Transactional Development with COM+/MTS

CHAPTER

18

IN THIS CHAPTER

- What Is COM+? 880
- Why COM? 880
- Services 881
- Runtime 906
- Creating COM+ Applications 908
- COM+ in Delphi 912

The release of Windows 2000 brought with it perhaps the largest single step forward for COM since its inception as the underpinnings of OLE 2.0: COM+. COM+ is the latest iteration of COM, and it ships as a standard part of Windows 2000 and Windows XP. This chapter is intended to bring you up to speed on all the various aspects of COM+ and how you can leverage its power in your Delphi applications.

What Is COM+?

Before we progress any further into describing COM+, allow us to set your mind at ease by saying this: Almost everything you know about COM still applies. After all, COM definitely takes no small degree of dedication to learn well, and it would be very disheartening to have to ride the same learning curve once again. The interesting thing about COM+ is that it isn't this strange, new monster, but merely some nice evolutionary changes to COM exist combined with the integration of some of Microsoft's COM-based services that you might already be familiar with. In plain English, COM+ can be boiled down to this: COM with a few new features, integrated with *Microsoft Transaction Server (MTS*) and *Microsoft Message Queue (MSMQ)*.

Because COM+ is based on and fully backward compatible with COM, you have no worries from a Delphi perspective. Delphi works just as great with COM+ as it does with COM. To build optimized COM+ components, there are certainly a few fundamental additions you'll need to know about, particularly with regard to a new type of components called configured that we'll discuss later. But, it's important for you to know that the entire world of COM+ is available to you as a Delphi developer.

Why COM?

Why did Microsoft choose to base COM+ on COM, rather than moving it to some completely different direction? This is a fair question, especially in light of some of the negative comments we all might hear about COM in its skirmishes with competing technologies such as CORBA and *Enterprise Java Beans (EJB)* in the battlefields of the industry tabloids. Not only is COM a good foundation to build on technologically, but also a business case around COM is very compelling when you consider that

- COM is programming language independent.
- COM is supported by every major Windows development tool.
- Every 32-bit Windows user is already running COM, which puts the installed base at somewhere around 150 million users (according to Microsoft).
- The Giga Information Group recently reported that COM is a $670 million market (not including Microsoft).

Probably the biggest drawback of COM is its reputation for being difficult to scale to large numbers of users involved in large numbers of transactions. In Microsoft fashion, a major intent of COM+ is to leverage the assets s, while attempting to eliminate the liabilities.

We can classify COM+ features into three distinct categories: administration, services, and runtime. Administration is primarily handled in the Component Services administration tool, which is discussed throughout this chapter. We will tackle the discussion of services and runtime in turn. Because services make up the bulk of the new features in COM+, we'll discuss those first.

Services

COM+ services are the things that we today consider to be add-ons to COM. Technology currently found in MTS and MSMQ, for example, make up some of the services found in COM+. Think of services as systems built by Microsoft on top of COM+ designed to somehow add value to component-based development. As we mentioned, some services, such as transactions and queued components, are present thanks to off-the-shelf technology. Consequently, if you have experience with these technologies already, you'll have an advantage as you begin to write COM+ applications. Other services, however, such as object pooling and late-bound-events are probably new to you and might take some getting used to.

Transactions

As the "T" in MTS, it should be no surprise to find transactions playing a major role in COM+. COM+ implements the MTS model for transactions, which is described in greater detail later in this chapter. Without transaction support, there is no way that collection objects would be able to support a complicated business application. For example, a transaction involving an online purchase of some item might involve the participation of several objects communicating with one or more databases to receive the request, check inventory, debit the credit card, update the accounting ledger, and issue a ship order. All these things needs to happen in concert; if something goes wrong in any of these processes, the state of all objects and data needs to be rolled back to the state they were in before the entire transaction began. As you can imagine, this process of managing transactions is even more complicated when the objects involved are spread across multiple machines.

Transactions are controlled centrally by the MS *Distributed Transaction Coordinator (DTC)*. When a COM+ application calls for transactions, the DTC will enlist the assistance of and coordinate other software elements, including transaction managers, resource managers, and resource dispensers. Each computer participating in a transaction has a transaction manager that tracks transaction activity on that specific machine. Transaction managers, however, are ignorant of data because persistent information such as database data or message queue

messages are managed by a resource manager. A resource dispenser manages non-persistent state information, such as database connections. Each of these specialized elements managed by the DTC knows how to commit and recover its specific resource.

Security

As the introduction of one new technology quickly follows another in today's insanely paced world of software development, we occasionally reflect with fond remembrance on the olden days of PC software development, when applications consisted of a .EXE or a .COM file and a network was a place to share data files with your co-workers. Business applications today often consist of multiple types of user interfaces (Windows, Web-based, Java, and so on) communicating with software components distributed across a network, which in turn communicates with one or more database servers on the network. Our success as developers is now linked not only to our ability to tie disparate application elements together, but also to provide a means by which they can communicate in privacy. This means building security into distributed applications that enables components to authenticate one another, determine what services they should offer one another, and provide a means for private communication between one another.

The notion of security has become common sense at this point. We all understand that most data needs to be protected; for example, human resources data shouldn't be accessible to all employees, sales data shouldn't be accessible to your competitors, and so on. Equally, component functionality also needs to be secure; perhaps only administrators should have the right to use certain objects or only department managers should have access to a particular business rules engine. In practice, however, building this type of security into distributed applications can be a time-consuming process, and security features naturally take a backseat to core functionality in project schedules.

COM+ provides a well-constructed set of security features that addresses many of these issues. COM+ makes security more of an administrative issue than a programmatic one, and therefore helps you to spend your time developing application logic and less time writing security code. Configuring COM+ application security in the Component Services administration tool is a one-time process, and your application can remain free of security-specific code. At the same time, COM+ does provide APIs for accessing security information for cases in which you do need to go beyond the provided functionality. My goal here is to provide you with an overview of security architecture for COM+ server applications and how to use security in your COM+ applications.

Role-Based Security

COM+'s security architecture is often referred to as role-based. Rather than managing accounts for individual users, COM+ applications rely on categories or groups of users referred to as

roles. Roles work hand-in-hand with the operating system-based security because the members of roles are the user accounts on the Windows 2000 server or domain. Roles can be created on an application-by-application basis using the Component Services administration tool, and the process is rather straightforward. This is done by right-clicking on the Roles node of the COM+ application in the treeview on the left in the Component Services administration tool. After a role has been added, another right-click can add users to the role. Figure 18.1 illustrates the process of adding users to a role.

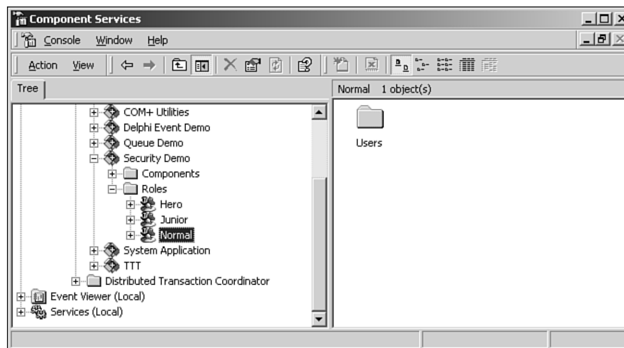

FIGURE 18.1
Using the Component Services administration tools to configure roles.

You can see from Figure 18.1 that in this example, the COM+ application has three roles: Junior, Normal, and Hero. These are simply names made up to indicate three different groups of users we plan to provide differing functionality for in our COM+ application. Noteworthy is the fact that the actual authentication is handled automatically by the OS, and COM+ builds on top of those services.

Role-Based Security Configuration

Arguably the slickest aspect of COM+'s role-based security system is that security can be established at the application, component, interface, or even method level! This means that you can control which roles have access to which methods without writing a line of code.

