared. Notice also that `AddText()` and `OnTextChanged()` each have one parameter of type `WideString`.

The first thing to do is implement the `AddText()` and `Clear()` methods. The implementation for these methods is shown here:

```
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;
```


You should be familiar with all this code except perhaps the last line of `Clear()`. This code ensures that there is a client sink advised on the event by checking for `nil`; then it first fires the event simply by calling `OnClear()`.

To set up the `OnTextChanged()` event, we first have to handle the `OnChange` event of the memo. We will do this by inserting a line of code into the `Initialized()` method of `TServerWithEvents` that points the event to the method in `TServerWithEvents`:

```
MainForm.Memo.OnChange := MemoChange;
```

The `MemoChange()` method is implemented as follows:

```
procedure TServerWithEvents.MemoChange(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnTextChanged((Sender as TMemo).Text);
end;
```

This code also checks to ensure a client is listening; then it fires the event, passing the memo's text as the parameter.

Believe it or not, that sums up the implementation of the server! Now on to the client.

The Client

The client is an application with one form that contains a `TEdit`, `TMemo`, and three `TButton` components, as shown in Figure 23.13.

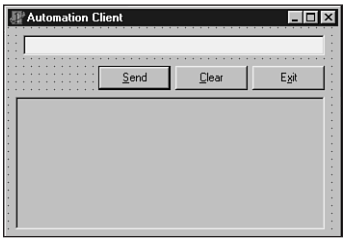

FIGURE 23.13
The Automation Client main form.

In the main unit for the client application, the `Server_TLB` unit has been added to the `uses` clause so that we have access to the types and methods contained within that unit. The main form object, `TMainForm`, of the client application will contain a field that references the server called `FServer` of type `IServerWithEvents`. We will create an instance of the server in `TMainForm`'s constructor using the helper class found in `Server_TLB`, like this:

```
FServer := CoServerWithEvents.Create;
```

The next step is to implement the event sink class. Because this class will be called by the server via Automation, it must implement `IDispatch` (and therefore `IUnknown`). The type declaration for this class is shown here:

```
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;
```

Most of the methods of `IUnknown` and `IDispatch` are not implemented, with the notable exceptions of `IUnknown.QueryInterface()` and `IDispatch.Invoke()`. These will be discussed in turn.

The `QueryInterface()` method for `TEventSink` is implemented as shown here:

```
function TEventSink.QueryInterface(const IID: TGUID; out Obj): HResult;
begin
  // First look for my own implementation of an interface
  // (I implement IUnknown and IDispatch).
  if GetInterface(IID, Obj) then
    Result := S_OK
  // Next, if they are looking for outgoing interface, recurse to return
  // our IDispatch pointer.
  else if IsEqualIID(IID, IServerWithEventsEvents) then
    Result := QueryInterface(IDispatch, Obj)
  // For everything else, return an error.
  else
    Result := E_NOINTERFACE;
end;
```

Essentially, this method returns an instance only when the requested interface is `IUnknown`, `IDispatch`, or `IServerWithEventsEvents`.

Here's the `Invoke` method for `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
        // First parameter is new string
        V := OleVariant(TDispParams(Params).rgvarg^[0]);
        FController.OnServerMemoChanged(V);
      end;
    2: FController.OnClear;
  end;
end;
```

`TEventSink.Invoke()` is hard-coded for methods having DispID 1 or DispID 2, which happen to be the DispIDs chosen for `OnTextChanged()` and `OnClear()`, respectively, in the server application. `OnClear()` has the most straightforward implementation: It simply calls the client main form's `OnClear()` method in response to the event. The `OnTextChanged()` event is a little trickier: This code pulls the parameter out of the `Params.rgvarg` array, which is passed in as a parameter to this method, and passes it through to the client main form's `OnServerMemoChanged()` method. Note that because the number and type of parameters are known, we are able to make simplifying assumptions in the source code. If you're clever, it is possible to implement `Invoke()` in a generic manner such that it figures out the number and types of parameters and pushes them onto the stack or into registers prior to calling the appropriate function. If you'd like to see an example of this, take a look at the `TOleControl.InvokeEvent()` method in the `OleCtrls` unit. This method represents the event-sinking logic for the ActiveX control container.

The implementation for `OnClear()` and `OnServerMemoChanged()` manipulate the contents of the client's memo. They are shown here:

```
procedure TMainForm.OnServerMemoChanged(const NewText: string);
begin
  Memo.Text := NewText;
end;

procedure TMainForm.OnClear;
begin
  Memo.Clear;
end;
```

The final piece of the puzzle is to connect the event sink to the server's source interface. This is easily accomplished using the `InterfaceConnect()` function found in the `ComObj` unit, which we will call from the main form's constructor, like so:

```
InterfaceConnect(FServer, IServerWithEventsEvents, FEventSink, FCookie);
```

The first parameter to this function is a reference to the source object. The second parameter is the IID of the outgoing interface. The third parameter holds the event sink interface. The fourth and final parameter is the cookie, and it is a reference parameter that will be filled in by the callee.

To be a good citizen, you should also clean up properly by calling `InterfaceDisconnect()` when you are finished playing with events. This is done in the main form's destructor:

```
InterfaceDisconnect(FEventSink, IServerWithEventsEvents, FCookie);
```

The Demo

Now that the client and server are written, we can see them in action. Be sure to run and close the server once (or run it with the `/regserver` switch) to ensure it is registered before attempting to run the client. Figure 23.14 shows the interactions between client, server, source, and sink.

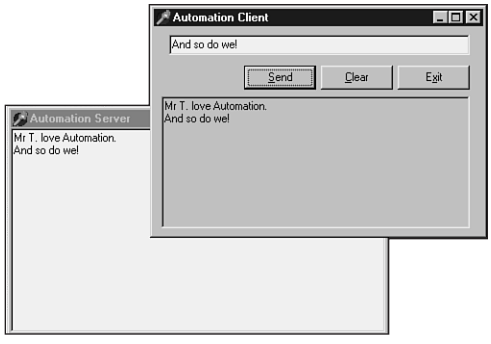

FIGURE 23.14
The Automation client manipulating the server and receiving events.

Events with Multiple Sinks

Although the technique just described works great for firing events back to a single client, it doesn't work so well when multiple clients are involved. You will often find yourself in situations where there are multiple clients connecting to your server, and you need to fire events back to all clients. Fortunately, you need just a little bit more code to add this type of functionality. In order to fire events back to multiple clients, you must write code that enumerates over each advised connection and calls the appropriate method on the sink. This can be done by making several modifications to the previous example.

First things first. In order to support multiple client connections on a connection point, we must pass `ckMulti` in the `Kind` parameter of `TConnectionPoints.CreateConnectionPoint()`. This method is called from the Automation object's `Initialize()` method, as shown here:

```
FConnectionPoints.CreateConnectionPoint(AutoFactory.EventIID, ckMulti,
  EventConnect);
```

Before connections can be enumerated, we need to obtain a reference to `IConnectionPointContainer`. From `IConnectionPointContainer`, we can obtain the `IConnectionPoint` representing the outgoing interface, and using the

IConnectionPoint.EnumConnections() method, we can obtain an IEnumConnections interface that can be used to enumerate the connections. All this logic is encapsulated into the following method:

```
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;
```

After the enumerator interface has been obtained, calling the sink for each client is just a matter of iterating over each connection. This logic is demonstrated in the following code, which fires the OnTextChanged() event:

```
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;
```

Finally, in order to enable clients to connect to a single active instance of the Automation object, we must call the RegisterActiveObject() COM API function. This function accepts as parameters an IUnknown for the object, the CLSID of the object, a flag indicating whether the registration is strong (the server should be AddRef-ed) or weak (do not AddRef the server), and a handle that is returned by reference:

```
RegisterActiveObject(Self as IUnknown, Class_ServerWithEvents,
    ACTIVEOBJECT_WEAK, FObjRegHandle);
```

Listing 23.10 shows the complete source code for the ServAuto unit, which ties all these tidbits together.

LISTING 23.10 ServAuto.pas

```
unit ServAuto;

interface

uses
  ComObj, ActiveX, AxCtrls, Server_TLB;

type
  TServerWithEvents = class(TAutoObject, IConnectionPointContainer,
    IServerWithEvents)
private
    { Private declarations }
    FConnectionPoints: TConnectionPoints;
    FObjRegHandle: Integer;
    procedure MemoChange(Sender: TObject);
  protected
    { Protected declarations }
    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);  // Make sure I'm removed from ROT
end;

procedure TServerWithEvents.Initialize;
begin
  inherited Initialize;
  FConnectionPoints := TConnectionPoints.Create(Self);
  if AutoFactory.EventTypeInfo <> nil then
    FConnectionPoints.CreateConnectionPoint(AutoFactory.EventIID, ckMulti,
      EventConnect);
```

continues

LISTING 23.10 Continued

```
  // Route main form memo's OnChange event to MemoChange method:
  MainForm.Memo.OnChange := MemoChange;
  // Register this object with COM's Running Object Table (ROT) so other
  // clients can connect to this instance.
  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.
```

On the client side, a small adjustment needs to be made in order enable clients to connect to an active instance if it is already running. This is accomplished using the `GetActiveObject` COM API function, as shown here:

```
procedure TMainForm.FormCreate(Sender: TObject);
var
  ActiveObj: IUnknown;
begin
  // Get active object if it's available, or create anew if not
  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;
```

Figure 23.15 shows several clients receiving events from a single server.

Automation Collections

Let's face it: We programmers are obsessed with bits of software code that serve as containers for other bits of software code. Think about it—whether it's an array, a `TList`, a `TCollection`, a template container class for you C++ folks, or a Java vector, it seems that we're always in search of the proverbial better mousetrap for software objects that hold other software objects. If you consider the time invested over the years in this pursuit for the perfect container class, it is clear that this is an important problem in the minds of developers. And why not? This logical separation of container and contained entities helps us better organize our algorithms and maps to the real world rather nicely (a basket can contain eggs, a pocket can contain coins, a parking lot can contain autos, and so on). Whenever you learn a new language or development model,

you have to learn "their way" of managing groups of entities. This leads to my point: Like any other software development model, COM also has its ways for managing these kinds of groups of entities, and to be an effective COM developer, we must learn how to master these things.

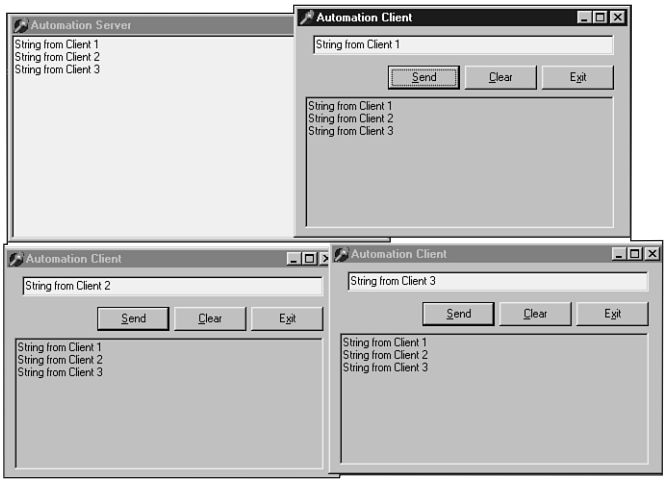

FIGURE 23.15
Several clients manipulating the same server and receiving events.

When we work with the `IDispatch` interface, COM specifies two primary methods by which we represent the notion of containership: arrays and collections. If you've done a bit of Automation or ActiveX control work in Delphi, you will probably already be familiar with arrays. You can easily create automation arrays in Delphi by adding an array property to your `IDispatch` descendant interface or dispinterface, as shown in the following example:

