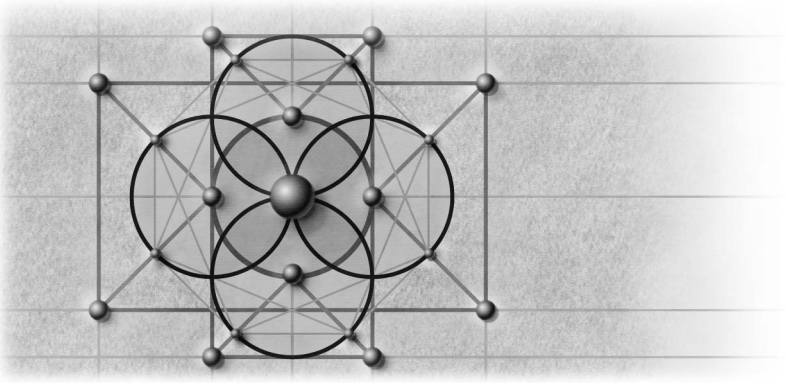

16

Upgrading COM+ Components

COM+ services and Microsoft Transaction Server components are the backbone of most major Microsoft Visual Basic business applications. This chapter starts with an introduction to implementing COM+ services in Microsoft .NET and then moves on to describe how to upgrade basic COM+ components. Although the emphasis is on transactional objects and object construction, the information here applies to the upgrading of all kinds of COM+ components. From this chapter you should take away a basic understanding of how COM+ components are implemented in Visual Basic .NET, and you should have an idea of the effort required to upgrade your existing components.

> **Note** This chapter is definitely not for someone who is not familiar with COM+. It makes no attempt to introduce the concepts behind the code. For that, a wealth of resources is available through Microsoft Press publications and the MSDN documentation, both online and included with Visual Basic .NET.

COM+ Application Types

To frame the discussion, let's start by taking a look at COM+ applications. According to the Microsoft Developer Network (MSDN), there are three basic types of user-creatable COM+ applications. Each has particular features,

advantages, and restrictions that make it appropriate for specific application architectures. The three types are as follows:

- **Server application** A COM+ application that runs in its own process. Server applications can support all COM+ services.
- **Library application** A COM+ application that runs in the process of the client that creates it. More specifically, the components in a library application are always loaded into the process of the creator. Library applications can use role-based security but do not support remote access or queued components.
- **Application proxy** A set of files containing registration information that allows a client to remotely access a server application. When run on a client computer, an application proxy file writes information about the COM+ server application—including its CLSID, ProgID, RemoteServerName, and marshaling information—to the client computer. The server application can then be accessed remotely from the client computer.

> **Note** Server applications differ from library applications in how they are instantiated. A library application's components are created in the same process as the calling application. A server application's components are created out of process. When you attempt to instantiate an object remotely, it is not possible to create that component in the caller's process; therefore, only server applications are capable of instantiation by remote applications, due to their out-of-process nature.

The vast majority of COM+ applications fall into the first two categories, and those will be the focus of the development content in this chapter. The third, application proxy, is a specialization of the server application and can be treated as a logical extension of the discussion contained here.

Using COM+ in Visual Basic .NET

Before getting into the how-to of upgrading, it's important to take a step back and investigate how COM+ services are implemented in .NET. For those of you who are curious as to what has happened to COM+ in .NET, fear not. COM+ is still the same sprightly beast that it was before you installed Visual Studio .NET. The

.NET Framework exposes all of the existing COM+ services to Visual Basic .NET applications. Table 16-1 lists the COM+ services available to .NET developers.