The first step to configuring COM+ application security is to enable security at the application level. This is done by editing the properties of the application in the Component Services administration tool and switching to the Security tab, which is shown in Figure 18.2.

Application security is enabled when the Enforce Access Checks For This Application check box is checked. This dialog also enables selection of the security level, which can be set to perform security checking at the process level only or at the process and component level.

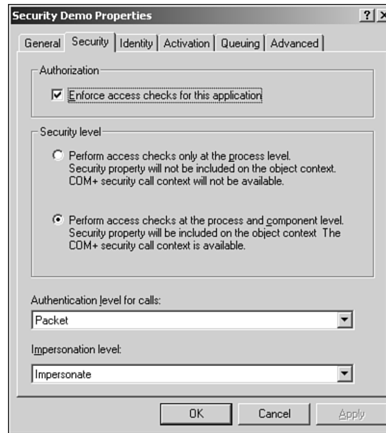

FIGURE 18.2
Configuring COM+ application security.

Enabling security only at the process level has the effect of locking the front door to the COM+ application, where all members of roles assigned to the application have the key to that door. When this option is selected, no security checking will be performed on the component, interface, or method level, and security context information will not be maintained for objects running in the application. This type of security is useful when you don't need granular security control, but simply want to limit overall access to the COM+ application to a specific group of users. This type of security also has the advantage in increased performance because security checks don't need to be made by COM+ during execution of the application.

Enabling security at the process and component level ensures that role-based security checks will be made at the component, interface, and method level and security context information will be available to objects in the application. Although this provides maximum control and flexibility, note that a performance of your COM+ application will suffer slightly because of the increased level of management that COM+ will need to perform during execution.

The security properties dialog box shown in the Figure 18.2 also provides for configuration of the authentication level of the COM+ application. The authentication level determines the degree to which authentication is performed on client calls into the application. Each successive authentication level option provides for a greater level of security, and the options are shown in Table 18.1.

TABLE 18.1 COM+ Authentication Levels

Level	*Description*
None	No authentication occurs.
Connect	Authenticates credentials only when the connection is made.
Call	Authenticates credentials at the beginning of every call.
Packet	Authenticates credentials and verifies that all call data is received. This is the default setting for COM+ server applications.
Packet Integrity	Authenticates credentials and verifies that no call data has been modified in transit.
Packet Privacy	Authenticates credentials and encrypts the packet, including the data and the sender's identity and signature.

Note that authentication requires the participation of the client as well as the server. COM+ will examine the client and the server preference for authentication and will use the maximum of the two. The client authentication preference can be set using any one of the following techniques:

- The machine-wide setting specified in the Component Services administration tool (or DCOMCNFG on non-Windows 2000/XP machines)
- The application-level setting specified in the Component Services administration tool (or DCOMCNFG on non-Windows 2000/XP machines)
- The process-level setting specified programmatically using the `CoInitializeSecurity()` COM API call
- An on-the-fly setting that can be specified programmatically using the `CoSetProxyBlanket()` API

Finally, the properties security dialog box shown in Figure 18.2 allows configuration of the application impersonation level. The impersonation level setting dictates to what degree the server application might impersonate its client in order to access other resources on behalf of clients. Table 18.2 explains the options for impersonation level.

TABLE 18.2 COM+ Impersonation Levels

Level	*Description*
Anonymous	The client is anonymous to the server.
Identify	The server can obtain the client's identity, and can impersonate the client only to perform Access Control checking.

TABLE 18.2 Continued

Level	*Description*
Impersonate	The server can impersonate the client while acting on its behalf, although with restrictions. The server can access resources on the same computer as the client. If the server is on the same computer as the client, it can access network resources as the client. If the server is on a computer different from the client, it can only access resources that are on the same computer as the server. This is the default setting for COM+ server applications.
Delegate	The server can impersonate the client while acting on its behalf, whether or not on the same computer as the client. During impersonation, the client's credentials can be passed to any number of machines. This is the broadest permission that can be granted.

Like authentication, impersonation can also only be accomplished with the consent of the client. The client's consent and preferences can be established exactly the same as authentication, using Component Services administration tool, DCOMCNFG, or the `CoInitializeSecurity()` and `CoSetProxyBlanket()` APIs.

After application security has been configured, security can then be configured for components, interfaces, and methods of the application. This is done in a similar manner by editing the properties of the item in the tree and choosing the Security tab. This will invoke a dialog box with a page similar to that shown in Figure 18.3.

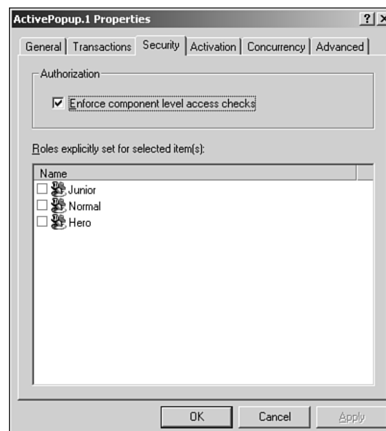

FIGURE 18.3
Configuring COM+ component security.

The dialog box shown in Figure 18.3 is fairly straightforward; it enables you to specify whether security checks should be enabled for the item and which roles are to be allowed access to the item.

Multitier Performance

When designing multitier applications that employ COM+ security, there are a number of performance considerations you should weigh. First and foremost, always bear in mind that one of the primary goals of a multitier system is to improve overall system scalability. One mistake that often compromises scalability and performance is over securing an application by implementing security at multiple tiers. A better solution would be to leverage COM+ services by implementing security only or mostly at the middle tier. For example, rather than impersonating the client in order to gain access to a database, it is more efficient to access the database using a common connection that can be pooled among multiple clients.

Programmatic Security

Up until now, we've focused primarily on declarative (or administration-driven) security; however we did mention that it is also possible to program security into COM+ applications. The most common thing you might want to do is determine whether the caller of a particular method belongs to a specific role. This enables you to control not only method access, but also method behavior, based on the role of the client. To serve this purpose, COM+ provides not one but two means for making this determination. There is a method of `IObjectContext` called `IsCallerInRole()`, which is defined as

```
function IsCallerInRole(const bstrRole: WideString): Bool; safecall;
```

This function is used by passing the name of the role in the `bstrRole` parameter, and it will return a Boolean value indicating whether the current caller belongs to the specified role. A reference to the current object context can be found by calling the `GetObjectContext()` API, which is defined as

```
function GetObjectContext: IObjectContext;
```

The following code checks to see if the caller is in the `Hero` role prior to performing a task:

```
var
  Ctx: IObjectContext;
begin
  Ctx := GetObjectContext;
  if (Ctx <> nil) and (Ctx.IsCallerInRole('Hero')) then
  begin
    // do something interesting
  end;
end;
```

Similarly, an `IsCallerInRole()` method is also found on the `ISecurityCallContext` interface, a reference that can be obtained using the `CoGetCallContext()` API. This version of the method is actually preferred, simply because `ISecurityCallContext` makes handy a lot of other security information, such as the caller and its authentication and impersonation level.

Just-In-Time Activation

Just-In-Time (JIT) activation refers to functionality already present in COM+ that enables an object to be transparently destroyed and re-created without the knowledge of the client application. JIT activation potentially enables a server to handle a higher volume of clients because resources used by an object can be reclaimed by the system when it is deactivated.

The object developers has full control over when an object is deactivated, and objects should only be deactivated when they have no state to maintain. An object can be deactivated using the `SetComplete()` or `SetAbort()` methods of `IObjectContext` or the `SetDeactivateOnReturn()` method of `IContextState`.

Queued Components

Delphi developers normally don't have to be lectured on the benefits of briefcase model applications. When MIDAS was introduced in Delphi 3, the barrier of entry was forever lowered for creating applications having the capability to operate even when the client is disconnected from the server. Delphi developers quickly realized the power of enabling their users to work with their data in a disconnected, briefcase model, and embraced MIDAS as well as other technologies that provide this capability. Rather than having to write complicated code to, for example, enable a salesman to edit his customer database on his laptop while on the road and synchronize when he gets back into the office, this functionality is now easily accessible simply by dropping a few components and writing a few lines of code.