```
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;
```

Arrays are useful in many circumstances, but they pose some limitations. For example, arrays make sense when you have data that can be accessed in a logical, fixed-index manner, such as the strings in an `IStrings`. However, if the nature of the data is such that individual items are frequently deleted, added, or moved, an array is a poor container solution. The classic example is a group of active windows. Because windows are constantly being created, destroyed, and changing z-order, there is no solid criteria for determining the order in which the windows should appear in the array.

Collections are designed to solve this problem by allowing you to manipulate a series of elements in a manner that doesn't imply any particular order or number of items. Collections are unusual because there isn't really a *collection* object or interface, but a collection is instead represented as a custom `IDispatch` that follows a number of rules and guidelines. The following rules must be adhered to in order for an `IDispatch` to qualify as a collection:

- Collections must contain a `_NewEnum` property that returns the `IUnknown` for an object that supports the `IEnumVARIANT` interface, which will be used to enumerate the items in the collection. Note that the name of this property must be preceded with an underscore, and this property must be marked as *restricted* in the type library. The DispID for the `_NewEnum` property must be `DISPID_NEWENUM` (`-4`), and it will be defined as follows in the Delphi type library editor:

  ```
  function _NewEnum: IUnknown [propget, dispid $FFFFFFFC, restricted];
  safecall;
  ```

- Languages such as Visual Basic that support the `For Each` construct will use this method to obtain the `IEnumVARIANT` interface needed to enumerate collection items. More on this later.
- Collections must contain an `Item()` method that returns an element from the collection based on the index. The DispID for this method must be `0`, and it should be marked with the *default collection element* flag. If we were to implement a collection of `IFoo` interface pointers, the definition for this method in the type library editor might look something like this:

  ```
  function Item(Index: Integer): IFoo [propget, dispid $00000000,
    defaultcollelem]; safecall;
  ```

  Note that it is also acceptable for the `Index` parameter to be an `OleVariant` so that an `Integer`, `WideString`, or some other type of value can index the item in question.
- Collections must contain a `Count` property that returns the number of items in the collection. This method would typically be defined in the type library editor as this:

  ```
  function Count: Integer [propget, dispid $00000001]; safecall;
  ```

In addition to the aforementioned rules, you should also follow these guidelines when creating your own collection objects:

- The property or method that returns a collection should be named with the plural of the name of the items in the collection. For example, if you had a property that returned a collection of listview items, the property name would probably be `Items`, whereas the name of the item in the collection would be `Item`. Likewise, an item called `Foot` would be contained in a collection property called `Feet`. In the rare case that the plural and singular of a word are the same (a collection of fish or deer, for example), the collection property name should be the name of the item with "`Collection`" tacked on the end (`FishCollection` or `DeerCollection`).

- Collections that support adding of items should do so using a method called `Add()`. The parameters for this method vary depending on the implementation, but you may want to pass parameters that indicate the initial position of the new item within the collection. The `Add()` method normally returns a reference to the item added to the collection.
- Collections that support deleting of items should do so using a method called `Remove()`. This method should take one parameter that identifies the index of the item being deleted, and this index should behave semantically in the same manner as the `Item()` method.

A Delphi Implementation

If you've ever created ActiveX controls in Delphi, you may have noticed that there are fewer controls listed in the combo box in the ActiveX Control Wizard than there are on the IDE's component palette. This is because Inprise prevents some controls showing in the list using the `RegisterNonActiveX()` function. One such control that is available on the palette but not in the wizard is the `TListView` control found on the Win32 page of the palette. The reason the `TListView` control isn't shown in the wizard is because the wizard doesn't know what to do with its `Items` property, which is of type `TListItems`. Because the wizard doesn't know how to wrap this property type in an ActiveX control, the control is simply excluded from the wizard's list rather than allowing the user to create an utterly useless ActiveX control wrapper of a control.

However, in the case of `TListView`, `RegisterNonActiveX()` is called with the `axrComponentOnly` flag, which means that a descendent of `TListView` will show up in the ActiveX Control Wizard's list. By taking the minor detour of creating a do-nothing descendent of `TListView` called `TListView2` and adding it to the palette, we can then create an ActiveX control that encapsulates the listview control. Of course, then we are faced with the same problem of the wizard not generating wrappers for the `Items` property and having a useless ActiveX control. Fortunately, ActiveX control writing doesn't have to stop at the wizard-generated code, and we are free to wrap the `Items` property ourselves at this point in order to make the control useful. As you might be beginning to suspect, a collection is the perfect way to encapsulate the `Items` property of the `TListView`.

In order to implement this collection of listview items, we must create new objects representing the item and the collection and add a new property to the ActiveX control default interface that returns a collection. We will begin by defining the object representing an item, which we will call `ListItem`. The first step to creating the `ListItem` object is to create a new Automation object using the icon found on the ActiveX page of the New Items dialog. After creating the object, we can fill out the properties and methods for this object in the type library editor. For the purposes of this demonstration, we will add properties for the `Caption`, `Index`, `Checked`, and `SubItems` properties of a listview item. Similarly, we will create yet another new Automation object for the collection itself. This Automation object is called `ListItems`, and it is provided with the `_NewEnum`, `Item()`, `Count()`, `Add()`, and `Remove()` methods mentioned

earlier. Finally, we will add a new property to the default interface of the ActiveX control called `Items` that returns a collection.

After the interfaces for `IListItem` and `IListItems` are completely defined in the type library editor, a little manual tweaking should be done in the implementation files generated for these objects. Specifically, the default parent class for a new automation object is `TAutoObject`; however, these objects will only be created internally (that is, not from a factory), so we will manually change the ancestor to `TAutoInfObject`, which is more appropriate for internally created automation objects. Also, because these objects won't be created from a factory, we will remove from the units the initialization code that creates the factories because it is not needed.

Now that the entire infrastructure is properly set up, it is time to implement the `ListItem` and `ListItems` objects. The `ListItem` object is the most straightforward because it is a pretty simple wrapper around a listview item. The code for the unit containing this object is shown in Listing 23.11.

LISTING 23.11 The Listview Item Wrapper

```
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;
```

continues

LISTING 23.11 Continued

```
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.
```

Note that `ComCtrls.TListItem()` is being passed into the constructor to serve as the listview item to be manipulated by this Automation object.

The implementation for the `ListItems` collection object is just a bit more complex. First, because the object must be able to provide an object supporting `IEnumVARIANT` in order to implement the `_NewEnum` property, `IEnumVARIANT` is supported directly in this object. Therefore, the `TListItems` class supports both `IListItems` and `IEnumVARIANT`. `IEnumVARIANT` contains four methods, which are described in Table 23.1.

TABLE 23.1 `IEnumVARIANT` Methods

Method	*Purpose*
`Next`	Retrieves the next *n* number of items in the collection
`Skip`	Skips over *n* items in the collection
`Reset`	Resets current item back to the first item in the collection.
`Clone`	Creates a copy of this `IEnumVARIANT`

The source code for the unit containing the `ListItems` object is shown in Listing 23.12.

LISTING 23.12 The Listview Items Wrapper

```
unit LVItems;

interface

uses
  ComObj, Windows, ActiveX, ComCtrls, LVCtrl_TLB;

type
  TListItems = class(TAutoIntfObject, IListItems, IEnumVARIANT)
  private
    FListItems: ComCtrls.TListItems;
    FEnumPos: Integer;
  protected
    { IListItems methods }
    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;
    { IEnumVariant methods }
    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;
```

continues

LISTING 23.12 Continued

```
  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);
      // trick to prevent variant from being garbage collected, since it needs
      // to stay alive because it is party of the elt array
      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;
```

continues

LISTING 23.12 Continued

```
begin
  Inc(FEnumPos, celt);
  Result := S_OK;
end;

end.
```

The only method in this unit with a nontrivial implementation is the `Next()` method. The `celt` parameter of the `Next()` method indicates how many items should be retrieved. The `elt` parameter contains an array of `TVarArgs` with at least `elt` elements. Upon return, `pceltFetched` (if not `nil`) should hold the actual number of items fetched. This method returns `S_OK` when the number of items returned is the same as the number requested; it returns `S_FALSE` otherwise. The logic for this method iterates over the array in `elt` and assigns a `TVarArg` representing a collection item to an element of the array. Note the little trick we are performing to clear out the `OleVariant` after assigning it to the array. This ensures that the array will not be garbage collected. Were we not to do this, the contents of `elt` could potentially become stale if the objects referenced by `V` are freed when the `OleVariant` is finalized.

Similar to `TListItem`, the constructor for `TListItems` takes `ComCtrls.TListItems` as a parameter and manipulates that object in the implementation of its methods.

Finally, we complete the implementation of the ActiveX control by adding the logic to manage the `Items` property. First, we must add a field to the object to hold the collection:

```
type
  TListViewX = class(TActiveXControl, IListViewX)
  private
    ...
    FItems: IListItems;
    ...
  end;
```

Next, we assign `FItems` to a new `TListItems` instance in the `InitializeControl()` method:

```
FItems := LVItems.TListItems.Create(FDelphiControl.Items);
```

Lastly, the `Get_Items()` method can be implemented to simply return `FItems`:

```
function TListViewX.Get_Items: IListItems;
begin
  Result := FItems;
end;
```

The real test to see whether this collection works is to load the control in Visual Basic 6 and try to use the `For Each` construct with the collection. Figure 23.16 shows a simple VB test application running.

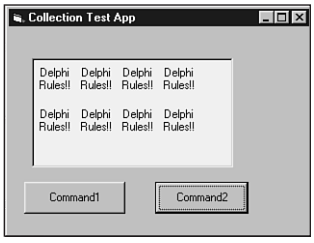

FIGURE 23.16
A Visual Basic application to test our collection.

Of the two command buttons you see in Figure 23.16, `Command1` adds items to the listview, whereas `Command2` iterates over all the items in the listview using `For Each` and adds exclamation points to each caption. The code for these methods is shown here:

```
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
```


Despite the feelings some of the Delphi faithful have toward VB, we must remember that VB is the primary consumer of ActiveX controls, and it's very important to ensure that our controls function properly in that environment.

Collections provide powerful functionality that can enable your controls and Automation servers to function more smoothly in the world of COM. Because collections are terribly difficult to implement, it's worth your while to get in the habit of using them when appropriate. Unfortunately, once you become comfortable with collections, it's very likely that someone will soon come along and create yet a newer and better container object for COM.

New Interface Types in the Type Library

As every well-behaved Delphi developer should, we have used the type library editor to define new interfaces for our Automation objects. However, it is not unusual to occasionally run into a situation where one of the methods for a new interface includes a parameter of a COM interface type that isn't supported by default in the type library editor. Because the type library editor does not let you work with types that it doesn't know about, how do you complete such a method definition?

Before this is explained, it's important that you understand why the type library editor behaves the way it does. If you create a new method in the type library editor and take a look at the types available in the Type column of the Parameters page, you will see a number of interfaces, including `IDataBroker`, `IDispatch`, `IEnumVARIANT`, `IFont`, `IPicture`, `IProvider`, `IStrings`, and `IUnknown`. Why are these the only interfaces available? What makes them so special? They're not special, really—they just happen to be types that are defined in type libraries that are used by this type library. By default, a Delphi type library automatically uses the Borland Standard VCL type library and the OLE Automation type library. You can configure which type libraries are used by your type library by selecting the root node in the tree view in the left pane of the type library editor and choosing the Uses tab in the page control in the right pane. The types contained in the type libraries used by your type library will automatically become available in the drop-down list shown in the type library editor.

Armed with this knowledge, you've probably already figured out that if the interface you want to use as the method parameter in question is defined in a type library, you can simply use that type library, and the problem is solved. But what if the interface isn't defined in a type library? There are certainly quite a few COM interfaces that are defined only by SDK in header or IDL files and are not found in type libraries. If this is the case, the best course is to define the method parameter as being of type `IUnknown`. This `IUnknown` can be `QueryInterfaced` in your method implementation for the specific interface type you want to work with. You should also be sure to document this method parameter as an `IUnknown` that must support the appropriate interface. The following code shows an example of how such a method could be implemented:

```
procedure TSomeClass.SomeMethod(SomeParam: IUnknown);
var
  Intf: ISomeComInterface;
begin
  Intf := SomeParam as ISomeComInterface;
  // remainder of method implementation
end;
```

You should also be aware of the fact that the interface to which you cast the `IUnknown` must be an interface that COM knows how to marshal. This means that it must either be defined in a type library somewhere, must be a type compatible with the standard Automation marshaler, or the COM server in question must provide a proxy/stub DLL capable of marshalling the interface.

Exchanging Binary Data

Occasionally you may want to exchange a block of binary data between an Automation client and server. Because COM doesn't support the exchange of raw pointers, you can't simply pass pointers around. However, the solution isn't much more difficult than that. The easiest way to exchange binary data between Automation clients and servers is to use safearrays of bytes.

Delphi encapsulates safearrays nicely in `OleVariants`. The admittedly contrived example shown in Listings 23.13 and 23.14 depict client and server units that use memo text to demonstrate how to transfer binary data using safearrays of bytes.

LISTING 23.13 The Server Unit

```
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
  // Move data from memo into array
  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;
```

continues

LISTING 23.13 Continued

```
begin
  // Move data from array into memo
  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.
```

LISTING 23.14 The Client Unit

```
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
  // Send memo data to server
  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
  // Get server's memo data
  V := FServer.Data;
  L := VarArrayHighBound(V, 1) - VarArrayLowBound(V, 1) + 1;
```

continues

LISTING 23.14 Continued

```
  SetLength(S, L);
  P := VarArrayLock(V);
  try
    Move(P^, S[1], L);
  finally
    VarArrayUnlock(V);
  end;
  Memo.Text := S;
end;

end.
```

Behind the Scenes: Language Support for COM

One thing often heard when folks talk about COM development in Delphi is what great language support Object Pascal provides for COM. (You won't get any static from us on that point.) With features such as interfaces, variants, and wide strings built right into the language, it's hardly a point to be argued. However, what does it mean to have these things built into the language? How do these features work, and what is the nature of their dependence on the COM APIs? In this section, we will take a low-level look at how all the pieces fit together to form Object Pascal's COM support and dig into some of the implementation details of the language features.

As I mentioned, Object Pascal's COM language features can basically be summed up into three categories:

- `Variant` and `OleVariant`, which encapsulate COM's variant record, safearrays, and late-bound Automation.
- `WideString`, which encapsulates COM's `BSTR`.
- `Interface` and `dispinterface`, which encapsulate COM interfaces and early- and ID-bound Automation.

You crusty old OLE developers from the Delphi 2 days might have noticed that the `automated` reserved word, through which late-bound Automation servers could be created, is conveniently ignored. Because this feature was superceded by the "real" Automation support first introduced in Delphi 3 and remains only for backward compatibility, it is not discussed here.

Variants

Variants are the oldest form of COM support in Delphi, dating back to Delphi 2. As you likely already know, a `Variant` is really just a big record that is used to pass around some bit of data that can be any one of a number of types. If you're interested in what this record looks like, it's defined in the `System` unit as `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;
```


The value of the `VType` field of this record indicates the type of data contained in the `Variant`, and it can be any of the variant type codes found at the top of the `System` unit and listed in the `variant` portion of this record (within the `case` statement). The only different between `Variant` and `OleVariant` is that `Variant` supports all the type codes, whereas `OleVariant` only supports those types compatible in Automation. For example, an attempt to assign a Pascal `string` (`varString`) to a `Variant` is an acceptable practice, but assigning the same string to an `OleVariant` will cause it to be converted to an Automation-compatible `WideString` (`varOleStr`).

When you work with the `Variant` and `OleVariant` types, what the compiler is really manipulating and passing around is instances of this `TVarData` record. In fact, you can safely typecast a `Variant` or `OleVariant` to a `TVarData` if you for some reason need to manipulate the innards of the record (although we don't recommend this practice unless you really know what you're doing).

In the harsh world of COM programming in C and C++ (without a class framework), variants are represented with the `VARIANT` struct defined in `oaidl.h`. When working with variants in this environment, you have to manually initialize and manage them using `Variant`*`XXX`*`()` API functions found in `oleaut32.dll`, such as `VariantInit()`, `VariantCopy()`, `VariantClear()`, and so on. This makes working with variants in straight C and C++ a high-maintenance task.

With support for variants built into Object Pascal, the compiler generates the necessary calls to the API's variant-support routines automatically as you use instances of the `Variant` and `OleVariant` types. This nicety in the language does saddle you with one bit of baggage you should know about, however. If you inspect the import table of a "do-nothing" Delphi EXE using a tool such as Borland's `TDUMP.EXE` or Microsoft's `DUMPBIN.EXE`, you will notice a few suspicious imports from `oleaut32.dll`: `VariantChangeTypeEx()`, `VariantCopyInd()`, and `VariantClear()`. What this means is that even in an application in which you do not explicitly employ `Variant` or `OleVariant` types, your Delphi EXE still has a dependence on these COM API functions in `oleaut32.dll`.

Variant Arrays

Variant arrays in Delphi are designed to encapsulate COM safearrays, which are a type of record used to encapsulate an array of data in Automation. They are called *safe* because they are self-describing; in addition to array data, the record contains information regarding the number of dimensions, the size of an element, and the number of elements in the array. Variant arrays are created and managed in Delphi using the `VarArray`*`XXX`*`()` functions and procedures found in the `System` unit and documented in the online help. These functions and procedures are essentially wrappers around the API's `SafeArray`*`XXX`*`()` functions. Once a `Variant` contains a variant array, standard array subscript syntax is used to access array elements. Once again, comparing this to manually coding safearrays as you would in C and C++, Object Pascal's language encapsulation is clean and much less cumbersome and error prone.

Late-Binding Automation

As you learned earlier in this chapter, `Variant` and `OleVariant` types enable to write late-binding Automation clients (*late-binding* means that functions are called at runtime using the `Invoke` method of the `IDispatch` interface). That's all pretty easy to take at face value, but the question is "Where is the magic connection between calling a method of an Automation server from a `Variant` and `IDispatch.Invoke()` somehow getting called with the right parameters?" The answer is more low-tech than you might expect.

When a method call is made on a `Variant` or `OleVariant` containing an `IDispatch`, the compiler simply generates a call to the `_DispInvoke` helper function declared in the `System` unit, which jumps to a function pointer called `VarDispProc`. By default, the `VarDispProc` pointer is assigned to a method that simply returns an error when it is called. However, if you include the `ComObj` unit in your `uses` clause, the `initialization` section for the `ComObj` unit redirects `VarDispProc` to another method with a line of code that looks like this:

```
VarDispProc := @VarDispInvoke;
```

`VarDispInvoke` is a procedure in the `ComObj` unit with the following declaration:

```
procedure VarDispInvoke(Result: PVariant; const Instance: Variant;
  CallDesc: PCallDesc; Params: Pointer); cdecl;
```

The implementation of the procedure handles the complexity of calling `IDispatch.GetIDsOfNames()` to obtain a DispID from the method name, setting up the parameters correctly, and making the call to `IDispatch.Invoke()`. What's interesting about this is that the compiler in this instance doesn't have any inherent knowledge of `IDispatch` or how the `Invoke()` call is made; it simply passes a bunch of stuff through a function pointer. Also interesting is the fact that because of this architecture, you could reroute this function pointer to your own procedure if you wanted to handle all Automation calls through `Variant` and `OleVariant` types yourself. You would only have to ensure that your function declaration matched that of `VarDispInvoke`. Certainly, this would be a task reserved for experts, but it's interesting to know that the flexibility is there when you need it.

WideString

The `WideString` data type was added in Delphi 3 to serve the dual purpose of providing a native double-byte, Unicode character string and a character string compatible with the COM `BSTR` string. The `WideString` type differs from its cousin `AnsiString` in a few key respects:

- The characters comprising a `WideString` string are all two bytes in size.
- `WideString` types are always allocated using `SysAllocStringLen()` and therefore are fully compatible with `BSTR`s.
- `WideString` types are never reference counted and therefore are always copied on assignment.

Like variants, `BSTR`s can be cumbersome to work with using standard API functions, so the native Object Pascal support via `WideString` is certainly a welcome language addition. However, because they consume twice the memory and are not reference counted, they are much more inefficient than `AnsiString`s, and you should therefore be judicious about their use.

Like the Pascal `Variant`, `WideString` causes a number of functions to be imported from `oleaut32.dll`, even if you don't employ this type yourself. Inspecting the import table of a "do-nothing" Delphi application reveals that `SysStringLen()`, `SysFreeString()`, `SysReAllocStringLen()`, and `SysAllocStringLen()` are all pulled in by the Delphi RTL in order to provide `WideString` support.

Interfaces

Perhaps the most important big-ticket COM feature in the Object Pascal language is the native support for interfaces. Somewhat ironically, although arguably smaller features such as `Variant`s and `WideString`s pull in functions from the COM API for implementation, Object Pascal's implementation of interfaces doesn't require COM at all. That is, Object Pascal provides a completely self-contained implementation of interfaces that adheres to the COM specification, but it doesn't necessarily require any COM API functions.