Table 16-1 COM+ Services Supported in Visual Basic .NET

Service	Description
Automatic transaction processing	Applies declarative transaction processing features.
BYOT (bring your own transaction)	Allows a form of transaction inheritance.
COM Transaction Integrator (COMTI)	Encapsulates Customer Information Control System (CICS) and Internet Mail Service (IMS) applications in Automation objects.
Compensating Resource Manager	Applies atomicity and durability properties to nontransactional resources.
Just-in-time activation	Activates an object on a method call and deactivates it when the call returns.
Loosely coupled events	Manages object-based events.
Object construction	Passes a persistent string value to a class instance on construction of the instance.
Object pooling	Provides a pool of ready-made objects.
Private components	Protects components from out-of-process calls.
Queued components	Provides asynchronous message queuing.
Role-based security	Applies security permission based on role.
SOAP services	Publishes components as XML Web services.
Synchronization	Manages concurrency.
XA interoperability	Supports the X/Open transaction processing model.

Despite implementation differences, underneath the .NET Framework the stalwart heart of COM+ still beats. The .NET Framework COM+ services are built on top of COM+, rather than being a substitute for it. These are the same services that you are already familiar with; they are just implemented differently.

The fact that COM+ is a set of native application services (it does not run on the common language runtime with Visual Basic .NET) raises interesting questions: Is this COM interop, and is there a performance penalty? The short and simple answer is no. COM+ services are able to call directly into the runtime environment and vice versa without COM interop wrappers or variable

marshaling overhead. The technical details as to why are too involved to go into here (this is not to say that the authors don't need to be reminded of them periodically), but rest assured—you will not pay a performance penalty for choosing to use COM+ services through the .NET Framework instead of through Visual Basic 6. In fact, thanks to the new power of Visual Basic .NET, more COM+ services are available to you than ever before, and you have more leverage to use them. In addition, Visual Basic .NET makes implementing COM+ applications far easier than in Visual Basic 6 and gives the developer a significant amount of control over component deployment issues.

> **Note** At this point, you might be wondering why COM interop doesn't come into play with COM+. Isn't COM+ the same as COM? Yes and no. COM+ was designed as an application infrastructure for developing COM components for the enterprise. While there is a high level of integration between the two technologies, COM+ is not really COM. This difference enables COM+ to call directly into the runtime (and vice versa) without the overhead and marshaling associated with a COM interop method call.

What about the differences between COM+ in Visual Basic 6 and COM+ in .NET? While they do exist, COM+ is COM+ no matter how you look at it. On the other hand, an ActiveX DLL is by no means a managed assembly (a Visual Basic .NET DLL), and this has specific benefits (see the discussion of attributes later in this chapter). Visual Basic .NET is designed to provide more integrated support for COM+, above and beyond that found in Visual Basic 6. To help illustrate the differences, let's take a look at how COM+ applications are built in Visual Basic .NET.

COM+ Requirements in Visual Basic .NET

By now you should be familiar with the namespace structure, if not the content, of the .NET Framework. Support for COM+ services is implemented in the System.EnterpriseServices namespace. Any component wishing to take advantage of COM+ (aside from adding a reference to the System.EnterpriseServices.dll assembly) must meet all of the following requirements:

- It must inherit from the *ServicedComponent* class or from another class derived from *ServicedComponent*.
- It must apply attributes specifying the COM+ services supported by the class.

- The assembly containing the COM+ component must have a strong name.

The sections that follow discuss each of these requirements.

Inheriting from the *ServicedComponent* Class

We have mentioned that there are differences between the implementation of COM+ in Visual Basic 6 and its implementation in Visual Basic .NET. One of these differences is that in Visual Basic 6 any class can implement a COM+ service by implementing specific interfaces from the COM+ services type library. The downside to this approach is that you have to implement a great deal of boilerplate code yourself. In .NET, the situation is quite different. All components that want to leverage COM+ services need to inherit their objects from the *ServicedComponent* class. This class provides a framework for your custom component that enables context sharing between COM+ and .NET Framework classes. In addition, it provides a host of features that enable self-registration and support for new .NET features like remoting and XML Web services.

