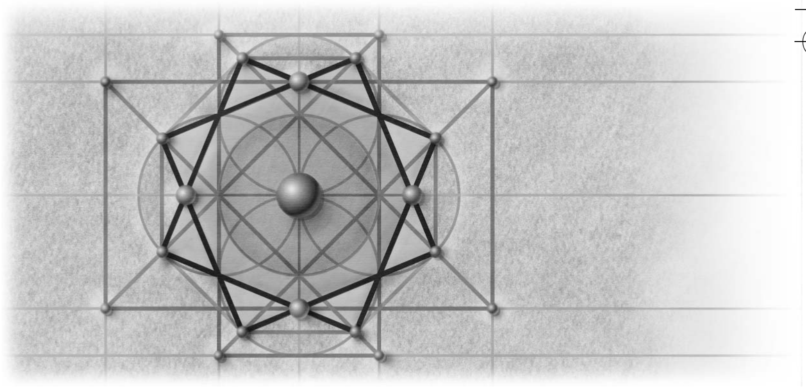


9 Using Visual Basic 6 with Visual Basic .NET: COM Interop

This chapter focuses on making your Visual Basic 6 and Visual Basic .NET applications work together. The mechanism that makes interoperability between the two products possible is known as COM interop. We'll start by looking at the various ways you can create or use existing components that communicate across COM and .NET component boundaries. We'll also show you how to debug across calls between Visual Basic 6 and Visual Basic .NET authored components. Finally, we'll discuss the role of binary compatibility in Visual Basic .NET.

If you've been creating multitiered applications using Visual Basic 6, your application has likely evolved into a large system spanning multiple components. Let's say, for example, that you have an application composed of a Visual Basic standard EXE front end containing ActiveX controls talking to a middle-tier Visual Basic DLL. The Visual Basic DLL in turn talks to a back-end SQL Server database. Upgrading such an application to Visual Basic .NET in one shot is nearly impossible. This is where COM interop swoops in to save the day.

COM interop allows you to upgrade one component at a time while keeping the system alive. For example, you can upgrade your Visual Basic 6 middle-tier component to Visual Basic .NET independently of the user interface (UI) component. Once you have tested your new Visual Basic .NET component with your Visual Basic 6 UI client, you can update the client to take advantage of the new Visual Basic .NET server component. At a later date you may decide to upgrade your Visual Basic 6 client components to Visual Basic .NET components. An ActiveX control vendor may offer a .NET upgrade to your favorite ActiveX control, leading you to replace all ActiveX versions of the control in your application with the .NET version. Eventually your entire system evolves to .NET, smoothly and without interruption.

Visual Studio .NET Is Built on COM Interop

You do not need to look far for an example of COM interop at work. If you're running Visual Studio .NET, COM interop is right under your nose. The Property Browser is written in C#, a language built on the .NET Framework. Most of the designers you will find, such as the Windows Forms designer, are written in a language supported by .NET. All of the wizards are written in either C# or Visual Basic .NET. The Visual Studio .NET environment is a traditional client application written in C++ that interoperates with these other .NET components using COM interop. The Upgrade Wizard relies heavily on COM interop to accomplish its tasks. The wizard is a .NET component that calls out to the upgrade engine, an out-of-process COM EXE server, to upgrade your Visual Basic 6 project. The upgrade engine in turn calls back to the wizard to provide status. As your application is being upgraded, the status text and progress bar updates you see are brought to you by way of COM interop.

We look forward to the day when 100 percent of our Visual Studio .NET components are written in Visual Basic .NET. Until that day, COM interop will be silently at work keeping Visual Studio .NET humming along.

It would be nice to be able to upgrade your entire application to .NET, but in some cases it may not be feasible. For example, what if your application relies on a COM component for which there is no .NET equivalent? A good example is a Visual Basic 6 ActiveX document or DHTML page designer for which there is no equivalent component in .NET. In such cases COM interop can help keep things running without hindering you from moving other parts of your system forward to Visual Basic .NET.

C0961587x.fm Page 177 Thursday, November 15, 2001 3:33 PM

Although we've been talking about interoperation among Visual Basic components, the concept of interoperation applies to all COM components. For example, your application may be composed of a Visual Basic front end talking to a C++ authored middle-tier component. As long as the components that make up your application are based on COM, your Visual Basic .NET application can continue to talk to them. Similarly, your Visual Basic 6 application can continue to talk to a .NET component—authored in any language supported by .NET—as if it were a COM component, without any changes in your Visual Basic 6 application.