As a part of adhering to the COM spec, all interfaces in Delphi implicitly descend from `IUnknown`. As you may know, `IUnknown` provides the identity and reference counting support that is the root of COM. This means that knowledge of `IUnknown` is built into the compiler, and `IUnknown` is defined in the `System` unit. By making `IUnknown` a first-class citizen in the language, Delphi is able to provide the automatic reference counting by having the compiler generate the calls to `IUnknown.AddRef()` and `IUnknown.Release()` at the appropriate times. Additionally, the `as` operator can be used as a shortcut for interface identity normally obtained via `QueryInterface()`. The root support for `IUnknown`, however, is almost incidental when you consider the low-level support that the language and compiler provide for interfaces in general.

Figure 23.17 shows a simplified diagram of how classes internally support interfaces. A Delphi object is really a reference that points to the physical instance. The first four bytes of an object instance are a pointer the object's virtual method table (VMT). At a positive offset from the VMT are all the object's virtual methods. At a negative offset are pointers to methods and data that are important to the internal function of the object. In particular, offset `-72` from the VMT contains a pointer to the object's interface table. The interface table is a list of `PInterfaceEntry` records (defined in the `System` unit) that essentially contain the IID and information on where to find the vtable pointer for that IID.

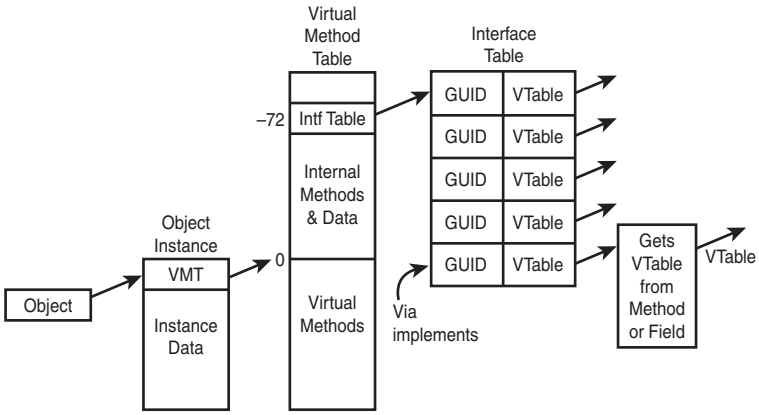

FIGURE 23.17
How interfaces are supported internally in Object Pascal.

After you have a moment to reflect on the diagram in Figure 23.17 and understand how things are put together, the details surrounding the implementation of interfaces just kind of fall into place. For example, `QueryInterface()` is normally implemented on Object Pascal objects by calling `TObject.GetInterface()`. `GetInterface()` walks the interface table looking for the IID in question and returns the vtable pointer for that interface. This also illustrates why new interface types must be defined with a GUID; otherwise, there would be no way for

`GetInterface()` to walk the interface table, and therefore there would be no identity via `QueryInterface()`. Typecasting of interfaces using the `as` operator simply generates a call to `QueryInterface()`, so the same rules apply there.

The last entry in the interface table in Figure 23.17 illustrates how an interface is implemented internally using the `implements` directive. Rather than providing a direct pointer for the vtable, the interface table entry provides the address of a little compiler-generated getter function that gets the interface vtable from the property upon which the implements directive was used.

Dispinterfaces

A dispinterface provides an encapsulation of a non–dual `IDispatch`. That is, an `IDispatch` in which methods can be called via `Invoke()` but not via a vtable. In this respect, a dispinterface is similar to Automation with variants. However, dispinterfaces are slightly more efficient than variants because `dispinterface` declarations contain the DispID for each of the properties or methods supported. This means that `IDispatch.Invoke()` can be called directly without first calling `IDispatch.GetIDsOfNames()`, as must be done with a variant. The mechanism behind dispinterfaces is similar to that of variants: When you call a method via a dispinterface, the compiler generates a call to `_IntfDispCall` in the `System` unit. This method jumps through the `DispCallByIDProc` pointer, which by default only returns an error. However, when the `ComObj` unit is included, `DispCallByIDProc` is routed to the `DispCallByID()` procedure, which is declared in `ComObj` as follows:

```
procedure DispCallByID(Result: Pointer; const Dispatch: IDispatch;
  DispDesc: PDispDesc; Params: Pointer); cdecl;
```


Microsoft Transaction Server (MTS)

The COM development community has been making a lot of noise of late about Microsoft Transaction Server (MTS), and not without good reason. MTS represents a new paradigm for COM developers. COM developers have long enjoyed the advantages of language-independent interfaces, location transparency, and automatic activation and deactivation. However, thanks to MTS, COM developers can now take advantage of powerful runtime services, such as lifetime management, security, resource pooling, and transaction management. Although MTS brings a lot of useful features to the table, it also requires some changes in system design that in some cases contradict ideas COM has pounded into our skulls over the years. In this section, we will discuss MTS technology, and in the following section, we will talk more specifically about MTS and Delphi, Delphi's MTS framework and IDE support, and walk through some sample MTS components and applications.

Before we leap into the technical details, you should know up front that transaction handling is only a small part of the MTS big picture, and the fact that *transaction* appears in the name of this technology is quite unfortunate. It's sort of like calling your new home entertainment

system a soap opera viewer. Yeah, it does that, but it's so much more. To their credit, when we've spoken with folks at Microsoft close to the technology, they generally hate the name. Fortunately, the name *MTS* won't be with us much longer; as mentioned earlier in this chapter, MTS will be folded into the operating system as a part of the upcoming enhancements to COM known as COM+.

Why MTS?

The magic word of system design these days is *scalability*. With the hypergrowth of the Internet and intranets, the consolidation of corporate data into centrally located data stores, and the need for everyone and their cousin to get at the data, it's absolutely crucial that a system be able to scale to ever-larger numbers of concurrent users. It's definitely a challenge, especially considering the rather unforgiving limitations we must deal with, such as finite database connections, network bandwidth, server load, and so on. In the good old days of the early 90s, client/server computing was all the rage and considered The Way to write scalable applications. However, as databases were bogged down with triggers and stored procedures and clients were complicated with various bits of code here and there in an effort to implement business rules, it shortly became obvious that such systems would never scale to a large number of users. The multitier architecture soon became popular as a way to scale a system to a greater number of users. By placing application logic and sharing database connections in the middle tier, database and client logic could be simplified and resource usage optimized for an overall higher-bandwidth system.

As a side note, it is interesting that the added infrastructure introduced in a multitier environment tends to increase latency as it increases bandwidth. In other words, you may very well need to sacrifice the performance of the system in order to improve scalability.

Microsoft extended to COM developers the ability to build applications that are distributed across multiple machines with the introduction of DCOM several years ago. DCOM was a step in the right direction. It provided the means by which things COM may communicate with one another over the wire, but it did not make many significant steps toward solving the real-world problems encountered by developers of distributed applications. Issues such as lifetime optimization, thread management, flexible security, and transaction support were still left to individual developers. Enter MTS.

What Is MTS?

MTS is a COM-based programming model and collection of runtime services for developing scalable or transactional COM-based applications. The programming model part of MTS isn't much different than what you are familiar with already as a COM developer. There are a few wrinkles that you will learn about shortly, but for the most part, any in-process (DLL) COM object with a type library can be an MTS object. However, it's not recommended that you run

non-MTS-aware COM components within MTS. MTS runtime services mean that MTS serves as the caregiver for your COM components. MTS can host them, manage their lifetime, provide security for them, and so on. This means that rather than running within the context of your application, MTS COM objects run within the context of the MTS runtime. All this adds up to a bunch of new features that you can take advantage of with little or no coding changes in your client or COM object code.

It's interesting to note that because MTS objects do not run directly within the context of a client like other COM objects, clients never really obtain interface pointers directly to an object instance. Instead, MTS inserts a proxy between the client and the MTS object such that the proxy is identical to the object from the client's point of view. However, because MTS has complete control over the proxy, it can control access to interface methods of the object for purposes such as lifetime management and security, as you will soon learn.

Stateful Versus Stateless

The number one topic of conversation among folks looking at, playing with, and working on MTS technology seems to be the discussion of stateful versus stateless objects. Although COM itself doesn't give a whit as to the state of an object, in practice most traditional COM objects are stateful. That is, they continuously maintain state information from the time they're created, while they're being used, and up until the time they're destroyed. The problem with stateful objects is that they aren't particularly scalable, because state information would have to be maintained for every object being accessed by every client. A *stateless object* is one that generally does not maintain state information between method calls. Stateless objects are preferred because they enable MTS to play some optimization tricks. If an object doesn't maintain any state between method calls, MTS can theoretically make the object go away between calls without causing any harm. Furthermore, because the client maintains pointers only to MTS's internal proxy for the object, MTS could do so without the client being any the wiser. It's more than a theory; this is actually how MTS works. MTS will destroy the instances of the object between calls in order to free up resources associated with the object. When the client makes another call to that object, the MTS proxy will intercept it and a new instance of the object will be created automatically. This helps the system scale to a larger number of users, because there will likely be comparatively few active instances of a class at any given time.

Writing interfaces to behave in a stateless manner will probably require a slight departure from your usual way of thinking for interface design. For example, consider the following classic COM-style interface:

```
ICheckbook = interface
['{2CCF0409-EE29-11D2-AF31-0000861EF0BB}']
  procedure SetAccount(AccountNum: WideString); safecall;
  procedure AddActivity(Amount: Integer); safecall;
end;
```

As you might imagine, you would use this interface in a manner something like this:

```
var
  CB: ICheckbook;
begin
  CB := SomehowGetInstance;
  CB.SetAccount('12345ABCDE');  // open my checking account
  CB.AddActivity(-100);                  // add a debit for $100
  ...
end;
```

The problem with this style is that the object is not stateless between method calls because state information regarding the account number must be maintained across the call. A better approach to this interface for use in MTS would be to pass all the necessary information to the `AddActivity()` method so that the object could behave in a stateless manner, as shown here:

```
procedure AddActivity(AccountNum: WideString; Amount: Integer); safecall;
```

The particular state of an active object is also referred to as a *context*. MTS maintains a context for each active object that tracks things such as security and transaction information for the object. An object can at any time call `GetObjectContext()` to obtain an `IObjectContext` interface pointer for the object's context. `IObjectContext` is defined in the `Mtx` unit as follows:

```
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;
```

The two most important methods in this interface are `SetComplete()` and `SetAbort()`. If either of these methods are called, the object is telling MTS that it no longer has any state to maintain. MTS will therefore destroy the object (unbeknownst to the client, of course), thereby freeing up resources for other instances. If the object is participating in a transaction, `SetComplete()` and `SetAbort()` also have the effect of a commit and rollback for the transaction, respectively.