This change means that your COM+ classes take on the following structure. (Note that this is by no means a complete implementation.)

```
Imports System.EnterpriseServices

' A trivial COM+ Class
Public Class COMPlusClass
   Inherits ServicedComponent ' Required to support COM+

   Public Sub New()
      MyBase.New()
   End Sub
End Class
```

Instead of implementing external interfaces to handle events such as *Construct*, *Activate*, and *Deactivate*, you use member functions of the *ServicedComponent* class. To implement specific features that you need, you override the base class functionality. This makes your class look like the following:

```
Imports System.EnterpriseServices

' A trivial COM+ Class
Public Class COMPlusClass
   Inherits ServicedComponent ' Required to support COM+

   Public Sub New()
      MyBase.New()
   End Sub
```

(continued)

```
    Protected Overrides Sub Construct(ByVal ConstructionString _
       As String)
    End Sub

    Protected Overrides Sub Activate()
    End Sub

    Protected Overrides Sub Deactivate()
    End Sub
End Class
```

Starting to look familiar yet? It's still not even close to being complete, however. We have created the base framework for our COM+ class, but we need to provide more information about what we intend it to do and what services it supports. That is where attributes come into play.

Working with Attributes

Attributes are a new feature in Visual Basic .NET. Attributes are specialized classes that are applied to code elements, allowing you to provide descriptive metadata about your component. At compile time, attributes are emitted into metadata that is available to the common language runtime or to custom tools and applications through the System.Reflection namespace. You can explicitly mark objects and methods with attributes that affect their behavior.

You attach an attribute to a component by preceding the component with a reference to the attribute and providing any relevant parameters or flags. This call to the constructor is placed within angle brackets (<>) in Visual Basic. Whenever you create a new application in Visual Basic .NET, a file called AssemblyInfo.vb is created. This file contains a set of common attributes for your project. The file typically looks like this:

```
Imports System.Reflection
Imports System.Runtime.InteropServices

' General Information about an assembly is controlled through
' the following set of attributes. Change these attribute values to
' modify the information  associated with an assembly.

' Review the values of the assembly attributes

<Assembly: AssemblyTitle("")>
<Assembly: AssemblyDescription("")>
<Assembly: AssemblyCompany("")>
<Assembly: AssemblyProduct("")>
<Assembly: AssemblyCopyright("")>
```

```
<Assembly: AssemblyTrademark("")>
<Assembly: CLSCompliant(True)>

' The following GUID is for the ID of the typelib if this project
' is exposed to COM

<Assembly: Guid("17465E40-15E9-4F52-8954-CF931AC54747")>

' Version information for an assembly consists of the following
' four values:

'      Major Version
'      Minor Version
'      Build Number
'      Revision

' You can specify all the values or you can default the Build and
' Revision Numbers by using the '*' as shown below:

<Assembly: AssemblyVersion("1.0.*")>
```

In addition to this standard set of attributes, the System.EnterpriseServices namespace defines a host of attributes specific to COM+. These attributes provide a combination of services. Some specify feature support; others specify your application's default COM+ settings. Table 16-2 contains a list of COM+ specific attributes.

Table 16-2 COM+ Specific Attributes

Attribute	Scope	Unconfigured Default Value	Configured Default Value
ApplicationAccessControl	Assembly	False	True
ApplicationActivation	Assembly	Library	No default
ApplicationID	Assembly	Generated GUID	No default
ApplicationName	Assembly	Assembly name	No default
ApplicationQueuing	Assembly	No default	No default
AutoComplete	Method	False	True
ComponentAccessControl	Class	False	True
COMTIIntrinsics	Class	False	True
ConstructionEnabled	Class	False	True
Description	Assembly Class Method Interface	No default	No default

Table 16-2 COM+ Specific Attributes *(continued)*

Attribute	Scope	Unconfigured Default Value	Configured Default Value
EventClass	Class	No default	*FireInParallel* = *False* *AllowInprocSubscribers* = *True* *PublisherFilter* = Null
EventTrackingEnabled	Class	False	True
ExceptionClass	Class	No default	No default
InterfaceQueuing	Class Interface	False	True
JustInTimeActivation	Class	False	True
LoadBalancingSupported	Class	False	True
MustRunInClientContext	Class	False	True
ObjectPooling	Class	False	True
PrivateComponent	Class	No default	Private
SecureMethod	Assembly Class Method	No default	No default
SecurityRole	Assembly Class Interface	No default	No default
Synchronization	Class	False	Synchronization-Option.Required
Transaction	Class	False	*Transaction-Option.Required* *TransactionIsolation-Level.Serializable* *Timeout* = *infinite*