This is all really great if you happen to be data, but what to do if you're an object? As object remoting technologies such as DCOM, MTS/COM+, and CORBA become easier to implement in our tools, our reliance on such technologies increases as we build solutions for our companies and clients. Consequently, this reliance increases as we employ object remoting technologies to build ever-more-complex distributed applications. As a result of all this, distributed component applications—like data applications—also have the need to function when disconnected from servers.

Queued Components: The Object Briefcase

COM+ queued components answer this need. Based on *MSMQ (Microsoft Message Queue)* technology, queued components provide a means for COM+ clients to asynchronously invoke methods of COM+ server components. In essence, this means that clients can create instances

of server objects and invoke their methods without regard to whether the server is actually accessible to the client. COM+ manages this by storing the method invocations in a queue and executing the methods at a later time when the server is accessible. What's more, the server objects likewise have little reason to know or care whether their methods are being invoked directly or via a COM+ queue. Our goal here is to cover the essential elements of working with COM+ queued components.

Figure 18.4 illustrates how queued components are internally implemented. When the client makes a method call on a queued component, that method call is captured by the recorder, which packages up the call and parameters and places it into a queue. Because the client has no knowledge that it isn't actually communicating with the server, you can see that the recorder server as a sort of a proxy for the server. The recorder knows how to behave because it obtains information on the server from its type library and its configuration or registration information. The listener removes the message, which contains the call information, from the queue and passes it on to the player. Finally, the player unpackages the call information (along with related information, such as the client's security context) and executes the method call on the server.

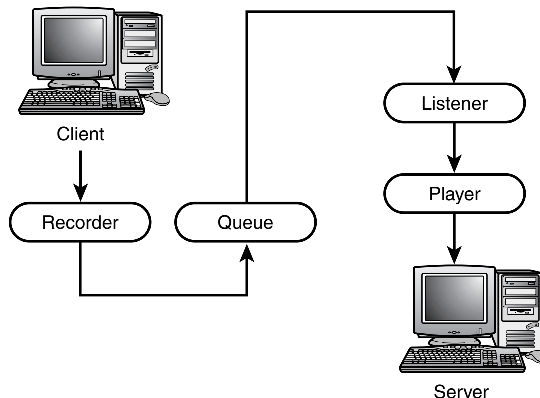

FIGURE 18.4
COM+ queued component architecture.

"All this sounds cool," you might be saying to yourself, "but I'll bet implementing it requires a degree in some new variety of non-Newtonian physics." If you did say that to yourself, you're only half right; it is cool, but it's also very easy to do, as you will soon see.

Why Queued Components?

Before jumping into implementation, however, we'd like to address some of the specific reasons for using queued components.

- System scalability—In a non-queued system, there will be a finite number of server objects capable of handling requests from clients at any given time. When all these objects become tied up handling client calls, other incoming client calls will be blocked until an object finishes and again becomes available. In a system having a large number of simultaneous transactions, this can seriously limit the number of concurrent clients that can be serviced. Using queues, the call always returns immediately to the client after being queued and played back to servers in the servers' own time. This enables the system to handle a greater number of concurrent transactions.

  Scalability is also increased on the back end because the client doesn't manage the lifetime of the server. Rather than being active while the client carries on with its processing and various method calls, a queued server only needs to be active while calls are being played back by the recorder. Reducing the amount of time a server needs to remain in memory means that a greater number of servers can be activated over a given period of time with a given amount of RAM.
- Briefcase model—As we mentioned, COM+ enables queued components to behave in a disconnected manner in much the same way MIDAS does for data. This enables clients to work without being connected to their network and method calls to be played back to the server when the client connects to the network at a later time.
- Fail-safety—If you are creating a mission-critical application that requires a high degree of availability, such as an e-commerce storefront, the last thing you want to happen is for the system to go down because your front end is having trouble communicating with server objects. Queued components provide an ideal safety net to prevent this problem because they will queue method calls intended for servers if the servers become unavailable and play them back when the server again comes online.
- Load scheduling—Rather than having your servers work like rented mules during their peak hours of activity and sit nearly dormant during the other hours of the day, using queued components you can spread processing throughout the day to even the workflow and place less demand on your servers at any specific time.

Creating a Server

There's little difference between creating a queued component and a creating normal COM/COM+ component. The biggest adjustment you will need to make is that all methods on queued interfaces must accept only in parameters and must not make use of return values. Of course, these limitations make perfect sense when you consider the fact that the client won't be sitting around waiting for the server to return any values or out parameters. Also, you will need to perform a few extra steps as far as component configuration at install time.

To illustrate, we will create a Delphi server that contains one COM+ class with one interface with one method. To make life easier, we'll get started using the Automation Object Wizard

accessible via the File, New Main menu item. We call this object `QTest`, and the wizard automatically names the primary interface `IQTest`. (Don't worry, it's easier than it sounds.) To the `IQTest` interface we add one method, which is defined in the type library editor as follows:

```
procedure SendText(Value: WideString; Time: TDateTime) [dispid $00000001];
  safecall;
```

The idea is that this method takes two parameters: the first a string message and the second the time on the client when the method was called. Our implementation of this method simply writes this information, in addition to the time the message was processed by the server, to a log file we create called `c:\queue.txt`. The implementation file for this Automation object is shown Listing 18.1.

LISTING 18.1 `TestImpl.pas`—Implementation of Queued Object

```
unit TestImpl;

interface

uses
  Windows, ComObj, ActiveX, Srv_TLB, StdVcl;

type
  TQTest = class(TAutoObject, IQTest)
  protected
    procedure SendText(const Value: WideString; Time: TDateTime); safecall;
  end;

implementation

uses ComServ, SysUtils;

procedure TQTest.SendText(const Value: WideString; Time: TDateTime);
const
  SFileName = 'c:\queue.txt';
  SEntryFormat = 'Send time:  %s'#13#10'Write time: %s'#13#10 +
    'Message: %s'#13#10#13#10;
var
  F: THandle;
  WriteStr: string;
begin
  F := CreateFile(SFileName, GENERIC_WRITE, FILE_SHARE_READ, nil, OPEN_ALWAYS,
    FILE_ATTRIBUTE_NORMAL, 0);
  if F = INVALID_HANDLE_VALUE then RaiseLastWin32Error;
  try
```

LISTING 18.1 Continued

```
    FileSeek(F, 0, 2);  // go to EOF
    WriteStr := Format(SEntryFormat, [DateTimeToStr(Time),
      DateTimeToStr(Now), Value]);
    FileWrite(F, WriteStr[1], Length(WriteStr));
  finally
    CloseHandle(F);
  end;
end;

initialization
  TAutoObjectFactory.Create(ComServer, TQTest, Class_QTest,
    ciMultiInstance, tmApartment);
end.
```

After the server has been created, it needs to be installed into a new COM+ application using either the Component Services management tool or the COM+ Administration Library API. Using the Component Services tool, the first step is to create a new empty application by selecting that option from the local menu of the COM+ Applications node in the tree and following the prompts. Once the application has been created, the next step is to edit the application's properties to mark the application as queued, as shown in Figure 18.5. We also chose to enable queue listening on this application so that it would immediately play any incoming messages on its queue when it is active. The configuration is shown in Figure 18.5.

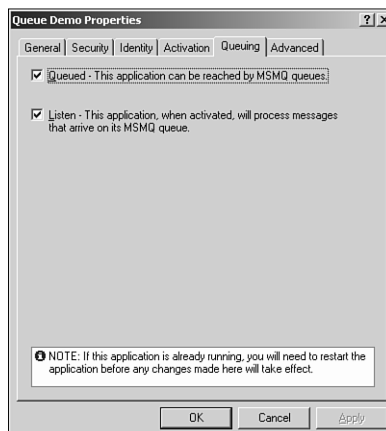

FIGURE 18.5
Configuring a queued COM+ application.