Where COM Interop Comes into Play

There are three common situations in which you will encounter COM interop: when using ActiveX controls, when calling a COM component from a .NET application or component, and when calling a .NET component from a COM application or component. Let's take a look at each of these situations.

ActiveX Controls

Visual Basic .NET allows you to place either .NET or ActiveX controls on a Windows form. Chapter 13covers ActiveX control hosting in more detail. Suffice it to say that the way you use ActiveX controls in Visual Basic .NET is nearly identical to the way you use them in Visual Basic 6. You add the control to the Toolbox, place the control on the form, set design-time property values, and then write code behind it. No sweat.

Communication Between a .NET Client and a COM Server Component

The .NET Framework enables .NET clients to communicate with COM components. It accomplishes this by wrapping the COM component in a runtime callable wrapper (RCW) to make it look like a .NET component, as illustrated in Figure 9-1. Exposing a COM component for use with .NET requires no additional work. You simply add the COM component as a reference, and Visual Studio .NET takes care of generating the RCW for you. Later in this chapter, we demonstrate how to call a COM server from a .NET Web application.

C0961587x.fm Page 178 Thursday, November 15, 2001 3:33 PM

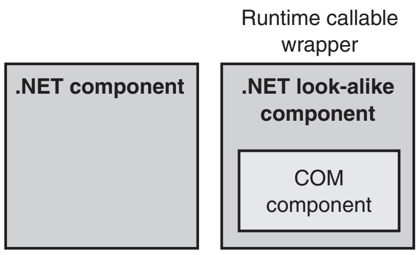


Figure 9-1 Runtime callable wrapper that enables a COM component to look like a .NET component.

Communication Between a COM Client and a .NET Server Component

In addition to allowing .NET clients to communicate with COM components, the .NET Framework allows COM applications to communicate with .NET components by making a .NET component look like a COM component. It accomplishes this by creating a wrapper around the .NET component called a COM callable wrapper (CCW), as illustrated in Figure 9-2. To expose a .NET component to COM, you must register your .NET component as being COM callable. Visual Studio .NET provides a way for .NET components to be registered automatically for COM. The next section demonstrates how to upgrade a Visual Basic 6 component to .NET and then expose that .NET component to COM.

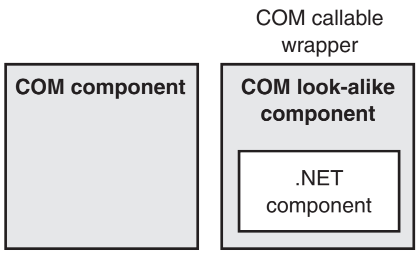


Figure 9-2 COM callable wrapper that enables a .NET component to look like a COM component.

Upgrading a Visual Basic 6 Client/Server Application

Suppose you have a Visual Basic 6 ActiveX DLL that acts as a language translator, translating a phrase in one language to another language. Your component is currently limited to rudimentary Spanish. The companion CD includes two Visual Basic 6 sample applications: TranslatorServer and TranslatorClient. In the following section we will demonstrate how to call the TranslatorServer component from a Visual Basic .NET client application. First, however, you need to build the Visual Basic 6 server component by following these steps:

1. Run Visual Basic 6.
2. Open TranslatorServer.Vbp provided on the companion CD.
3. Open TranslatorServer.Cls, and you will find that it contains the following code:

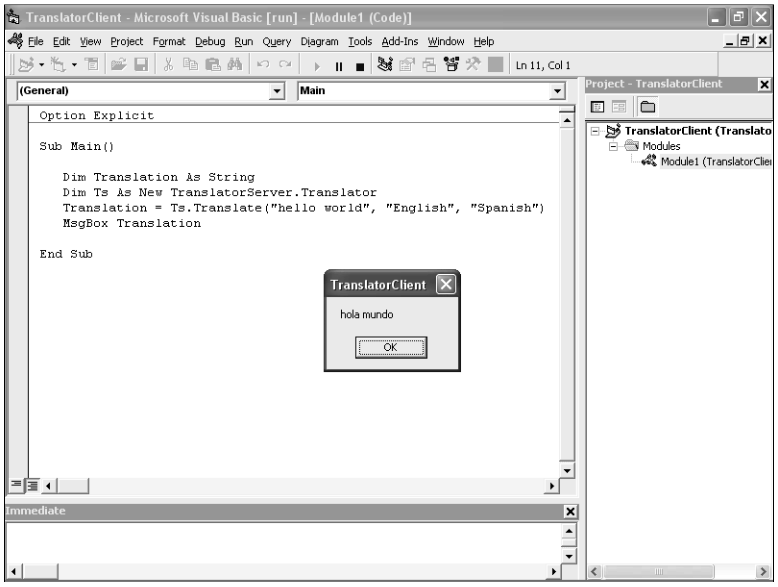
```
Public Function Translate(ByVal SentenceFrom As String, _
   ByVal LanguageFrom As String, _
   ByVal LanguageTo As String) As String

   ' Note to self: Find someone who speaks Spanish and VB who
   ' is willing to expand this component to translate any
   ' common English phrase
   If LCase(LanguageFrom) = "english" And _
      LCase(LanguageTo) = "spanish" Then
      Select Case LCase(SentenceFrom)
         Case "hello world"
            Translate = "hola mundo"
      End Select
   End If

End Function
```

4. From the File menu, choose Make TranslatorServer.dll and make the .DLL file into a directory on your hard drive.

To test your application in Visual Basic 6, run Visual Basic 6 and open the Visual Basic 6 TranslatorClient application provided on the companion CD. Open Module1.Bas and you'll find the following code to call the TranslatorServer component.

```
Sub Main()

   Dim Translation As String
   Dim Ts As New TranslatorServer.Translator
   Translation = Ts.Translate("hello world", "English", "Spanish")
   MsgBox Translation

End Sub
```

Now select References from the Project menu, click the Browse button, and locate the TranslateServer.Dll that you built by following the previous steps. Press F5 to run the project. You should see "hola mundo" displayed, as shown in Figure 9-3.


TranslatorClient - Microsoft Visual Basic [run] - [Module1 (Code)]
File Edit View Project Format Debug Run Query Diagram Tools Add-Ins Window Help
Ln 11, Col 1
(General)
Main
Project - TranslatorClient
TranslatorClient (Translato
Modules
Module1 (TranslatorClie
TranslatorClient
hola mundo
OK
Immediate

```
Option Explicit

Sub Main()

   Dim Translation As String
   Dim Ts As New TranslatorServer.Translator
   Translation = Ts.Translate("hello world", "English", "Spanish")
   MsgBox Translation

End Sub
```

Figure 9-3 “Hola mundo” translation successfully received from Visual Basic 6 server.

You may be wondering what the point of all this is. So far, all we’ve done is call a Visual Basic 6 component from Visual Basic 6. Now, however, we’ll upgrade each component separately to see how to call a Visual Basic 6 component from Visual Basic .NET and vice versa. Finally, we’ll see how to tie the upgraded client and server components together to form a complete Visual Basic .NET solution.

Creating a .NET Client That Talks to a COM Server

Let’s create a .NET client that talks to a COM server. Since the whole point of Visual Studio .NET is to help you create applications quickly for the Web, let’s create a Web client that calls our COM server. You might want to do this, for example, if you have business logic stored internally—say, a Visual Basic 6 ActiveX DLL function that returns a list of your company’s products—that you want to make available for viewing by external partners or customers.

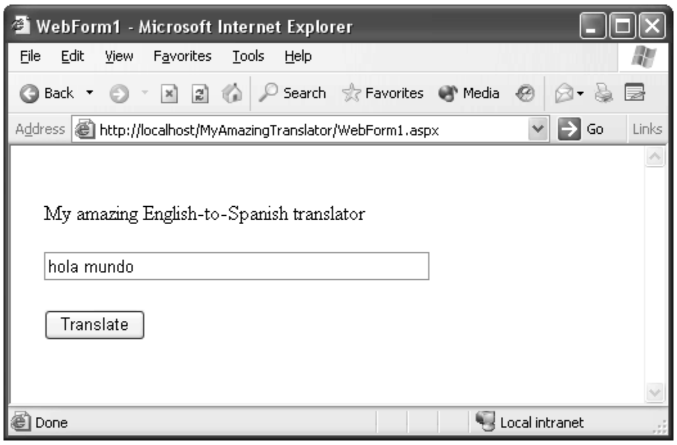
1. Run Microsoft Visual Studio .NET.
2. Choose New Project from the File menu.
3. Select Visual Basic ASP.NET Web Application, name the application MyAmazingTranslator, and click OK.