Let's see how our sample COM+ class looks with some of these important attributes:

```
Imports System.EnterpriseServices

<Assembly: ApplicationName("Sample COMPlus Application")>
<Assembly: ApplicationActivation(ActivationOption.Library)>

' A trivial COM+ Class with attributes
```

```
<Transaction(TransactionOption.Required), _
   ConstructionEnabled([default]:=" This is a test"), _
   ObjectPooling()> _

Public Class COMPlusClass

   Inherits ServicedComponent ' Required to support COM+

   Public Sub New()
      MyBase.New()

   End Sub

   Protected Overrides Sub Construct(ByVal ConstructionString _
      As String)
      Trace.WriteLine("Construct() : """ & ConstructionString & """")
   End Sub

   Protected Overrides Sub Activate()
      Trace.WriteLine("Activate()")
   End Sub

   Protected Overrides Sub Deactivate()
      Trace.WriteLine("Activate()")
   End Sub

   <AutoComplete()> Public Sub DoTasks()

   End Sub
End Class
```

This example demonstrates the use of all three scopes (method, class, and assembly) of attributes and shows how you can specify support for various COM+ features. Some of these attributes are fairly obvious and are not too big a leap from Visual Basic 6 (the *Transaction* attribute, for example). Others highlight the ability of attributes to better control deployment. An example of the latter is the *ConstructionEnabled* attribute. This particular attribute allows the developer to specify a default construction string that will be visible (and modifiable) in the Component Services Microsoft Management Console (MMC).

While other COM+ attributes provide only default settings for use during self-registration, the *AutoComplete* attribute provides a fundamentally new twist on programming with transactions. Marking a method with *AutoComplete* causes *SetCommit* to be called automatically if the method terminates normally. This saves the developer a fair bit of work. If an unexpected error occurs (in the form of an unhandled exception), *SetAbort* is called automatically. You can still call *SetAbort* on your own based on a logical violation, but it is no longer necessary for you to explicitly commit or abort the transaction in every circumstance. This attribute reduces a great deal of repetition by requiring code only for the extraordinary cases. The following example shows how simple implementing transactions now becomes with the *AutoComplete* attribute:

```
Imports System.EnterpriseServices
<Transaction(TransactionOption.Required)>
Public Class Sample COMPlusClass

   Inherits ServicedComponent

   <AutoComplete()>Public Function DoTransactionTask() As Boolean

     ' Do Your stuff.
     ' Calls set complete automatically if no exception is generated

   End Function
End Class
```

All COM+ settings can be set using attributes on a class or method, instead of using the property pages found in Visual Basic 6. This approach makes settings more explicit and less prone to accidental change.

We've only been able to scratch the surface of the topic of attributes here, but hopefully this discussion has given you a decent idea of how to use them. Attributes are used extensively throughout Visual Basic .NET to implement all sorts of functionality—not only COM+ features but also newer features like XML Web services.

Creating a Strong Name for Your Assembly

If you are registering a component as part of a COM+ application, you must provide a strong name for the assembly. A **strong name** consists of an assembly's identity—its simple text name, version number, and culture information (if provided)—strengthened by a public key and a digital signature generated over the assembly. Assemblies with the same strong name are expected to be identical.

Generating Key-Pair Files

To sign an assembly with a strong name, you must have a public/private key pair. This public and private cryptographic key pair is used during compilation to create a strong-named assembly. You can create a key pair using the Strong Name tool (Sn.exe). Key-pair files usually have an .snk extension.

> **Note** The Sn.exe tool is provided with the .NET Framework SDK and Visual Basic .NET, but it is not normally available from the standard command-line prompt. To use the tool, on the Programs menu, point to Microsoft Visual Studio .NET and then to Visual Studio .NET Tools, and choose Visual Studio .NET Command Prompt. This command-line tool has all of the necessary environment variables set to run all of the Framework SDK and Visual Studio .NET tools.