To install the server into the COM+ application, select New, Component from the local menu of the Component node of the application in the tree. This invokes the COM Component Install Wizard, which can install a new component using the defaults and select the name of the COM+ server DLL created earlier. After installation into the application, edit the properties of the `IQTest` interface on this object to support queuing as shown in Figure 18.6.

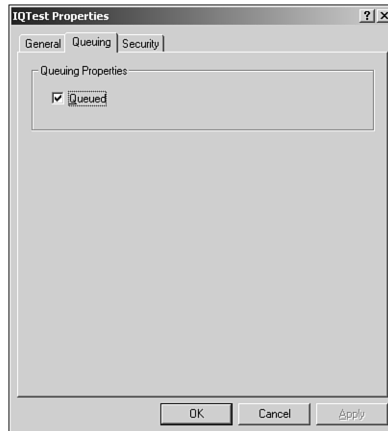

FIGURE 18.6
Specifying an interface as queued.

Note that COM+ requires that queuing be enabled on both the COM+ application and the interface level.

Creating a Client

The workflow for creating a queued component client is identical to creating a client of any old Automation client. In this case, create an application with a main form as shown in Figure 18.7.

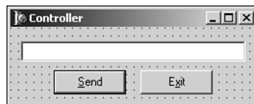

FIGURE 18.7
A client application for a queued component.

When the Send button is clicked, the contents of the edit is sent to the server via its `SendText()` method. The code for this unit corresponding to this form is shown in Listing 18.2.

LISTING 18.2 `Ctrl.pas`—The Main Unit for a Queued Component Client

```
unit Ctrl;

interface

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

type
  TControlForm = class(TForm)
    BtnExit: TButton;
    Edit: TEdit;
    BtnSend: TButton;
    procedure BtnExitClick(Sender: TObject);
    procedure BtnSendClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    FIntf: IQTest;
  end;

var
  ControlForm: TControlForm;

implementation

{$R *.DFM}

uses ComObj, ActiveX;

// Need to import CoGetObject because import in the ActiveX unit is incorrect:
function MyCoGetObject(pszName: PWideChar; pBindOptions: PBindOpts;
  const iid: TIID; out ppv): HResult; stdcall;
  external 'ole32.dll' name 'CoGetObject';

procedure TControlForm.BtnExitClick(Sender: TObject);
begin
  Close;
end;

procedure TControlForm.BtnSendClick(Sender: TObject);
begin
  FIntf.SendText(Edit.Text, Now);
  Edit.Clear;
end;
```

LISTING 18.2 Continued

```
procedure TControlForm.FormCreate(Sender: TObject);
const
  SMoniker: PWideChar = 'queue:/new:{64C576F0-C9A7-420A-9EAB-0BE98264BC9D}';
begin
  // Create object using a moniker that specifies queued creation
  OleCheck(MyCoGetObject(SMoniker, nil, IQTest, FIntf));
end;

end.
```

The only element in this unit that sets it apart from a standard Automation controller is the means by which it creates the server object instance. Rather than using, for example, the `CoCreateInstance()` COM API function, this client uses the `CoGetObject()` API. `CoGetObject()` enables an object to be created via a moniker, and COM+ allows a special string moniker syntax that can be used to invoke components in a queued manner. The general syntax of this moniker is `queue:/new:` followed by the CLSID or program ID of the server object. The following are all examples of properly formatted queue monikers:

```
queue:/new:Srv.IQTest
queue:/new:{64C576F0-C9A7-420A-9EAB-0BE98264BC9D}
queue:/new:64C576F0-C9A7-420A-9EAB-0BE98264BC9D
```

There are also a number of queue moniker parameters that you can incorporate into the string to modify the destination queue or queue behavior. The following list describes these moniker parameters:

- `ComputerName`—The parameter's value is the string name of the computer containing the queue. Specifies the computer name portion of a queue pathname. If not specified, the computer name associated with the configured application is used.
- `QueueName`—The parameter's value is the string name of the queue on the target server machine. Specifies the queue name. If not specified, the queue name associated with the configured application is used.
- `PathName`—The queue pathname must be formatted as `ComputerName\QueueName`. Specifies the complete queue pathname. If not specified, the queue pathname associated with the configured application is used.
- `FormatName`—The parameter's value is the format name of queue, for example, `DIRECT=9CA3600F-7E8F-11D2-88C5-00A0C90AB40E`. Specifies the queue format name.
- `AppSpecific`—For example, `AppSpecific=8675309`. An unsigned integer design for application-specific use.

- `AuthLevel`—`MQMSG_AUTH_LEVEL_NONE` (0) or `MQMSG_AUTH_LEVEL_ALWAYS` (1). Specifies the message authentication level. An authenticated message is digitally signed and requires a certificate for the user sending the message.
- `Delivery`—`MQMSG_DELIVERY_EXPRESS` (0) or `MQMSG_DELIVERY_RECOVERABLE` (1). Specifies the message delivery option. Ignored for transacted queues.
- `EncryptAlgorithm`—`CALG_RC2`, `CALG_RC4`, or other integer value recognized by COM+ as an identifier representing an acceptable encryption algorithm. Specifies the encryption algorithm to be used by COM+ to encrypt and decrypt the message.
- `HashAlgorithm`—`CALG_MD2`, `CALG_MD4`, `CALG_MD5`, `CALG_SHA`, `CALG_SHA1`, `CALG_MAC`, `CALG_SSL3_SHAMD5`, `CALG_HMAC`, `CALG_TLS1PRF`, or other integer value recognized by COM+ as acceptable. Specifies a cryptographic hash function.
- `Journal`—`MQMSG_JOURNAL_NONE` (0), `MQMSG_DEADLETTER` (1), or `MQMSG_JOURNAL` (2). Specifies the COM+ queue message journal option.
- `Label`—Any string. Specifies a message label string up to `MQ_MAX_MSG_LABEL_LEN` characters.
- `MaxTimeToReachQueue`—`INFINITE`, `LONG_LIVED`, or an integer value indicating a specific number of seconds. Specifies a maximum time, in seconds, for the message to reach the queue.
- `MaxTimeToReceive`—`INFINITE`, `LONG_LIVED`, or an integer value indicating a specific number of seconds. Specifies a maximum time, in seconds, for the message to be received by the target application.
- `Priority`—`MQ_MIN_PRIORITY` (0), `Q_MAX_PRIORITY` (7), `MQ_DEFAULT_PRIORITY` (3), or any integer between 0 and 7. Specifies a message priority level, within the MSMQ values permitted.
- `PrivLevel`—`MQMSG_PRIV_LEVEL_NONE`, `NONE`, `MQMSG_PRIV_LEVEL_BODY`, `BODY`, `MQMSG_PRIV_LEVEL_BODY_BASE`, `BODY_BASE`, `MQMSG_PRIV_LEVEL_BODY_ENHANCED`, or `BODY_ENHANCED`. Specifies the privacy level that is used to encrypt messages.
- `Trace`—`MQMSG_TRACE_NONE` (0) or `QMSG_SEND_ROUTE_TO_REPORT_QUEUE` (1). Specifies trace options, used in tracing COM+ queue routing.

Using some of these options, other valid queue monikers might be

```
queue:Priority=6,ComputerName=foo/new:{64C576F0-C9A7-420A-9EAB-0BE98264BC9D}
queue:PathName=drevil\myqueue/new:{64C576F0-C9A7-420A-9EAB-0BE98264BC9D}
```

Running the Server

After invoking the client and typing a few strings into the edit, you can check for yourself on your hard disk, and you will see that the file `c:\queue.txt` isn't present on your hard disk.