4. Right-click References on the Solution Explorer tab, and choose Add Reference.
5. Select the COM tab.
6. Select TranslatorServer from the list, and click OK.
7. Open WebForm1.aspx.
8. Drag and drop the following controls from the Toolbox to WebForm1: A Label, a TextBox, and a Button.
9. For the text of the label, enter **My amazing English-to-Spanish translator**.
10. For the text of Button1, enter **Translate**.
11. Double-click the Translate button and insert the following code in the *Button1_Click* event handler:

```
Dim ts As New TranslatorServer.Translator()
TextBox1.Text = ts.Translate(TextBox1.Text, "English", _
   "Spanish")
```

12. Choose Start from the Debug menu.
13. Type **hello world** into the text box, and click the button. See Figure 9-4 for an example of the output.

You have now taken a simple desktop application and made it available to the world. Hola mundo indeed.


WebForm1 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/MyAmazingTranslator/WebForm1.aspx
Go Links

My amazing English-to-Spanish translator

hola mundo

Translate

Figure 9-4 My amazing English-to-Spanish translator at work on the Web.

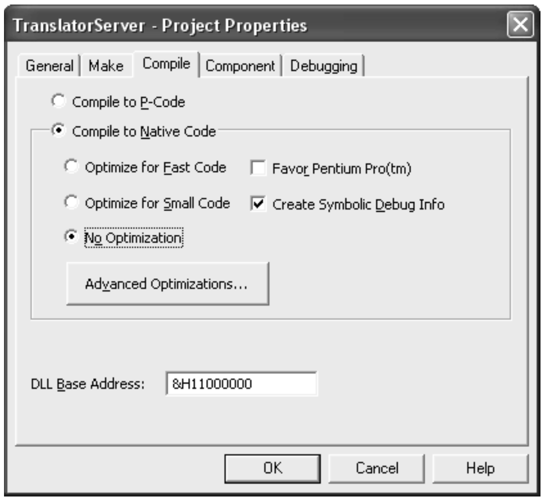
Debugging Between the Visual Basic .NET Client and Visual Basic 6 Server

When upgrading Visual Basic 6 code to Visual Basic .NET, it is critical that you be able to debug the changes that you made yourself or that were made by the Upgrade Wizard. The Microsoft Visual Studio .NET development environment makes it possible to debug between Visual Basic 6 applications and Visual Basic .NET. However, you need to make a few changes to your Visual Basic 6 code in order to debug it using the Visual Studio .NET debugger.

Preparing Your Visual Basic 6 Project to Debug Using Visual Studio .NET

To be able to set breakpoints in your Visual Basic 6 source code, you need to build your Visual Basic 6 project with debugging symbols turned on. To do so, perform the following steps:

1. Run Visual Basic 6 and load the application you want to debug. In this case, load TranslatorServer.vbp.
2. From the Project menu, choose TranslatorServer Properties.
3. Click the Compile tab and select Create Symbolic Debug Info.
4. For best results, select No Optimization. See Figure 9-5.


TranslatorServer - Project Properties

General | Make | Compile | Component | Debugging

- ○ Compile to P-Code
- ● Compile to Native Code
  - ○ Optimize for Fast Code
  - ○ Optimize for Small Code
  - ● No Optimization
  - ☐ Favor Pentium Pro(tm)
  - ☑ Create Symbolic Debug Info

Advanced Optimizations...