Lifetime Management

From the time we were itty-bitty COM programmers, we were taught to hold onto interface pointers only for as long as necessary and to release them as soon as they were unneeded. In traditional COM, this makes a lot of sense because we don't want to occupy the system with

maintaining resources that aren't being used. However, because MTS will automatically free up stateless objects after they call `SetComplete()` or `SetAbort()`, there is no expense associated with holding a reference to such an object indefinitely. Furthermore, because the client never knows that the object instance may have been deleted under the sheets, clients do not have to be rewritten to take advantage of this feature.

Packages

The word *package* is already overloaded enough—Delphi packages, C++Builder packages, and Oracle packages are all examples of the overuse of this word. MTS also has a notion of *packages* that no doubt differs from those other varieties. An MTS package is more logical than physical, because it represents a programmer-defined collection of MTS objects with like activation, security, and transaction attributes. The physical part of a package is a file that contains references to the COM server DLLs and MTS objects within those servers that make up the package. The package file also contains information on the attributes of the MTS objects within.

MTS will run all components within a package in the same process. This enables you to configure your well-behaved and error-free packages so that they are insulated from the potential problems that could be caused by faults or errors in other packages. It is also interesting to note that the physical location of components has no bearing on eligibility for package inclusion: A single COM server can contain several COM objects, each in a separate package.

Packages are created and manipulated using either the Run, Install MTS Objects menu in Delphi or the Transaction Server Explorer that is installed with MTS and shown in Figure 23.18.

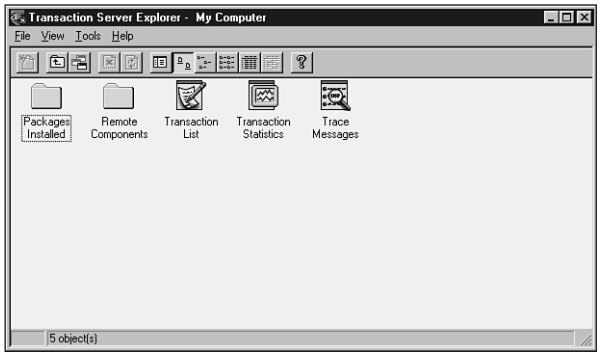

FIGURE 23.18
The Windows 98 Transaction Server Explorer.

Security

MTS provides a roll-based security system that is much more flexible than the standard Windows NT security normally used with DCOM. A *roll* is a category of user (for example, in

a banking system typical rolls might be teller, supervisor, and manager). MTS allows you to specify the degree to which any particular roll can manipulate an object on a per-interface basis. For example, you can specify that the manager roll has access to the `ICreateHomeLoan` interface, but the teller roll does not. If you need to get more granular than access to entire interfaces, you can determine the roll of the user in the current context by calling the `IsCallerInRole()` method of `IObjectContext`. Using this, for example, you could enforce a business rule that stipulates that tellers can approve normal account closures, but only supervisors can approve an account closure when the account balance is over $100,000. Security rolls can be configured in the Transaction Server Explorer.

Oh, It Also Does Transactions

Of course, as the name implies, MTS also does transactions. You might be thinking to yourself, "Big deal, my database server already supports transactions. Why do I need my components to support them as well?" This is a fair question, and luckily there's a good answer. Transaction support in MTS can enable you to perform transactions across multiple databases or can even make a single atomic action out of some set of operations having nothing to do with databases. In order to support transactions on your MTS objects, you must set the correct transaction flag on your object's coclass in the type library either during development (this is what the Delphi MTS Wizard does) or after deployment in the Transaction Server Explorer.

When should you use transactions in your objects? That's easy: You should use transactions whenever you have a process involving multiple steps that you want to make into a single, atomic transaction. In doing so, the entire process can be either committed or rolled back, but you will never leave your logic or data in an incorrect or indeterminate state somewhere in between. For example, if you are writing software for a bank and you want to handle the case where a client bounces a check, there would likely be several steps involved in handling that, including debiting account for the amount of the check, debiting the account for the bounced check service charge, and sending a letter to the client.

In order to properly process the bounced check, each of these things must happen. Therefore, wrapping them in a single transaction would ensure that all will occur (if no errors are encountered) or all will roll back to their original pretransaction state if an error occurs.

Resources

With objects being created and destroyed all the time and transactions happening everywhere, it's important for MTS to provide a means for sharing certain finite or expensive resources (such as database connections) across multiple objects. MTS does this using resource managers and resource dispensers. A *resource manager* is a service that manages some type of durable data, such as account balances or inventory. Microsoft provides a resource manager in MS SQL Server. A *resource dispenser* manages nondurable resources, such as database connections. Microsoft provides a resource dispenser for ODBC database connections, and Borland provides a resource dispenser for BDE database connections.

When a transaction makes use of some type of resource, it *enlists* the resource to become a part of the transaction so that all changes made to the resource during the transaction will participate in the commit or rollback operation of the transaction.

MTS in Delphi

Now that you've got the "what" and "why" down, it's time to talk about the "how." In particular, we will focus on Delphi's support of MTS and how to build MTS solutions in Delphi. Before we jump right in, however, you should first know that MTS support is built only into the Enterprise version of Delphi. Although it's technically possible to create MTS components using the facilities available in the Standard and Professional versions, it is not the most productive use of your time. Therefore, this section will help you leverage the features of Delphi Enterprise.

MTS Wizards

Delphi provides two wizards for building MTS components that are both found on the Multitier tab of the New Items dialog: the MTS Remote Data Module Wizard and the MTS Object Wizard. The MTS Remote Data Module Wizard enables you to build MIDAS servers that operate in the MTS environment. The MTS Object Wizard will serve as the starting point for your MTS objects, and it will be the focus of this discussion. Upon invoking this wizard, you will be presented with the dialog shown in Figure 23.19.

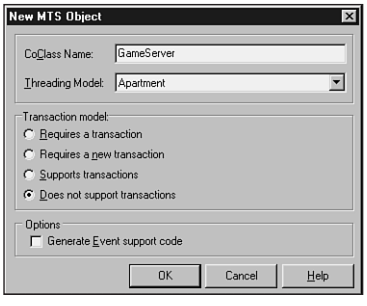

FIGURE 23.19
The New MTS Object Wizard.

The dialog in Figure 23.19 is similar to the Automation Object Wizard discussed earlier in this chapter. The obvious difference is the facility provided by this wizard to select the transaction model supported by your MTS component. The available transaction models are as follows:

- *Requires a transaction.* The component will always be created within the context of a transaction. It will inherit the transaction of its creator if one exists; otherwise, it will create a new one.

- *Requires a new transaction.* A new transaction will always be created for the component to execute within.
- *Supports transactions.* The component will inherit the transaction of its creator if one exists; otherwise, it will execute without a transaction.
- *Does not support transactions.* The component will never be created within a transaction.

The transaction model information is stored as an attribute along with the component's coclass in the type library.

After you click OK to dismiss the dialog, the wizard will generate an empty definition for a class that descends from `TMtsAutoObject`, and it will leave you off in the Type Library Editor in order to define your MTS components by adding properties, methods, interfaces, and so on. This should be familiar territory because the workflow is identical at this point to developing Automation objects in Delphi. It's interesting to note that although the Delphi wizard–created MTS objects are Automation objects (that is, COM objects that implement `IDispatch`), MTS doesn't technically require this. However, because COM inherently knows how to marshal `IDispatch` interfaces accompanied by type libraries, employing this type of object in MTS enables you to concentrate more on your components' functionality and less on how they integrate with MTS. You should also be aware that MTS components must reside in in-process COM servers (DLLs); MTS components are not supported in out-of-process servers (EXEs).

MTS Framework

The aforementioned `TMtsAutoObject` class, which is the base class for all Delphi wizard–created MTS objects, is defined in the `MtsObj` unit. `TMtsAutoObject` is a relatively straightforward class that is defined as follows:

```
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;
```

`TMtsAutoObject` is essentially a `TAutoObject` that adds two important bits of functionality:

- `TMtsAutoObject` implements the `IObjectControl` interface, which manages initialization and cleanup of MTS components. Here are the methods of this interface:

Method Name	*Description*
`Activate`	Allows an object to perform context-specific initialization when activated. This method will be called by MTS prior to any custom methods on your MTS component.
`Deactivate`	Enables you to perform context-specific cleanup when an object is deactivated.
`CanBePooled`	This method is currently unused because MTS does not yet support object pooling.

  `TMtsAutoObject` provides virtual `OnActivate()` and `OnDeactivate()` methods, which are fired from the private `Activate()` and `Deactivate()` methods. Simply override these to create special context-specific activation or deactivation logic.

- `TMtsAutoObject` also maintains a pointer to MTS's `IObjectContext` interface in the form of the `ObjectContext` property. As previously explained, `IObjectContext` is the interface provided by MTS that provides a component the ability to manipulate its current context. As a shortcut for users of this class, `TMtsAutoObject` also surfaces each of `IObjectContext`'s methods, which are implemented to simply call into `ObjectContext`. For example, the implementation of the `TMtsAutoObject.SetComplete()` method simply checks `FObjectContext` for `nil` and then calls `FObjectContext.SetComplete()`. Here's a list of `IObjectContext`'s methods and a brief explanation of each:

Method Name	*Description*
`CreateInstance`	Creates an instance of another MTS object. You can think of this method as performing the same task for MTS objects as `IClassFactory.CreateInstance` does for normal COM objects.
`SetComplete`	Signals to MTS that the component has completed whatever work it needs to do and no longer has any internal state to maintain. If the component is transactional, it also indicates that the current transactions can be committed. After the method calling this function returns, MTS may deactivate the object, thereby freeing up resources for greater scalability.

Method Name	*Description*
`SetAbort`	Similar to `SetComplete()`, this method signals to MTS that the component has completed work and no longer has state information to maintain. However, calling this method also means that the component is in an error or indeterminate state and any pending transactions must be aborted.
`EnableCommit`	Indicates that the component is in a "committable" state, such that transactions can be committed when the component calls `SetComplete`. This is the default state of a component.
`DisableCommit`	Indicates that the component is in an inconsistent state, and further method invocations are necessary before the component will be prepared to commit transactions.
`IsInTransaction`	Enables the component to determine whether it is executing within the context of a transaction.
`IsSecurityEnabled`	Allows a component to determine whether MTS security is enabled. This method always returns `True` unless the component is executing in the client's process space.
`IsCallerInRole`	Provides a means by which a component can determine whether the user serving as the client for the component is a member of a specific MTS role. This method is the heart of MTS's easy-to-use, role-based security system. (More on roles later in this chapter.)

The `Mtx` unit contains the core MTS support. It is the Pascal translation of the `mtx.h` header file, and it contains the types (such as `IObjectControl` and `IObjectContext`) and functions that make up the MTS API.