To create a key pair, type the following at the command prompt:

sn –k <*filename*>

In this command, filename is the name of the output file containing the key pair. The following example creates a key pair called COMPlusClass.snk.

sn –k COMPlusClass.snk

Once you create the key pair, you must put the file where Visual Basic can find it. When signing an assembly with a strong name, Visual Basic looks for the key file in the directory containing the Visual Studio solution. Put the key file in the appropriate project directory, and add the following assembly attribute near the top of the COMPlusClass.vb file:

```
<Assembly: AssemblyKeyFile("COMPlusClass.snk")>
```

> **Note** Although we chose to name the key file after the class it was created for, the key file itself has nothing specifically to do with the class. The key file is applied at the assembly level to ensure that the application has a unique signature. That is, after all, why the strong name feature exists. It allows for the definition of unique applications regardless of any possible naming conflicts. This uniqueness becomes very important when dealing with applications that have machine scope.

Registering COM+ Applications

Now that you know all of the requirements for COM+ components, it's time to talk about how to register the components with COM+. There are two ways to register .NET COM+ applications. The first is easier: dynamic registration. This method requires the least amount of work on your part and will be done for you as long as you specify the appropriate attributes in your assembly manifest. The second option is manual registration. It requires you to do a little more work but is not substantially more difficult than dynamic registration.

Dynamic Registration

Dynamic registration is a process whereby COM+ components are self-registered. Other than specifying a set of assembly attributes, you need only make sure that your COM+ application (whether it is in a separate assembly or not) resides in your application's bin directory. Fusion will find the assembly and automatically register the COM+ application, using all of the attributes you have specified.

What Is Fusion?

Fusion is the mechanism that enables the common language runtime to handle run-time assembly binding. It is responsible for finding an appropriate assembly that contains the objects required by your application. This is the essential tool that enables side-by-side application deployment and eliminates the need for registering assemblies and type libraries. While fusion is a complex concept, all you really need to know is that it is responsible for binding your application's assemblies at run time and makes it possible to eliminate DLL conflicts from Visual Basic .NET.

As an example, we have provided the COMPlusClass project on the companion CD. This project contains one form and one COM+ component. Running the project brings up the main form, shown in Figure 16-1.

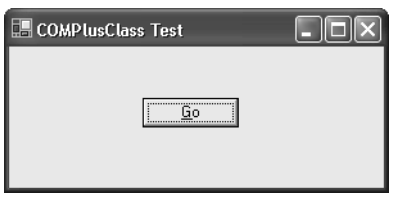

Figure 16-1 COMPlusClass sample application.

Clicking the Go button causes the COMPlusClass to be instantiated. When this happens, the component is automatically registered in COM+. You can see this in the Component Services MMC, which should look something like Figure 16-2.

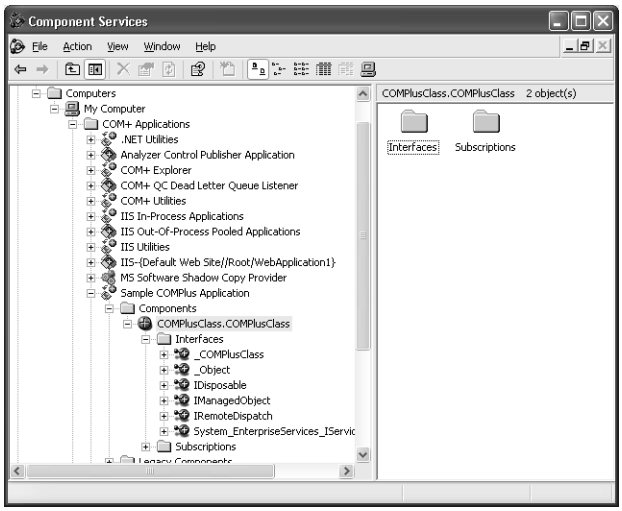

Figure 16-2 Installed COMPlusClass application in COM+.