That is because the server application needs to start running before queued messages will be played back. Three ways to start the server are as follows:

1. Manually—Using the Component Services tool. This can be done simply by selecting Start from the local menu of the application node in the tree.
2. Programmatically—Using the COM+ Administration Library API.
3. Scheduled—Using scripting. This can be done using a script similar to the following in the task scheduler:

```
dim cat
set cat = CreateObject("COMAdmin.COMAdminCatalog");
cat.StartApplication("YourApplication");
```

After starting the application, you will see the `c:\queue.txt` file present on your hard disk. Its contents will look something like this:

```
Send time:  7/6/2001 7:15:08 AM
Write time: 7/6/2001 7:15:18 AM
Message: this is a test

Send time:  7/6/2001 7:15:10 AM
Write time: 7/6/2001 7:15:18 AM
Message: this is another
```


Object Pooling

You might remember that wacky `CanBePooled()` method of `IObjectControl` that MTS simply ignored. The good news is that `CanBePooled()` is no longer ignored, and COM+ does support object pooling. Object pooling provides the ability to keep a pool of some particular number of instances of a particular object, and have the objects in this pool used by multiple clients. Similar to JIT activation, the goal is to increase overall throughput of the system. However, JIT activation carries the assumption that objects aren't expensive to create or destroy (because it is done frequently). If an object is expensive to create or destroy, it makes more sense to keep instances around after their creation by pooling them.

A number of limitations are imposed on objects that want to support pooling. These include

- The object must be stateless so that it maintains no instance-specific data between method calls.
- The object must have no thread affinity. That is, they shouldn't be bound to any particular thread and they shouldn't use thread local storage (TLS, or "threadvar" variables in the Delphi world).
- The object must be aggregatable.

- Resources must be manually enlisted in transactions. The resource manager cannot automatically enlist resources on the object's behalf.
- The object must implement `IObjectControl`.

Events

Delphi developers don't need to be sold on the importance of events. How else would we know when a button was clicked or a record posted? However, although COM developers have also been aware of the importance of events, they often avoided them because of the complexity of implementation. COM+ introduces a new event model, which—thank heavens—isn't tied to the Byzantine connection points model that has been common in COM to this point.

The typical picture we imagine when we think about the relationship between COM client and server objects is fairly linear; clients invoke methods on servers and servers do useful things in response to the client call and optionally provide some data back to the client in the form of a return value and out parameters. It's probably true that this relationship is an accurate representation of probably more than 90% of COM client/server interactions, but you don't have to be a COM guru to realize that this model is limited, particularly with regard to clients having the ability to be quickly updated when some server data changes.

The simplest way to obtain such a notification would be for clients to poll servers on a periodic basis in order to check whether the information in which they're interested changes. However, the disadvantages of polling are pretty self-evident; clients waste a lot of cycles sending polls, servers likewise waste a lot of clocks responding to polls, extraneous network traffic can be generated, and the overall scalability of the system is diminished to the sum of all this increased load on client, server, and wire.

More desirable, but still low tech, is a system whereby clients can pass servers one or more predefined interfaces to call back on when the information in question changes. However, this system essentially has to be re-invented for every different interface you want to use, and it is incumbent upon the server to write specialized code to track multiple client connections.

Traditional COM provides a more efficient and structured solution to this problem, called events. This solution involves the use of the connection points, which provide servers with the capability to track clients that want to be notified of information changes as well as the means for servers to call client methods to make the notifications. Connection points are an example of what is known as a *tightly coupled event (TCE)* system. In a TCE system, clients and servers are mutually aware of the other's identity. Additionally, TCE systems require that clients and servers be running simultaneously, and they provide no means for filtering of events. The connection point system also has the inherent disadvantages because it is rather complex to implement and use, and clients are forced to implement entire event interfaces, even if they are only interested in a single method of the interface.

COM+ contains a new event system that solves some of these problems and adds some nice additional features. The COM+ event model is known as a *Loosely Coupled Event (LCE)* system. It is referred to as such because there is no hard connection between servers (known as *event publishers*) and clients (known as *event subscribers*). Instead, publishers register with the COM+ catalog the events they want to publish, and subscribers separately register with the COM+ catalog the events in which they are interested. When a publisher fires an event, the COM+ runtime reviews its database to determine which clients should receive an event notification and sends the notification to those clients. What's more, clients don't even have to be running when the event is fired; COM will activate clients upon invocation of the event. Additionally, the event registration model supports method-level granularity. This means that subscribers aren't forced to implement methods for events for which they have no interest. Figure 18.8 provides an illustration of the COM+ event system.

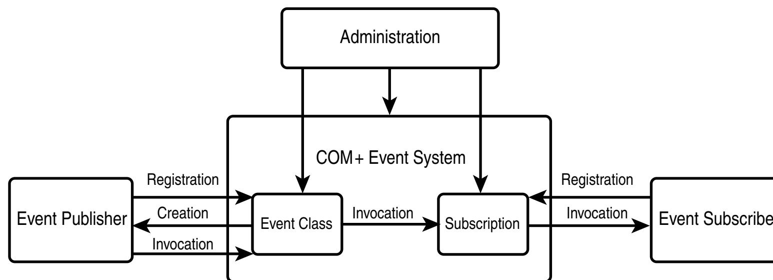

FIGURE 18.8
The COM+ event system architecture.

As Figure 18.8 shows, the process begins when the publisher registers a new event class. This can be done using the Component Services administration tool or using the `ICOMAdminCatalog.InstallEventClass()` method. Once registered, the object that implements the event class will reside in the COM+ runtime. The publisher or another object can then call the `CoCreateInstance()` COM API call to create an instance of this object and call methods on this object to fire events.

On the subscriber side, the subscriber can register for an event class permanently, using the Component Services administration tool, or in a transient manner using the COM admin catalog API. Permanent subscription means that the subscribing component doesn't need to be active when the event fires; the COM+ runtime will automatically create the component before invoking the event. Transient subscriptions are intended for already active components that want to receive event notifications only temporarily. When the publisher fires an event, COM+ will iterate over all the registered subscribers, invoking the event on each. Note that it isn't possible to determine the order in which COM+ will iterate over the clients when invoking an

event. However, it is possible to gain some control over the firing of events using event filters, which we describe in more detail later.

Speaking practically, creating a COM+ event can be boiled down to a five-step process:

1. Creating an event class server
2. Registering and configuring the event class server
3. Creating a subscriber server
4. Registering and configuring the subscriber servers
5. Publishing of events

We'll take these steps one at a time to demonstrate a Delphi implementation of COM+ events.

Creating an Event Class Server

The first step to creating an event class server is to create an in-process COM server to which you will add a COM object. The important distinction to bear in mind between creating an event class server and creating a regular COM server is that an event class server carries with it no implementation—it only serves as a vehicle for definition of the event class.

You create an event class server in Delphi by using the ActiveX Library wizard to create a new COM server DLL and the Automation Object wizard to generate the event class and interface. Call this object `EventObj`. The wizards leave you off in the Type Library Editor in order to complete the definition of the server, where you add a method called `MyEvent` to the `IEventObj` interface that will serve as the event method. The implementation file produced for this type library is shown in Listing 18.3.

LISTING 18.3 `PubMain.pas`—The Main Unit for an Event Class Server

```
unit PubMain;

interface

uses
  ComObj, ActiveX, Publisher_TLB, StdVcl;

type
  TEventObj = class(TAutoObject, IEventObj)
  protected
    function MyEvent(const EventParam: WideString): HResult; safecall;
  end;

implementation
```

LISTING 18.3 Continued

```
uses ComServ;

function TEventObj.MyEvent(const EventParam: WideString): HResult;
begin

end;

initialization
  TAutoObjectFactory.Create(ComServer, TEventObj, Class_EventObj,
    ciMultiInstance, tmApartment);
end.
```

That's all there is to creating the event class server. Note that it's not necessary to register this server. Registration is handled specially, and we discuss it in the next step.