Tic-Tac-Toe: A Sample Application

Enough theory. Now it's time to write some code and see how all this MTS stuff performs on the open road. MTS ships with a sample tic-tac-toe application that's a bit on the ugly side, so it inspired us to implement the classic game from the ground up in Delphi. To start, we use the MTS Object Wizard to create a new object called `GameServer`. Using the Type Library Editor, we add to the default interface for this object, `IGameServer`, three methods: `NewGame()`, `ComputerMove()`, and `PlayerMove()`. Additionally, we add two new enums, `SkillLevels` and `GameResults`, which are used by these methods. Figure 23.20 shows all of these items displayed in the type library editor.

The logic behind the three methods of this interface is simple, and these methods make up the requirements to support a game of human versus computer tic-tac-toe. `NewGame()` initializes a new game for the client. `ComputerMove()` analyzes the available moves and makes a move for the computer. `PlayerMove()` enables the client to let the computer know how he or she has chosen to move. Earlier in this chapter we mentioned that MTS component development

requires a different frame of mind than development of standard COM components. This component offers a nice opportunity to illustrate this fact.

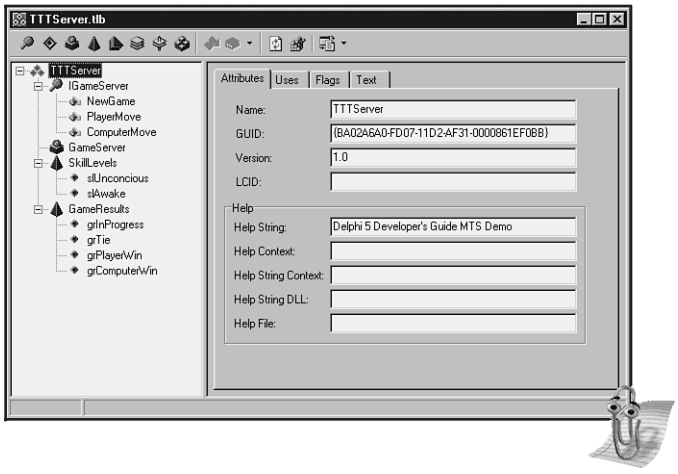

FIGURE 23.20
The tic-tac-toe server, as shown in the type library editor.

If this were your average, everyday, run-of-the-mill COM component, you might approach design of the object by initializing some data structure to maintain game state in the `NewGame()` method. That data structure would probably be an instance field of the object, which the other methods would access and manipulate throughout the life of the object.

What's the problem with this approach for an MTS component? One word: state. As you learned earlier, objects must be stateless in order to realize the full benefit of MTS. However, a component architecture that depends on instance data to be maintained across method calls is far from stateless. A better design for MTS would be to return a "handle" identifying a game from the `NewGame()` method and using that handle to maintain per-game data structures in some type of shared resource facility. This shared resource facility would need to be maintained outside the context of a specific object instance, because MTS may activate and deactivate object instances with each method call. Each of the other methods of the component could accept this handle as a parameter, enabling it to retrieve game data from the shared resource facility. This is a stateless design because it doesn't require the object to remain activated between method calls, because each method is a self-contained operation that gets all the data it needs from parameters and a shared data facility.

This shared data facility that we are speaking abstractly about is known as a *resource dispenser* in MTS. Specifically, the Shared Property Manager is the MTS resource dispenser that is used to maintain component-defined, process-wide shared data. The Shared Property Manager is

represented by the `ISharedPropertyGroupManager` interface. The Shared Property Manager is the top level of a hierarchical storage system, maintaining any number of shared property groups, which are represented by the `ISharedPropertyGroup` interface. In turn, each shared property group may contain any number of shared properties, represented by the `ISharedProperty` interface. Shared properties are convenient because they exists within MTS, outside the context of any specific object instance, and access to them is controlled by locks and semaphores managed by the Shared Property Manager.

With all that in mind, the implementation of the `NewGame()` method is shown in the following code:

```
procedure TGameServer.NewGame(out GameID: Integer);
var
  SPG: ISharedPropertyGroup;
  SProp: ISharedProperty;
  Exists: WordBool;
  GameData: OleVariant;
begin
  // Use caller's role to validate security
  CheckCallerSecurity;
  // Get shared property group for this object
  SPG := GetSharedPropertyGroup;
  // Create or retrieve NextGameID shared property
  SProp := SPG.CreateProperty('NextGameID', Exists);
  if Exists then GameID := SProp.Value
  else GameID := 0;
  // Increment and store NextGameID shared property
  SProp.Value := GameID + 1;
  // Create game data array
  GameData := VarArrayCreate([1, 3, 1, 3], varByte);
  SProp := SPG.CreateProperty(Format(GameDataStr, [GameID]), Exists);
  SProp.Value := GameData;
  SetComplete;
end;
```

This method first checks to ensure the caller is in the proper role to invoke this method (more on this in a moment). It then uses a shared property to obtain an ID number for the next game. Next, this method creates a variant array into which to store game data and saves that data as a shared property. Finally, this method calls `SetComplete()` so that MTS knows its okay to deactivate this instance after the method returns.

This leads to the number one rule of MTS development: Call `SetComplete()` or `SetAbort()` as often as possible. Ideally, you will call `SetComplete()` or `SetAbort()` in every method so that MTS can reclaim resources previously consumed by your component instance after the method returns. A corollary to this rule is that object activation and deactivation should not be expensive, because that code is likely to be called quite frequently.

The implementation of the `CheckCallerSecurity()` method illustrates how easy it is to take advantage of role-based security in MTS:

```
procedure TGameServer.CheckCallerSecurity;
begin
  // Just for fun, only allow those in the "TTT" role to play the game.
  if IsSecurityEnabled and not IsCallerInRole('TTT') then
    raise Exception.Create('Only those in the TTT role can play tic-tac-toe');
end;
```

This code begs the obvious question, "How does one establish the TTT role and determine what users belong to that role?" Although it's possible to define roles programmatically, the most straightforward way to add and configure roles is using the Windows NT Transaction Server Explorer. After the component is installed (you'll learn how to install the component shortly), you can set up roles using the "Roles" node found under each package node in the Explorer. It's important to note that roles-based security is supported only for components running on Windows NT. For components running on Windows 9*x*, `IsCallerInRole()` will always return `True`.

The `ComputerMove()` and `PlayerMove()` methods are shown in the following code:


```
procedure TGameServer.ComputerMove(GameID: Integer;
  SkillLevel: SkillLevels; out X, Y: Integer; out GameRez: GameResults);
var
  Exists: WordBool;
  PropVal: OleVariant;
  GameData: PGameData;
  SProp: ISharedProperty;
begin
  // Get game data shared property
  SProp := GetSharedPropertyGroup.CreateProperty(Format(GameDataStr, [GameID]),
    Exists);
  // Get game data array and lock it for more efficient access
  PropVal := SProp.Value;
  GameData := PGameData(VarArrayLock(PropVal));
  try
    // If game isn't over, then let computer make a move
    GameRez := CalcGameStatus(GameData);
    if GameRez = grInProgress then
    begin
      CalcComputerMove(GameData, SkillLevel, X, Y);
      // Save away new game data array
      SProp.Value := PropVal;
      // Check for end of game
      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
  // Get game data shared property
  SProp := GetSharedPropertyGroup.CreateProperty(Format(GameDataStr, [GameID]),
    Exists);
  // Get game data array and lock it for more efficient access
  PropVal := SProp.Value;
  GameData := PGameData(VarArrayLock(PropVal));
  try
    // Make sure game isn't over
    GameRez := CalcGameStatus(GameData);
    if GameRez = grInProgress then
    begin
      // If spot isn't empty, raise exception
      if GameData[X, Y] <> EmptySpot then
        raise Exception.Create('Spot is occupied!');
      // Allow move
      GameData[X, Y] := PlayerSpot;
      // Save away new game data array
      SProp.Value := PropVal;
      // Check for end of game
      GameRez := CalcGameStatus(GameData);
    end;
  finally
    VarArrayUnlock(PropVal);
  end;
  SetComplete;
end;
```

These methods are similar in that they both obtain the game data from the shared property based on the `GameID` parameter, manipulate the data to reflect the current move, save the data away again, and check to see whether the game is over. The `ComputerMove()` method also calls `CalcComputerMove()` to analyze the game and make a move. If you're interested in seeing this and the other logic of this MTS component, take a look at Listing 23.15, which contains the entire source code for the `ServMain` unit.

LISTING 23.15 ServMain.pas: Containing 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('Failed to obtain object context');
```

continues

LISTING 23.15 Continued

```
  // Create shared property group for this object
  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('Failed to obtain property group');
end;

procedure TGameServer.NewGame(out GameID: Integer);
var
  SPG: ISharedPropertyGroup;
  SProp: ISharedProperty;
  Exists: WordBool;
  GameData: OleVariant;
begin
  // Use caller's role to validate security
  CheckCallerSecurity;
  // Get shared property group for this object
  SPG := GetSharedPropertyGroup;
  // Create or retrieve NextGameID shared property
  SProp := SPG.CreateProperty('NextGameID', Exists);
  if Exists then GameID := SProp.Value
  else GameID := 0;
  // Increment and store NextGameID shared property
  SProp.Value := GameID + 1;
  // Create game data array
  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
  // Get game data shared property
  SProp := GetSharedPropertyGroup.CreateProperty(Format(GameDataStr, [GameID]),
    Exists);
```

```
  // Get game data array and lock it for more efficient access
  PropVal := SProp.Value;
  GameData := PGameData(VarArrayLock(PropVal));
  try
    // If game isn't over, then let computer make a move
    GameRez := CalcGameStatus(GameData);
    if GameRez = grInProgress then
    begin
      CalcComputerMove(GameData, SkillLevel, X, Y);
      // Save away new game data array
      SProp.Value := PropVal;
      // Check for end of game
      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
  // Get game data shared property
  SProp := GetSharedPropertyGroup.CreateProperty(Format(GameDataStr, [GameID]),
    Exists);
  // Get game data array and lock it for more efficient access
  PropVal := SProp.Value;
  GameData := PGameData(VarArrayLock(PropVal));
  try
    // Make sure game isn't over
    GameRez := CalcGameStatus(GameData);
    if GameRez = grInProgress then
    begin
      // If spot isn't empty, raise exception
      if GameData[X, Y] <> EmptySpot then
        raise Exception.Create('Spot is occupied!');
      // Allow move
      GameData[X, Y] := PlayerSpot;
      // Save away new game data array
      SProp.Value := PropVal;
```

continues

LISTING 23.15 Continued