DLL Base Address: &H11000000

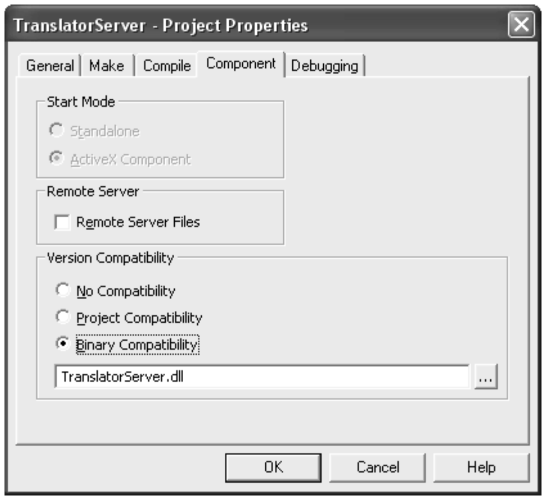
OK Cancel Help

Figure 9-5 Recommended Visual Basic 6 options for debugging.

At this point it's a good idea to turn on binary compatibility so that you do not need to update your .NET client application.

5. Click the Component tab.
6. Change the Version Compatibility setting from Project Compatibility to Binary Compatibility, as shown in Figure 9-6.
7. Click OK to close the Project Properties dialog box.

Now rebuild your application. Choose Make TranslatorServer.dll from the File menu, click OK, and click Yes to replace the existing file, if prompted to do so.



Start Mode
- Standalone [✓ ActiveX Component]

Remote Server
- Remote Server Files ☐

Version Compatibility
- No Compatibility
- Project Compatibility
- [✓ Binary Compatibility]

TranslatorServer.dll

Figure 9-6 Setting binary compatibility in Visual Basic 6.

Now you're ready to debug.

1. Run Visual Studio .NET and open the MyAmazingTranslator Web project you created earlier.
2. Select the MyAmazingTranslator project in Solution Explorer.
3. From the Project menu, choose Properties.
4. Expand the Configuration Properties folder and select Debugging.
5. Under Debuggers, click Unmanaged Code Debugging.
6. Click OK to close the dialog box.
7. Open WebForm1.aspx, and double-click the Translate button.

8. Insert a breakpoint on the following line:

```
TextBox1.Text = ts.Translate(TextBox1.Text, "English", _
   "Spanish")
```

9. From the File menu, choose Open File, and open TranslatorServer.cls, saved as part of the Visual Basic 6 TranslatorServer project.
10. Right-click the first line of code in the *Translate* function, and select Insert Breakpoint.
11. Choose Start from the Debug menu.
12. Click the Translate button. Execution should break on the following line:

```
TextBox1.Text = ts.Translate(TextBox1.Text, "English", _
   "Spanish")
```

13. Step over the line, and execution should break in your Visual Basic 6 TranslatorServer.cls code file. You can now step through your Visual Basic 6 code to ensure that everything is working properly.

Note In order to debug an ASP.NET application on your local machine, you need to be added as a member of the Debugger Users group. If you do not have administrative privileges on the machine, you also need to change the Machine.config file for Aspnet _wp.exe to run Aspnet_wp.exe with User privileges rather than System account privileges. See Visual Studio .NET's Help system for more details.

Exposing a Visual Basic .NET Component to Be Called by a Visual Basic 6 Client

In some cases, you will want to upgrade your Visual Basic 6 server and make it available to your Visual Basic 6 or other COM client applications. For example, since Visual Basic .NET allows you to create multithreaded components, you may want to upgrade your Visual Basic 6 component to a .NET component. You can then register the component to take advantage of a multithreaded environment such as Microsoft Transaction Server (MTS) in order to use object pooling, for example.

Let's take the Visual Basic 6 TranslatorServer component located on the companion CD and upgrade it to Visual Basic .NET by performing the following steps:

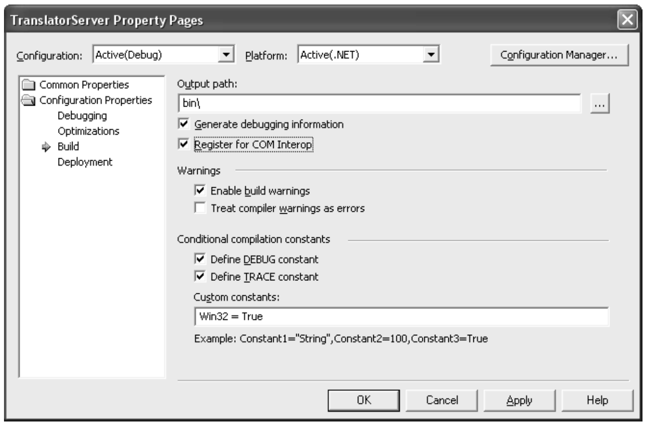
1. Run Visual Studio .NET.
2. From the File menu, choose Open Project and open TranslatorServer.vbp.
3. Step through the Upgrade Wizard by clicking Next and selecting the default options as you go.