Registration and Configuration of the Event Class Server

In this phase, we will again make use of the Component Services administration tool. You'll use this tool often as you develop COM+ applications. You'll find this tool in the Administrative Tools group of the Programs section of the Start menu. The first thing you'll need to do in the Component Services administration tool is create a new COM+ application. You can do this by selecting New, Application from the local menu of the COM+ Applications node in the tree view on the left. This will invoke the COM+ Application Install Wizard as shown in Figure 18.9. In this wizard, I choose to create a new application from scratch and call it Delphi Event Demo.

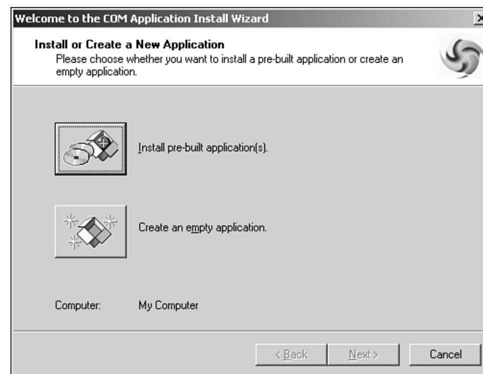

FIGURE 18.9
Using the Component Services tool to add a COM+ application.

After the COM+ application has been installed, you can install the event class server into the application. This is done by selecting New, Component from the local menu of the Components node under the new application in the tree. This invokes the COM Component Install Wizard, a frame of which is shown in Figure 18.10.

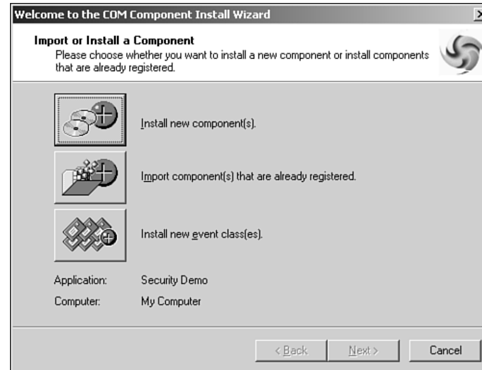

FIGURE 18.10
Using the Component Services tool to add a COM+ component.

In this wizard, install a new event class, and select the filename of the event class server that was just created. With that done, it's time to move on to the creation of the subscriber server.

Creation of a Subscriber Server

A subscriber server is essentially a standard Delphi Automation server. The only catch is that you need to implement the event interface that you defined when creating the event class server. We accomplish this by using the type library from the event class server in the subscriber server and adding the `IEventObj` interface to the implements list of the co-class. Figure 18.11 shows the `SubObj` coclass, containing both `ISubObj` and `IEventObj`, and the implementation file for this type library is shown in Listing 18.4.

LISTING 18.4 `SubMain.pas`—The Implementation Unit for the Event Server

```
unit SubMain;

interface

uses
  ComObj, ActiveX, Subscriber_TLB, StdVcl, Publisher_TLB;

type
  TSubObj = class(TAutoObject, ISubObj, IEventObj)
  protected
```

LISTING 18.4 Continued

```
    function MyEvent(const EventParam: WideString): HResult; safecall;
    { Protected declarations }
  end;

implementation

uses ComServ, Windows;

function TSubObj.MyEvent(const EventParam: WideString): HResult;
begin
  MessageBox(0, PChar(string(EventParam)), 'COM+ Event!', MB_OK);
  Result := S_OK;
end;

initialization
  TAutoObjectFactory.Create(ComServer, TSubObj, Class_SubObj,
    ciMultiInstance, tmApartment);
end.
```


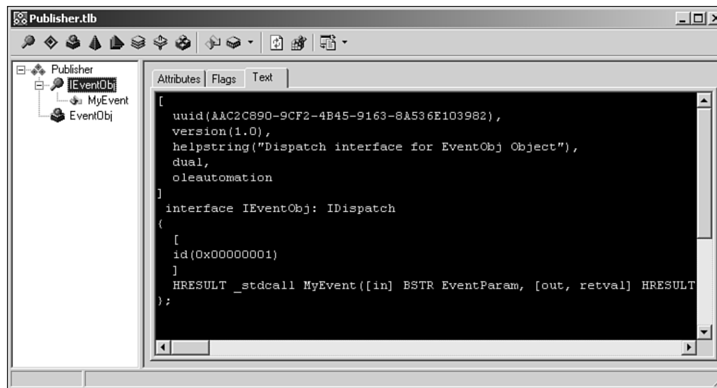

FIGURE 18.11
The `IEventObj` *interface in the type library editor.*

You can see that the implementation of the event is quite earth shattering; a message box is displayed showing a real, live text string! Again, there is no need to register this server as you would a standard COM server. That housekeeping is handled in the next step.

Registration and Configuration of the Subscriber Servers

To register the subscriber server, reopen the Component Services administration tool, and choose New, Component from the local menu just as you did for the event class server. The

difference is that this time you should choose to install a new component in the COM Component Install Wizard and select the subscriber DLL.

After the subscriber server is installed, you can create a new subscription for the subscriber server by selecting New, Subscription from the Subscriptions node under your new subscriber server. This brings up the New Subscription Wizard, which allows you to define the correlation between the publisher and subscriber interfaces or methods. In this case, select `IEventObj` for the subscriber method(s) and `Publisher.EventObj` for the event class. Enter **`Subscription of Doom`** as the name of this subscription and choose to enable the server immediately, as shown in Figure 18.12.

FIGURE 18.12
Subscription wizard in the Component Services tool.

Figure 18.13 shows the complete COM+ application definition as shown in the Component Services administration tool.

Publishing of Events

The setup is now complete, so all that is left is to publish the event by creating an instance of the `EventObj` class and calling the `IEventObj.MyEvent` method. The simplest way to do this is in a simple test application, as shown in Listing 18.5.

LISTING 18.5 `TestU.pas`—Unit to Fire the Loosely Coupled Event

```
unit TestU;

interface
```

LISTING 18.5 Continued

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    FEvent: IEventObj;
  end;

var
  Form1: TForm1;

implementation

uses ComObj, ActiveX;

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  OleCheck(CoCreateInstance(CLASS_EventObj, nil, CLSCTX_ALL, IEventObj,
    FEvent));
  FEvent.MyEvent('This is a clever string');
end;