It can't get much easier than that, can it? If you check the properties of the COM+ application in the MMC, you will see the effects of the attributes you specified in the application itself. Replacing the application defined in COM+ is as simple as deleting it from the Component Services console and running the application again.

Manual Registration

Manually registering an assembly can be done for both library and server applications. You register an application manually by running the RegSvcs.exe utility on your application's assembly. Here is the general syntax of this command:

regsvcs [*assemblyname*]

This utility does the following:

1. Loads the assembly.
2. Registers the assembly.
3. Generates a type library.
4. Registers the type library.

5. Installs the type library into the request application.
6. Configures the class.

You can use the RegSvcs.exe utility to customize the registration process and also as a general maintenance utility that you use to remove COM+ applications.

Upgrading COM+ Components

Unfortunately, because a Visual Basic .NET COM+ services application has to be implemented in quite a different manner than a Visual Basic 6 COM+ services application, the Upgrade Wizard cannot do much for your Visual Basic 6 COM+ components. It upgrades the business logic, database code, and other parts of your application, but it doesn't upgrade the transaction attributes. The implementation differences are not a one-to-one mapping, so upgrading the parts dealing with COM+ support is best left to the developer. This leaves you with a fair bit of work to do on your own. To better illustrate what is going on here, let's look at a Visual Basic 6 sample called SimpleTransaction, included on the companion CD. This is an ActiveX DLL project with a single class, *SimpleClass*, that is marked as *RequiresNewTransaction*. The SimpleClass.cls code looks like this:

```
Implements ObjectControl
Implements IObjectConstruct

Dim ctxt As ObjectContext
Dim connStr As String
Const m_module As String = "SimpleTransaction.SimpleClass"

Private Sub IObjectConstruct_Construct(ByVal pCtorObj As Object)
   connStr = pCtorObj.ConstructString
End Sub

Private Sub ObjectControl_Activate()
   Set ctxt = GetObjectContext(m_module)
End Sub

Private Function ObjectControl_CanBePooled() As Boolean
   ObjectControl_CanBePooled = True
End Function

Private Sub ObjectControl_Deactivate()
End Sub
```

```
Public Sub DoTasks()
   On Error GoTo HandleError
   Dim conn As Connection
   Set conn = New Connection

   Dim sqlCmd As String
   sqlCmd = "This is a sample query"

   conn.Open connStr
   conn.Execute sqlCmd

   ctxt.SetComplete
   Exit Sub

HandleError:
   ctxt.SetAbort
End Sub
```

After running this sample project through the Upgrade Wizard, you have a Visual Basic .NET class that looks like this:

```
Option Strict Off
Option Explicit On
Public Class SimpleClass
   Implements COMSVCSLib.ObjectControl
   Implements COMSVCSLib.IObjectConstruct

   Dim ctxt As COMSVCSLib.ObjectContext
   Dim connStr As String

   Const m_module As String = "SimpleTransaction.SimpleClass"

   Private Sub IObjectConstruct_Construct(ByVal pCtorObj As Object) _
     Implements COMSVCSLib.IObjectConstruct.Construct
     ' UPGRADE_WARNING: Couldn't resolve default property of object
     ' pCtorObj.ConstructString...
     connStr = pCtorObj.ConstructString
   End Sub

   Private Sub ObjectControl_Activate() Implements _
     COMSVCSLib.ObjectControl.Activate
     ctxt = COMSVCSLibAppServer_definst.GetObjectContext(m_module)
   End Sub

   Private Function ObjectControl_CanBePooled() As Boolean _
     Implements COMSVCSLib.ObjectControl.CanBePooled
     ObjectControl_CanBePooled = True
   End Function
```