   Let's change the name of the .NET server so that it doesn't conflict with the server name of its COM predecessor.
4. Select the TranslatorServer project in the Solution Explorer.
5. From the Project menu, choose Properties.
6. Change the Assembly Name from TranslatorServer to TranslatorServer.net.
7. Click OK to close the Project Properties dialog box.
8. View the code for TranslatorServer.vb:

```
Option Strict Off
Option Explicit On
Public Class Translator

Public Function Translate(ByVal SentenceFrom As String, _
   ByVal LanguageFrom As String, ByVal LanguageTo As String) _
   As String

      ' Note to self: Find someone who speaks Spanish and VB
      ' who is willing to expand this component to translate
      ' any common English phrase
      If LCase(LanguageFrom) = "english" And _
         LCase(LanguageTo) = "spanish" Then
         Select Case LCase(SentenceFrom)
            Case "hello world"
               Translate = "hola mundo"
         End Select
      End If

   End Function
End Class
```

Note that the upgraded code, specifically the contents of the *Translate* function, is exactly the same as the Visual Basic 6 code. The point is that you can create a Visual Basic .NET server in exactly the same way that you create a Visual Basic 6 ActiveX DLL server. Now for the gotcha. Although you can build the upgraded Visual Basic .NET server, you cannot call it from Visual Basic 6. Why not? Because by default a Visual Basic .NET server is meant to be called by other .NET components, not by a COM component. To call a Visual Basic .NET server from a COM application such as Visual Basic 6, you need to register the server for COM. The simple way to expose the component to COM is to turn on the Register For COM Interop attribute by doing the following:

1. Select the TranslatorServer project in the Solution Explorer.
2. From the Project menu, choose Properties.
3. Under the Configuration Properties folder, select Build.
4. Select the Register For COM Interop check box, as shown in Figure 9-7, and click OK to close the dialog box.
5. From the Build menu, choose Build Solution, and click the Save button to save the Solution file. This step creates the Visual Basic .NET DLL and also registers it for COM.


TranslatorServer Property Pages

Configuration: Active(Debug) Platform: Active(.NET) Configuration Manager...

- Common Properties
- Configuration Properties
  - Debugging
  - Optimizations
  - Build
  - Deployment

Output path:
bin\

☑ Generate debugging information
☑ Register for COM Interop

Warnings
☑ Enable build warnings
☐ Treat compiler warnings as errors

Conditional compilation constants
☑ Define DEBUG constant
☑ Define TRACE constant
Custom constants:
Win32 = True
Example: Constant1="String",Constant2=100,Constant3=True

OK Cancel Apply Help

Figure 9-7 Registering the component for COM interop.

Note Registering a .NET component for COM interop does not necessarily mean that only COM clients can use the component. Instead, it opens that component up to the world of both COM and .NET. Exposing your .NET server to COM allows you to upgrade old clients or to create new clients in .NET that use the COM server. At the same time you can make a quick modification to other COM clients to allow them to work with your .NET server.

Now you are ready to call the Visual Basic .NET TranslatorServer component from Visual Basic 6. This process involves the same steps used to create the Visual Basic 6 TranslatorClient application located on the companion CD. To call the Visual Basic .NET TranslatorServer component from Visual Basic .NET, follow these steps:

1. Run Visual Basic 6 and open the TranslatorClient application located on the companion CD.
2. Choose References from the Project menu.
3. Deselect TranslatorServer, the Visual Basic 6 server component.
4. Select TranslatorServer_net, the Visual Basic .NET server component.
5. Open TranslatorClient.bas.
6. Change the following declaration from

   ```
   Dim Ts As New TranslatorServer.Translator
   ```

   to

   ```
   Dim Ts As New TranslatorServer_net.Translator
   ```

7. Run the application.

   You should see the results shown in Figure 9-3.