end.
```

Figure 18.14 shows the result of pushing the magic button. Note that the event subscriber is created automatically by COM+ and the event handler code is executed.

You might notice that COM+ takes a few moments to invoke the event the first time through. This is because of the fairly substantial amount of internal infrastructure that needs to be loaded in order to fire COM+ events. The bottom line here is that you shouldn't depend on events being fired back to subscribers in real time. They'll get there soon, but not instantly.

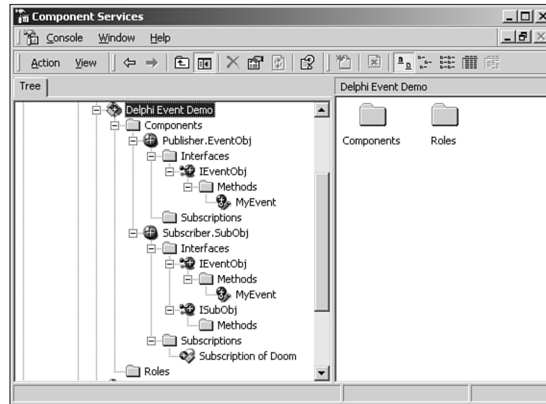

FIGURE 18.13
Event demo application in the Component Services tool.

FIGURE 18.14
Event demo application in action.

Beyond the Basics

Although this information provides a solid grounding in the fundamentals of the COM+ event model, there are a couple of powerful features that a we'd like to mention. The first is queued events. Queued events are the synthesis of COM+ events and queued components (MSMQ components in pre-COM+ days). Essentially, this functionality provides the capability to fire events to disconnected components, and those events can be played back at a later time. The other advanced topic worthy of mention is event filters, which come in two flavors: publisher filters and parameter filters. *Publisher* filters provide a means for publishers to control the order and firing of an event method by an event class. *Parameter* filters enable the publisher to intercept events based on the value of the parameters of that event.

Runtime

You can think of the COM+ runtime a as essentially the COM you already know and love. The COM+ runtime is comprised of all the various COM API functions (you know, all those functions that start with `Co...`) and the underlying code that makes those functions go. The runtime handles things like object creation and lifetime, marshaling, proxies, memory management,

and all the other low-level things that make up the foundation of COM+. In order to support many of the nifty services you just learned about, Microsoft has added a number of new features to the COM+ runtime, including configured components, a registration database, the promotion of the contexts concept, and a new neutral threading model.

Registration Database (RegDB)

In COM, the attributes of a particular COM object are generally kept in two places: the system registry and a type library. COM+ now introduces the concept of a registration database that will be used to hold attribute information for COM+ object. Type libraries will continue to be used, but the system registry has distinctly fallen out of favor as the place to store object attributes, and use of the registry for this purpose is supported only for the sake of backward compatibility. Common attributes stored in the RegDB include the transaction level supported by an object and whether it supports JIT activation.

Configured Components

Components that store attributes in RegDB are referred to as configured components, whereas components that don't are called non-configured. The best example of a non-configured component is a COM or MTS component that you are using unchanged in the COM+ environment. In order to participate in most of the services we mentioned earlier, your components will need to be configured.

Contexts

Contexts is a term originally introduced in MTS that described the state of the current execution environment of a given component. Not only has this term moved forward in COM+, but also it has been promoted. In COM, an apartment is the most granular description of the runtime context of a given object, referring to an execution context bounded by a thread or process. In COM+ that honor goes to a context, which runs within some particular apartment. A context implies a description on a more granular level than an apartment, such as transaction and activation state.

Neutral Threading

COM+ introduces a new threading model, known as *Thread Neutral Apartment (TNA)*. TNA is designed to provide the performance and scalability benefits of a free threaded object without the programming problems of dealing with interlocking access to shared data and resources within the server. TNA is the preferred threading model for COM+ components that don't surface UI elements. Components containing UI should continue to use apartment threading because window handles are tied to a specific thread. There is a limitation of one TNA per process.

Creating COM+ Applications

With all the knowledge of individual COM+ features under your belt, now is a good time to learn more about creating applications that leverage COM+ features such as transactions, lifetime management, and shared resources.

The Goal: Scale

The magic word of system design these days is scalability. With the hyper-growth of the Internet (intranets, extranet, and all other things net), 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 systems 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.

NOTE

The added infrastructure introduced in a multitier environment tends to increase latency as it increases bandwidth. In other words, you might very well need to sacrifice the performance of the system in order to improve scalability!

Execution Context

It's important to bear in mind that because COM+ object don't run directly within the context of a client like other COM objects, clients never really obtain interface pointers directly to an object instance. Instead, COM+ inserts a proxy between the client and the COM+ object such that the proxy is identical to the object from the client's point of view. However, because COM+ 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 COM+ 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 that they're created, while they're being used, and up until the time that 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 doesn't maintain state information between method calls. COM+ prefers stateless objects because they enable COM+ to play some optimization tricks. If an object doesn't maintain any state between method calls, COM+ could theoretically make the object go away between calls without causing any harm. Furthermore because the client maintains pointers only to COM+'s internal proxy for the object, COM+ could do so without the client being any the wiser. It's more than a theory; this is actually how COM+ works. COM+ 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 COM+ 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 isn't 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 COM+ would be to pass all the necessary information to the `AddActivity()` method so that the object could behave in a stateless manner:

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

The particular state of an active object is also referred to as a context. COM+ maintains a context for each active object that tracks things like 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

```
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 COM+ that it no longer has any state to maintain. COM+ will therefore destroy the object (unbeknown 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 effect of a commit or rollback for the transaction, respectively.

Lifetime Management

From the time we were tiny COM programmers, we were taught to hold on to interface pointers only for as long as necessary and to release them as soon as they are 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 COM+ 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 might have been deleted under the sheets, clients don't have to be rewritten to take advantage of this feature.

COM+ Application Organization

Remember that a collection of COM+ components that share common configuration and attributes are referred to in the Component Services tools as an *application*. Prior to COM+ MTS,

used the word *package* to refer to what we now call applications, but we are happy with the change in terminology—the term *package* was already overloaded enough, with Delphi packages, C++Builder packages, Oracle packages, and holiday gifts all coming to mind as examples of the overuse of this word.

By default, COM+ will run all components within a package in the same process. This enables you to configure well behaved and error-free packages that 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.

Applications can be created and manipulated using either the Run, Install COM+ Objects menu in Delphi or the Component Services tool.

Thinking About Transactions

And of course, COM+ 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?" A fair question, and luckily we're equipped with good answers. Transaction support in COM+ 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 COM+ objects, you must either set the correct transaction flag on your object's coclass in the type library during development (this is what the Delphi Transactional Object 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 in which a client bounces a check, there would likely be several steps involved in handling that, including

- debiting the account for amount of check
- debiting the account for bounced check service charge
- 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. All will roll back to their original pre-transaction state if an error occurs.

Resources

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

COM+ in Delphi

Now that you've got the "what" and "why" down, it's time to talk about the "how." In particular we intend to focus on Delphi's support of COM+ and how to build COM+ solutions in Delphi. Before we jump right in, however, you should first know that COM+ support is built only into the Client/Server version of Delphi. Although it's technically possible to create COM+ components using the facilities available in the Standard and Professional versions, we wouldn't consider it the most productive use of your time, so we intend to help you leverage the features of Delphi to build COM+ applications.

COM+ Wizards

Delphi provides two wizards for building COM+ components: the Transactional Data Module Wizard found on the Multitier tab of the New Items dialog box and the Transactional Object Wizard found on the ActiveX tab. The Transactional Data Module Wizard enables you to build MIDAS servers that operate in the COM+ environment. The Transactional Object Wizard will serve as the starting point for your COM+ transactional objects, and it is this wizard upon which I will focus my discussion. Upon invoking this wizard, you will be presented with the dialog box shown in Figure 18.15.

This dialog box is similar to the Automation Object Wizard with which you are probably already familiar based on your previous COM development experience in Delphi. The obvious difference is the facility provided by this wizard to select the transaction model supported by your COM+ 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, or it will otherwise 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, or it will execute without a transaction otherwise.
- Does Not Support Transactions—The component will never be created within a transaction.
- Ignores Transactions—The component doesn't care about the transaction context.

The transaction model information is stored along with the component's co-class in the type library.

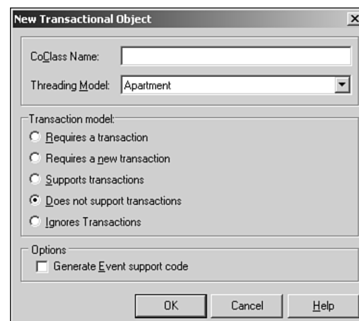

FIGURE 18.15
COM+ Transactional Object Wizard.

After you click OK to dismiss the dialog box, the wizard will generate an empty definition for a class that descends from `TMtsAutoObject` and it will present the Type Library Editor in order to define your COM+ 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 COM+ objects are automation objects (that is, COM objects that implement `IDispatch`), COM+ 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 COM+ enables you to concentrate more on your components' functionality and less on how they integrate with COM+. You should also be aware that COM+ components must reside in in-process COM servers (.DLLs); COM+ components aren't supported in out-of-process servers (.EXEs).

COM+ Framework

The aforementioned `TMtsAutoObject` class, which is the base class for all Delphi wizard-created COM+ 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 functionality to manage initialization, cleanup, and context.