```
      // Check for end of game
      GameRez := CalcGameStatus(GameData);
    end;
  finally
    VarArrayUnlock(PropVal);
  end;
  SetComplete;
end;

function TGameServer.CalcGameStatus(GameData: PGameData): GameResults;
var
  I, J: Integer;
begin
  // First check for a winner
  if GameData[1, 1] <> EmptySpot then
  begin
    // Check top row, left column, and top left to bottom right diagonal for
    win
    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; // Game result is spot ID + 1
      Exit;
    end;
  end;
  if GameData[3, 3] <> EmptySpot then
  begin
    // Check bottom row and right column for win
    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; // Game result is spot ID + 1
      Exit;
    end;
  end;
  if GameData[2, 2] <> EmptySpot then
  begin
    // Check middle row, middle column, and bottom left to top right diagonal
    for win
    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; // Game result is spot ID + 1
      Exit;
    end;
  end;
  // Finally, check for game still in progress
  for I := 1 to 3 do
    for J := 1 to 3 do
      if GameData[I, J] = 0 then
      begin
        Result := grInProgress;
        Exit;
      end;
  // If we get here, then we've tied
  Result := grTie;
end;

procedure TGameServer.CalcComputerMove(GameData: PGameData;
  Skill: SkillLevels; var X, Y: Integer);
type
  // Used to scan for possible moves by either row, column, or diagonal line
  TCalcType = (ctRow, ctColumn, ctDiagonal);
  // mtWin = one move away from win, mtBlock = opponent is one move away from
  // win, mtOne = I occupy one other spot in this line, mtNew = I occupy no
  // spots on this line
  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;
```

continues

LISTING 23.15 Continued

```
    end
    else if CalcType = ctColumn then
    begin
      I := 1;
      J := Position;
    end
    else begin
      I := 1;
      case Position of
        1: J := 1; // scanning from top left to bottom right
        2: J := 3; // scanning from top right to bottom left
      else
        Exit;   // bail; only 2 diagonal scans
      end;
    end;
    // Mask masks off Player or Computer bit, depending on whether we're
    // thinking
// offensively or defensively.  Checktotal determines whether that is a row
    // we need to move into.
    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;
    // loop through all lines in current CalcType
    repeat
      // Get status of current spot (X, O, or empty)
      PosVal := GameData[I, J];
```

```
      // Save away last empty spot in case we decide to move here
      if PosVal = 0 then
      begin
        X := I;
        Y := J;
      end
      else
        // If spot isn't empty, then add masked value to 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);
    // If RowData adds up, then we must block or win, depending on
    // whether we're thinking offensively or defensively.
    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
    // First look to win the game, next look to block a win
    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;
    // Next look to take the center of the board
    if GameData[2, 2] = 0 then
    begin
      GameData[2, 2] := ComputerSpot;
```

continues

LISTING 23.15 Continued

```
      X := 2;
      Y := 2;
      Exit;
    end;
    // Next look for the most advantageous position on a line
    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;
  // Finally (or if skill level is unconscious), just find the first open place
  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
  // Just for fun, only allow those in the "TTT" role to play the game.
  if IsSecurityEnabled and not IsCallerInRole('TTT') then
    raise Exception.Create('Only those in the TTT role can play tic-tac-toe');
end;

initialization
  TAutoObjectFactory.Create(ComServer, TGameServer, Class_GameServer,
    ciMultiInstance, tmApartment);
end.
```

Installing the Server

Once the server has been written and you're ready to install it into MTS, Delphi makes your life very easy. Simple select Run, Install MTS Objects from the main menu, and you will invoke the Install MTS Objects dialog. This dialog enables you to install your object(s) into a new or existing package, and it is shown in Figure 23.21.

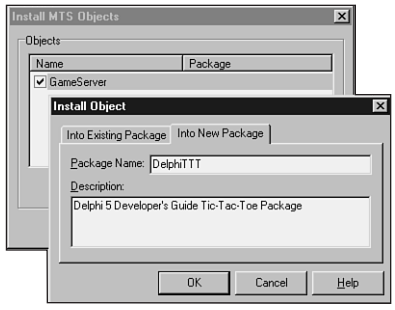

FIGURE 23.21
Installing an MTS object via the Delphi IDE.

Select the component(s) to be installed, specify whether the package is new or existing, click OK, and that's it; the component is installed. Alternatively, you can also install MTS components via the Transaction Server Explorer application. Note that this installation procedure is markedly different than that of standard COM objects, which typically involves using the RegSvr32 tool from the command line to register a COM server. Transaction Server Explorer also makes it similarly easy to set up MTS components on remote machines, providing a welcome alternative to the configuration hell experienced by many of those trying to configure DCOM connectivity.

The Client Application

Listing 23.16 shows the source code for the client application for this MTS component. Its purpose is to essentially map the engine provided by the MTS component to a tic-tac-toe–looking user interface.

LISTING 23.16 `UiMain.pas`: The Main Unit for the Client Application

```
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;
```

continues

LISTING 23.16 Continued

```
  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
  // load "X" and "O" images from resource into TBitmaps
  FXImage := TBitmap.Create;
  FXImage.LoadFromResourceName(MainInstance, 'x_img');
  FOImage := TBitmap.Create;
  FOImage.LoadFromResourceName(MainInstance, 'o_img');
  // set default skill
  FCurrentSkill := slAwake;
  // init record UI
  with FRec do
   StatusBar.SimpleText := Format(RecStr, [Wins, Loses, Ties]);
  // Get server instance
  FGameServer := CoGameServer.Create;
  // Start a new game
  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
  // Show About box
```

continues

LISTING 23.16 Continued

```
  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 = (
```

continues

LISTING 23.16 Continued

```
    'Tie game', 'You win', 'Computer wins');
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! Play again?', [EndMsg[GameRez]]), mtConfirmation,
      [mbYes, mbNo], 0) = mrYes then
      NewGameItemClick(nil);
  end;
end;

end.
```

Figure 23.22 shows this application in action. The user is X, and the computer is O.

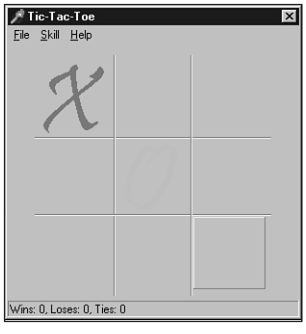

FIGURE 23.22
Playing tic-tac-toe.

Debugging MTS Applications

Because MTS components run within MTS's process space rather than the client's, you might think that they would be difficult to debug. However, MTS provides a side door for debugging purposes that makes debugging a snap. Just load the server project and use the Run Parameters dialog to specify `mtx.exe` as the host application. As a parameter to `mtx.exe`, you must pass

`/p:{package guid}`, where "`package guid`" is the GUID of the package as shown in the Transaction Server Explorer. This dialog is shown in Figure 23.23. Next, set your desired breakpoints and run the application. You won't see anything happen initially because the client application is not yet running. Now you can run the client from Windows Explorer or a command prompt, and you will be off and debugging.

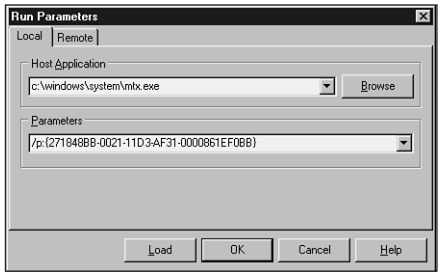

FIGURE 23.23
Using the Run Parameters dialog to set up an MTS debug session.

MTS is a powerful addition to the COM family of technologies. By adding services such as lifetime management, transaction support, security, and transactions to COM objects without requiring significant changes to existing source code, Microsoft has leveraged COM into a more scalable technology, suitable for large-scale distributed development. This section took you through a tour of the basics of MTS and on to the specifics of Delphi's support for MTS and how to create MTS applications in Delphi. What's more, you've hopefully caught a few tips and tricks along the way for developing optimized and well-behaved MTS components. MTS packs a wallop out of the box by providing services such as lifetime management, transaction support, and security, all in a familiar framework. MTS and Delphi combine to provide you with a great way to leverage your COM experience into creating scalable multitier applications. Just don't forget those differences in design nuances between normal COM components and MTS components!

TOleContainer

Now that you have some ActiveX OLE background under your belt, take a look at Delphi's `TOleContainer` class. `TOleContainer` is located in the `OleCntrs` unit, and it encapsulates the complexities of an OLE Document and ActiveX Document container into an easily digestible VCL component.

NOTE

If you were familiar with using Delphi 1.0's TOleContainer component, you can pretty much throw that knowledge out the window. The 32-bit version of this component was redesigned from the ground up (as they say in the car commercials), so any knowledge you have of the 16-bit version of this component may not be applicable to the 32-bit version. Don't let that scare you, though; the 32-bit version of this component is of a much cleaner design, and you'll find that the code you must write to support the object is perhaps a quarter of what it used to be.

A Small Sample Application

Now let's jump right in and create an OLE container application. Create a new project and drop a `TOleContainer` object (found on the System page of the Component Palette) on the form. Right-click the object in the Form Designer and select Insert Object from the local menu. This invokes the Insert Object dialog, as shown in Figure 23.24.

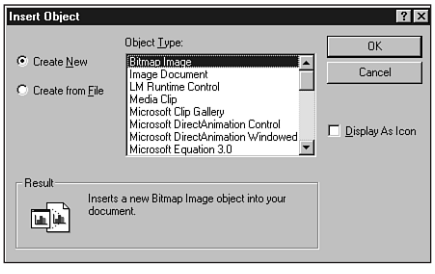

FIGURE 23.24
The Insert Object dialog.

Embedding a New OLE Object

By default, the Insert Object dialog contains the names of OLE server applications registered with Windows. To embed a new OLE object, you can select a server application from the Object Type list box. This causes the OLE server to execute in order to create a new OLE object to be inserted into `TOleContainer`. When you close the server application, the `TOleContainer` object is updated with the embedded object. For this example, we will create a new MS Word 2000 document, as shown in Figure 23.25.

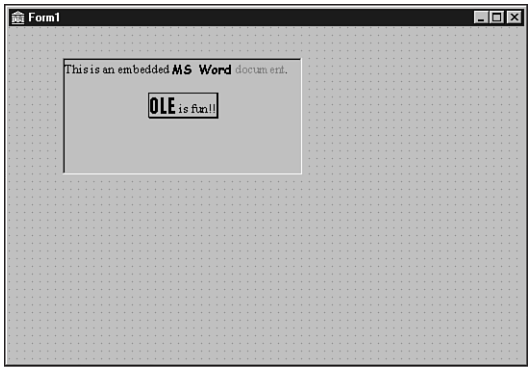

FIGURE 23.25
An embedded MS Word 2000 document.

NOTE

An OLE object will not activate in place at design time. You will only be able to take advantage of the in-place activation capability of `TOleContainer` at runtime.

If you want to invoke the Insert Object dialog at runtime, you can call the `InsertObjectDialog()` method of `TOleContainer`, which is defined as follows:

```
function InsertObjectDialog: Boolean;
```

This function returns `True` if a new type of OLE object was successfully chosen from the dialog.

Embedding or Linking an Existing OLE File

To embed an existing OLE file into the `TOleContainer`, select the Create From File radio button on the Insert Object dialog. This enables you to pick an existing file, as shown in Figure 23.26. After you choose the file, it behaves much the same as a new OLE object.

To embed a file at runtime, call the `CreateObjectFromFile()` method of `TOleContainer`, which is defined as follows:

```
procedure CreateObjectFromFile(const FileName: string; Iconic: Boolean);
```

To link (rather than embed) the OLE object, simply check the Link check box in the Insert Object dialog shown in Figure 23.26. As described earlier, this creates a link from your application to the OLE file so that you can edit and view the same linked object from multiple applications.

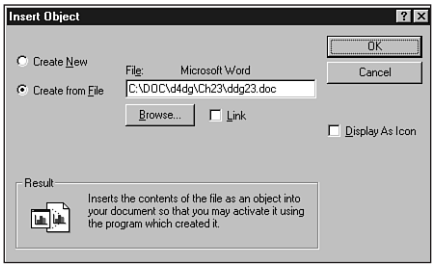

FIGURE 23.26
Inserting an object from a file.

To link to a file at runtime, call the `CreateLinkToFile()` method of `TOleContainer`, which is defined as follows:

```
procedure CreateLinkToFile(const FileName: string; Iconic: Boolean);
```

A Bigger Sample Application

Now that you have the basics of OLE and the `TOleContainer` class behind you, we will create a more sizable application that truly reflects the usage of OLE in realistic applications.

Start by creating a new project based on the MDI application template. The main form makes only a few modifications to the standard MDI template, and it is shown in Figure 23.27.

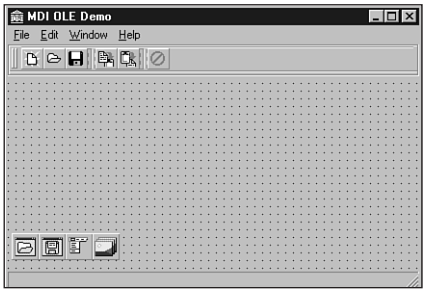

FIGURE 23.27
The MDI OLE Demo main window.

The MDI child form is shown in Figure 23.28. It is simply an `fsMDIChild`-style form with a `TOleContainer` component aligned to `alClient`.

Listing 23.17 shows `ChildWin.pas`, the source code unit for the MDI child form. Note that this unit is fairly standard except for the addition of the `OLEFileName` property and the associated method and `private` instance variable. This property stores the path and filename of the OLE file, and the property accessor sets the child form's caption to the filename.

FIGURE 23.28
The MDI OLE Demo child window.

LISTING 23.17 The Source Code for `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);
```

continues

LISTING 23.17 Continued

```
begin
  Action := caFree;
end;

end.
```

Creating a Child Form

When a new MDI child form is created from the File, New menu of the MDI OLE Demo application, the Insert Object dialog is invoked using the `InsertObjectDialog()` method mentioned earlier. Additionally, a caption is assigned to the MDI child form using a global variable called `NumChildren` to provide a unique number. The following code shows the main form's `CreateMDIChild()` method:

```
procedure TMainForm.FileNewItemClick(Sender: TObject);
begin
  inc(NumChildren);
  { create a new MDI child window }
  with TMDIChild.Create(Application) do
  begin
    Caption := 'Untitled' + IntToStr(NumChildren);
    { bring up insert OLE object dialog and insert into child }
    OleContainer.InsertObjectDialog;
  end;
end;
```

Saving to and Reading from Files

As discussed earlier in this chapter, OLE objects lend themselves to the capability of being written to and read from streams and, therefore, files. The `TOleContainer` component has the methods `SaveToStream()`, `LoadFromStream()`, `SaveToFile()`, and `LoadFromFile()`, which make saving an OLE object out to a file or stream very easy.

The `MDIOLE` application's main form contains methods for saving and opening OLE object files. The following code shows the `FileOpenItemClick()` method, which is called in response to choosing File, Open from the main form. In addition to loading a saved OLE object from a file specified by `OpenDialog`, this method also assigns the `OleFileName` field of the `TMDIChild` instance to the filename provided by `OpenDialog`. If an error occurs loading the file, the form instance is freed. Here's the code:

```
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;  // free form on error
        raise;    // reraise exception
      end;
    end;
end;
```

The following code handles the File, Save As and File, Save menu items. Note that the `FileSaveItemClick()` method invokes `FileSaveAsItemClick()` when the active MDI child does not have a name specified. Here's the code:

```
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
    { if no name is assigned, then do a "save as" }
    if TMDIChild(ActiveMDIChild).OLEFileName = '' then
      FileSaveAsItemClick(Sender)
    else
      { otherwise save under current name }
      with TMDIChild(ActiveMDIChild) do
        OleContainer.SaveToFile(OLEFileName);
end;
```


Using the Clipboard to Copy and Paste

Thanks to the universal data-transfer mechanism described earlier, it also is possible to use the Windows Clipboard to transfer OLE objects. Again, the `TOleContainer` component automates these tasks to a great degree.

Copying an OLE object from a `TOleContainer` to the Clipboard, in particular, is a trivial task. Simply call the `Copy()` method:

```
procedure TMainForm.CopyItemClick(Sender: TObject);
begin
  if ActiveMDIChild <> Nil then
```

```
    TMDIChild(ActiveMDIChild).OleContainer.Copy;
end;
```

After you think you have an OLE object on the Clipboard, only one additional step is required to properly read it out into a `TOleContainer` component. Prior to attempting to paste the contents of the Clipboard into a `TOleContainer`, you should first check the value of the `CanPaste` property to ensure that the data on the Clipboard is a suitable OLE object. After that, you can invoke the Paste Special dialog to paste the object into the `TOleContainer` by calling its `PasteSpecialDialog()` method, as shown in the following code (the Paste Special dialog is shown in Figure 23.29):

```
procedure TMainForm.PasteItemClick(Sender: TObject);
begin
  if ActiveMDIChild <> nil then
    with TMDIChild(ActiveMDIChild).OleContainer do
      { Before invoking dialog, check to be sure that there }
      { are valid OLE objects on the clipboard. }
      if CanPaste then PasteSpecialDialog;
end;
```

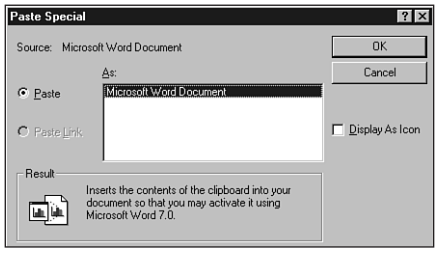

FIGURE 23.29
The Paste Special dialog box.

When the application is run, the server controlling the OLE object in the active MDI child merges with or takes control of the application's menu and toolbar. Figures 23.30 and 23.31 show OLE's in-place activation feature—the MDI OLE application is controlled by two different OLE servers.

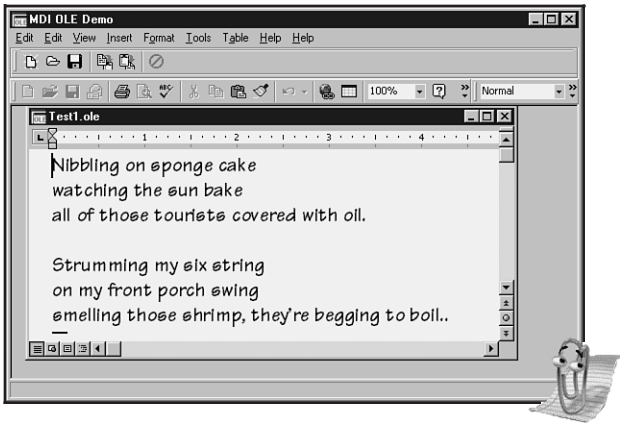

FIGURE 23.30
Editing an embedded Word 2000 document.

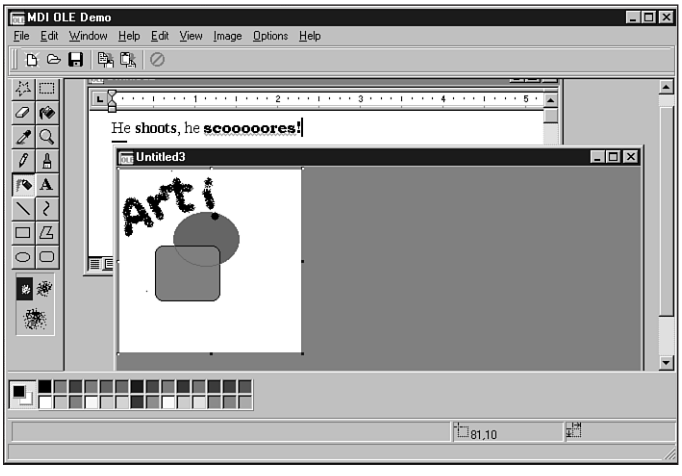

FIGURE 23.31
Editing an embedded Paint graphic.

The complete listing for `Main.pas`, the MDI OLE application's main unit, is shown in Listing 23.18.

LISTING 23.18 The source code for `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
  { Show hints on status bar }
  StatusBar.Panels[0].Text := Application.Hint;
end;

procedure TMainForm.FileNewItemClick(Sender: TObject);
begin
  inc(NumChildren);
```

continues

LISTING 23.18 Continued

```
  { create a new MDI child window }
  with TMDIChild.Create(Application) do
  begin
    Caption := 'Untitled' + IntToStr(NumChildren);
    { bring up insert OLE object dialog and insert into child }
    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;  // free form on error
        raise;    // reraise exception
      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
    { if no name is assigned, then do a "save as" }
```

```
    if TMDIChild(ActiveMDIChild).OLEFileName = '' then
      FileSaveAsItemClick(Sender)
    else
      { otherwise save under current name }
      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
      { Before invoking dialog, check to be sure that there }
      { are valid OLE objects on the clipboard. }
      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
  { Must be done backwards through the MDIChildren array }
  for I := MDIChildCount - 1 downto 0 do
    MDIChildren[I].WindowState := wsMinimized;
end;
```

continues

LISTING 23.18 Continued

```
procedure TMainForm.UpdateMenuItems(Sender: TObject);
var
  DoIt: Boolean;
begin
  DoIt := MDIChildCount > 0;
  { only enable options if there are active children }
  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;         // use Release, not Free!
    Application.ProcessMessages;    // let Windows take care of business
  end;
end;

end.
```

Summary

That wraps up this chapter on COM, OLE, and ActiveX. This chapter covered an enormous amount of information! First, you received a solid foundation in COM-based technologies, which should help you understand what goes on behind the scenes. Next, you got some insight and information on various types of COM clients and servers. Following that, you were immersed in various advanced techniques for Automation in Delphi. With all that under your belt, the chapter led you through the theory and practice of MTS. In addition to in-depth coverage of COM, Automation, and MTS, you should be familiar with the workings of VCL's `TOleContainer` component.

If you'd like to know more about COM, you'll find more information on the COM and ActiveX technologies in other areas of this book. Chapter 24, "Extending the Windows Shell," shows real-world examples of COM server creation, and Chapter 25, "Creating ActiveX Controls," discusses ActiveX control creation in Delphi.