> **Note** If you want to add features to your upgraded Visual Basic .NET component, you should add COM attributes to ensure binary compatibility. Binary compatibility causes your Visual Basic .NET component to look, act, and smell like existing COM clients no matter what new features you add to the component. If you break compatibility, your COM clients will not be able to find the .NET component. You will be forced to recompile your COM client against the updated .NET server and redistribute it. For an example of how to add COM attributes to ensure binary compatibility, see "Replacing COM with .NET: Binary Compatibility" later in this chapter.

Debugging Between the Visual Basic 6 Client and .NET Server

Earlier we showed you how to debug both Visual Basic 6 and Visual Basic .NET code using the Microsoft Visual Studio .NET debugger. Now we're going to debug our Visual Basic 6 client application and .NET server, using both the

Visual Basic 6 debugger and the Visual Studio .NET debugger to step across the call from Visual Basic 6 code to Visual Basic .NET code. Let's start in the Visual Studio .NET development environment.

1. Run Visual Studio .NET and open the TranslatorServer application you upgraded to Visual Basic .NET earlier.
2. Open TranslatorServer.vb and place a breakpoint within the *Translate* function on the following line:

   ```
   If LCase(LanguageFrom) = "english" And _
      LCase(LanguageTo) = "spanish" Then
   ```
3. Select the TranslatorServer project in Solution Explorer.
4. From the Project menu, choose Properties.
5. Under the Configuration Properties folder, select Debugging.
6. Select Start External Program, click the Browse ("…") button, and search for VB6.exe.
7. From the Debug menu, choose Start. Visual Basic 6 will launch.
8. From Visual Basic 6, open TranslatorClient.vbp.
9. Place a breakpoint on the following line:

   ```
   Translation = Ts.Translate("hello world", "English", _
      "Spanish")
   ```
10. Run the Visual Basic 6 client application.

Execution will break in the Visual Basic 6 application. Step over the line of code.

The Visual Studio .NET debugger will appear, and execution will break in the Visual Basic .NET server application. You can step through and debug your Visual Basic .NET server application code. When execution returns from the Visual Basic .NET server, function execution resumes in the Visual Basic 6 debugger. Pretty cool, eh?

Tying It All Together

As you have seen, COM interop enables you to upgrade your application one piece at a time. You can choose to upgrade one part to .NET while keeping other parts based on COM. If your goal is to move your application to the Web or an intranet environment, plenty of options are available to you. For example, you can create a Web front end that uses the same back-end logic currently in use by your traditional Windows client applications. You can also make your back-end functions—your business logic—available to remote clients on the

C0961587x.fm Page 189 Thursday, November 15, 2001 3:33 PM

Internet. To do so, you can expose your functions as Web services. In the case of the *Translator* class, you can expose the *Translate* function over the Web by adding a Web service class to the project, copying the contents of the *Translator* class to the Web service class, and marking the *Translate* function as a *WebMethod*. We leave this task as an exercise for you to complete.

Replacing COM with .NET: Binary Compatibility

In creating Visual Basic versions 4, 5, and 6, the Microsoft Visual Basic development team worked hard to allow you to create Visual Basic COM components that are backward compatible. This feature was enabled by an innocent-looking check box on the Component tab of the Project Properties dialog box, shown earlier in Figure 9-6. Binary compatibility in Visual Basic 6 enabled you to create version 2 of a COM component that is a compatible replacement for version 1 of that component. What this means is that the version 2 component contains exactly the same public objects, properties, methods, and events as the version 1 component. In addition, it means that the properties, methods, and events appear in exactly the same order as in the version 1 component. The version 2 component also understands how to initialize itself using property settings saved by the version 1 component.

Although the version 2 component needs to look, act, and smell like a version 1 component, the version 2 component can include additional objects, properties, and methods that improve upon the version 1 component. No matter the improvements, however, in order to keep clients of the version 1 server component working with the version 2 component, the version 2 component must expose some subset of itself as a version 1 component. If the version 2 component doesn't expose itself as a version 1 component, you need to do one of the following:

A. Invest in fixing the version 2 component to behave identically to the version 1 component and then invest further in testing the component to ensure that no existing version 1 clients are broken by the changes.

B. Rename the version 2 component and change its attributes in such a way that it exposes itself as a completely new component. Clients that are using the version 1 component continue to use that component and are not disturbed by the distribution of the version 2 component.