`TMtsAutoObject` implements the `IObjectControl` interface, which manages initialization and cleanup of COM+ components. The methods of this interface are as follows:

`Activate()`—Allows an object to perform context-specific initialization when activated. This method will be called by COM+ prior to any custom methods on your COM+ component.

`Deactivate()`—Enables you to perform context-specific cleanup when an object is deactivated.

`CanBePooled()`—Was unused in MTS, but is supported in COM+, as described earlier in this chapter.

`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 COM+'s `IObjectContext` interface in the form of the `ObjectContext` property. 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 TMtsAutoObject's SetComplete() method simply checks FObjectContext for nil and then calls FObjectContext.SetComplete().

The following is a list of IObjectContext's methods and a brief explanation of each:

CreateInstance()—Creates an instance of another COM+ object. You can think of this method as performing the same task for COM+ objects as IClassFactory.CreateInstance() does for normal COM objects.

SetComplete()—Signals to COM+ 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, COM+ might deactivate the object, thereby freeing up resources for greater scalability.

SetAbort()—Similar to SetComplete(), this method signals to COM+ 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 COM+ 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 COM+ role. This method is the heart of COM+'s easy-to-use, role-based security system. We'll speak more on roles later.

The Mtx unit contains the core COM+ 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 COM+ API.

Tic-Tac-Toe: A Sample Application

That's enough theory. Now it's time to write some code and see how all this COM+ stuff performs on the open road. COM+ ships with a sample tic-tac-toe application that's a bit on the ugly side, so it inspired me to implement the classic game from the ground up in Delphi. To start, you use the Transactional Object Wizard to create a new object called `GameServer`. Using the Type Library Editor, add to the default interface for this object, `IGameServer`, three methods, `NewGame()`, `ComputerMove()`, and `PlayerMove()`. Additionally, add two new enums, `SkillLevels` and `GameResults`, that are used by these methods. Figure 18.16 shows all these items displayed in the Type Library Editor.

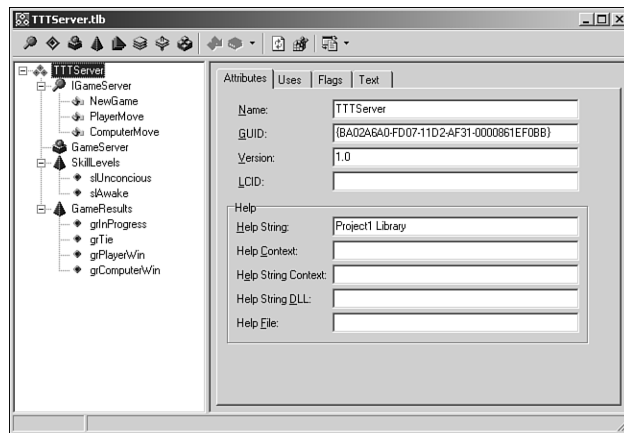

FIGURE 18.16
Tic-Tac-Toe server in the Type Library Editor.

The logic behind the three methods of this interface is simple, and they 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 has chosen to move. Earlier, we mentioned that COM+ component development requires a frame of mind different from the development of standard COM components. This component offers a nice opportunity to illustrate this fact.

If this were your average, everyday, run-of-the-mill COM component, you might approach the 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 a COM+ component? One word: state. As you learned earlier, object must be stateless in order to realize the full benefit of COM+. However, a component architecture that depends on instance data to be maintained across method calls is far from stateless. A better design for COM+ 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 COM+ might 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 and 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 is known as a *resource dispenser* in COM+. Specifically, the Shared Property Manager is the COM+ resource dispenser 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 can contain any number of shared properties, represented by the `ISharedProperty` interface. Shared properties are convenient because they exist within COM+, 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 listing:

```
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 that 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 COM+ knows it's okay to deactivate this instance after the method returns.

This leads us to the number one rule of COM+ development: call `SetComplete()` or `SetAbort()` as often as possible. Ideally, you will call `SetComplete()` or `SetAbort()` in every method so that COM+ can reclaim resources previously consumed by your component instance after the method returns. A corollary to this rule is that object activation and deactivation shouldn't 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 COM+:

```
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 raises 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 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 9x/Me, `IsCallerInRole()` will always return `True`.

The `ComputerMove()` and `PlayerMove()` methods are shown here:

```
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 if 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 COM+ component, take a look at Listing 18.6, which contains the entire source code for the `ServMain` unit.

LISTING 18.6 `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;
```

LISTING 18.6 Continued

```
    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');
  // 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;
```

LISTING 18.6 Continued

```
  // 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;
```

LISTING 18.6 Continued

```
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;

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
```

LISTING 18.6 Continued

```
    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;
```

LISTING 18.6 Continued

```
  // 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;
    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;
```

LISTING 18.6 Continued

```
// 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
```

LISTING 18.6 Continued

```
      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;
      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
```

LISTING 18.6 Continued

```
          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 COM+, Delphi makes your life very easy. Simple select Run, Install COM+ Objects from the main menu, and you will invoke the Install COM+ Objects dialog box. This dialog box enables you to install your object(s) into a new or existing package, and it is shown in Figure 18.17.

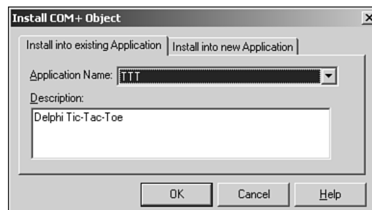

FIGURE 18.17
Installing a COM+ object via the Delphi IDE.

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 COM+ components via the Transaction Server Explorer application. Note that this installation procedure is markedly different from 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 make it similarly easy to set up COM+ 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 18.7 shows the source code for the client application for this COM+ component. Its purpose is to essentially map the engine provided by the COM+ component to a Tic-Tac-Toe–looking user interface.

LISTING 18.7 `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;

  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;
```

LISTING 18.7 Continued

```
    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
```

LISTING 18.7 Continued

```
  // 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
  with TFrmAbout.Create(Application) do
    try
      ShowModal;
    finally
      Free;
    end;
end;

procedure TFrmMain.TagToCoord(ATag: Integer; var Coords: TPoint);
begin
  case ATag of
```

LISTING 18.7 Continued

```
    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;
```

LISTING 18.7 Continued

```
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 = (
    '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;
```

LISTING 18.7 Continued

```
    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 18.18 shows this application in action. Human is X and computer is O.

FIGURE 18.18
Playing tic-tac-toe.

Debugging COM+ Applications

Because COM+ components run within COM+'s process space rather than the client's, you might think that they would be difficult to debug. However, COM+ provides a side door for debugging purposes that makes debugging a snap. Just load the server project, and use the Run Parameters dialog box 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 Component Services tool. This dialog box is shown in Figure 18.19. Next, set your desired breakpoints and run the application. You won't see anything happen initially because the client application isn't yet running. Now you can run the client from Windows Explorer or a command prompt, and you will be off and debugging.

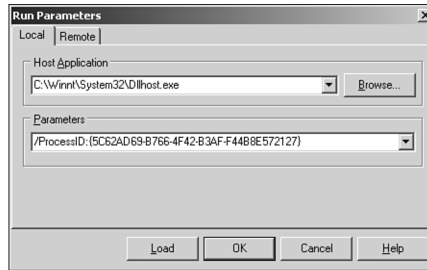

FIGURE 18.19
The Run Parameters dialog box.

Summary

COM+ 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 chapter took you through a tour of the basics of COM+ and on to the specifics of Delphi's support for COM+ and how to create COM+ applications in Delphi. What's more, you've hopefully caught a few tips and tricks along the way for developing optimized and well-behaved COM+ components. COM+ packs a wallop out of the box by providing services such as lifetime management, transaction support, security, all in a familiar framework. COM+ and Delphi combine to provide you with a great way to leverage your COM experience into creating scalable multi-tier applications. Just don't forget those differences in design nuances between normal COM components and COM+ components!