(continued)

```
   Private Sub ObjectControl_Deactivate() _
         Implements COMSVCSLib.ObjectControl.Deactivate
   End Sub

   Public Sub DoTasks()
      On Error GoTo HandleError
      Dim conn As ADODB.Connection
      conn = New ADODB.Connection

      Dim sqlCmd As String
      sqlCmd = "This is a sample query"

      conn.Open(connStr)
      conn.Execute(sqlCmd)

      ctxt.SetComplete()
   Exit Sub

HandleError:
     ctxt.SetAbort()
   End Sub
End Class
```

As you can see, the Upgrade Wizard leaves the work of implementing the COM+ parts of the class to you. While this task is quickly accomplished for this simple example, you should be aware that bigger applications will take time if the classes have a large set of transactional components. You will need to refer to the Visual Basic 6 version of each class to ensure that you transfer the correct transactional attributes to Visual Basic .NET. Otherwise, you may see unexpected and potentially undesirable behavior.

Using the previous example, let's walk through the steps necessary to get the class working:

1. Add a reference to System.EnterpriseServices.
2. Change the class to inherit from *EnterpriseServices.ServicedComponent*, and remove all COMSVCS *Implements* statements.
3. Add the necessary COM+ attributes to the class (*Transaction*, *ConstrutionEnabled*, *ObjectPooling*, and so on).
4. Change the COM+ interface methods *Activate*, *Construct*, and *Deactivate* to be member methods that override base methods on *ServicedComponent*.
5. Add the *AutoComplete* attribute to the *DoTasks* method. Remove the explicit commit and abort code.

After you've completed all of these steps, the class looks like this:

```
Option Strict Off
Option Explicit On

Imports System
Imports System.EnterpriseServices

<Transaction(TransactionOption.RequiresNew), ConstructionEnabled(),
ObjectPooling()> Class SimpleClass
   Inherits ServicedComponent

   Dim connStr As String
   Const m_module As String = "SimpleTransaction.SimpleClass"

   Protected Overrides Sub Construct(ByVal constructionString _
      As String)
      connStr = constructionString
   End Sub

   <AutoComplete()> Public Sub DoTasks(ByVal fail As Boolean)
      Dim conn As ADODB.Connection
      conn = New ADODB.Connection()

      Dim sqlCmd As String

      sqlCmd = "This is a sample query"

      conn.Open(connStr)
      conn.Execute(sqlCmd)

       If fail Then ContextUtil.SetAbort()
   End Sub
End Class
```

Notice that the *DoTasks* method does not explicitly commit or abort the transaction except when a logical condition is true. This helps to demonstrate that you can still explicitly commit or abort your transactions. If you encounter a method in which the commit and abort logic is too tangled to reasonably remove, you can forgo using the *AutoComplete* attribute altogether and commit your transactions manually.

Making .NET and COM Components Work Together

Previous chapters have discussed issues regarding COM interop and how best to handle it in your applications. Although these discussions were directed at standard COM components, much of the information also applies to using COM+ applications from .NET. It is possible to import server, library, and proxy applications into your Visual Basic .NET projects. The imported objects are used just as they are in any other COM interop scenario. In addition, if your application has a *ServicedComponent* class that calls into the imported COM+ application, the context will propagate transparently across the interop boundary. You need to take into account some special considerations if you intend for your .NET serviced components to work with COM clients:

- Avoid using parameterized constructors.
- Avoid using static methods.
- Define event-source interfaces in managed code.
- Include HRESULTs in user-defined exceptions.
- Supply GUIDs for types that require them.
- Expect inheritance differences.

Conclusion

Upgrading your COM+ application to Visual Basic .NET is not a totally automatic process. The Upgrade Wizard does the hard work, upgrading your application structure and business logic, but legwork on your part is necessary to add the *ServicedComponent Inherits* statement, the COM+ attributes, and calls to the *SetComplete* and *SetAbort* methods.

We've covered how to upgrade COM+ (and Microsoft Transaction Server) transactional components; however, there is a lot more to COM+ services than simply transactions. So in a way, we are just scratching the surface of what is possible. To learn more about using COM+ services, see the topics "Serviced Component Overview," "Writing Serviced Components," and "Summary of Services" in the Visual Basic .NET Help.