C. Recompile all clients that use the version 1 component to use the version 2 component and then distribute the client and a version 2 server component as a matched pair.

Which option makes more sense? Option B is generally the recommended approach if you have widely distributed clients that rely on version 1 of the server component. If there is any doubt in your mind about whether you can update an existing component or create a new component as a direct replacement of a version 1 component, don't do it. Instead, go with option B. It's the safe bet.

When it comes to creating Visual Basic .NET components that are direct replacements of your Visual Basic 6 components, Visual Basic .NET adopts the option B approach. Simply put, you can't do it, although if the truth be told, it's possible but not easy. You'll sleep much better if you simply assume that you can't directly replace a Visual Basic 6 component with a Visual Basic .NET one.

The Visual Basic team deliberately chose not to include features that would make it easy for you to mark a Visual Basic .NET component as being a compatible replacement for a Visual Basic 6 component. We'll let the following Visual Basic code speak for itself:

```
Const Old_compiler = "VB6.EXE"
Const New_compiler = "VBC.EXE"
Const Old_runtime = "MSVBVM60.DLL"
Const New_runtime = ".NET Framework and Microsoft.VisualBasic.Dll"

If New_compiler <> Old_compiler And New_runtime <> Old_runtime Then
   MsgBox "Minor behavioral differences that are expensive to " & _
          "find and fix"
End If
```

Indirect Replacement Model

Visual Basic .NET takes an indirect approach in creating Visual Basic .NET components that are replacements of your Visual Basic 6 COM components. This approach means that you can create a component that looks and smells like a Visual Basic .NET component but doesn't necessarily act 100 percent like the Visual Basic 6 component it's "replacing." Since to take advantage of new version 2 component features you need to update your existing clients, why not live a little and roll those features into a Visual Basic .NET component? Putting the new features in a separate component has the advantage of not disturbing the version 1 component. Clients that are running against the version 1 component continue to run against it, unaffected by the changes in the version 2 component. When you are ready to move your clients to the version 2 component, you can update and redistribute them as needed.

By using the Visual Basic Upgrade Wizard, you can quickly upgrade your Visual Basic 6 components to Visual Basic .NET components. Although the wizard doesn't give you a full, 100 percent binary-compatible replacement for your

Visual Basic 6 component, what you end up with is pretty close. All properties, methods, and events are brought forward, preserving the public interface of your component. Once you have worked out all of the upgrade-related comments and issues in the upgrade report, you can effectively replace your Visual Basic 6 component with the .NET version by making a quick change to the Visual Basic 6 client application so that it uses your .NET component instead of the Visual Basic 6 component. You then need to rebuild and redeploy your Visual Basic 6 application to take advantage of the new Visual Basic .NET server component.

Enabling Binary Compatibility in Visual Basic .NET Classes

If you want to expose your upgraded Visual Basic .NET server component to a COM client and you plan on making changes to the Visual Basic .NET server component over time, we strongly suggest that you do the following to ensure compatibility:

- Declare all the public class properties and methods in an interface.
- Declare all the public events in a separate event interface.
- Add attributes to the class to declare ID attributes for the class, interface, and event interface.
- Change the class declaration so that it implements both the programmable and event interfaces.

Let's step through an example that demonstrates how to enable binary compatibility in a Visual Basic .NET class. Doing so will ensure that a COM client that uses the class will continue to work with it even after you have added new functionality to the class.

We'll start with a Visual Basic .NET class that was upgraded from Visual Basic 6. Assume that the filename for the class is Class1.vb.

```
Option Strict Off
Option Explicit On

Public Class Translator

   Public Event TranslationError(ByVal ErrorMessage As String)

   Public Function Translate(ByVal SentenceFrom As String, _
                             ByVal LanguageFrom As String, _
                             ByVal LanguageTo As String) As String
```

(continued)

```
        Dim fSuccess As Boolean

        ' Note to self: Find someone who speaks Spanish and VB who
        ' is willing to expand this component to translate any common
        ' English phrase
        If LCase(LanguageFrom) = "english" And _
           LCase(LanguageTo) = "spanish" Then
           Select Case LCase(SentenceFrom)
              Case "hello world"
                 Translate = "hola mundo"
                 fSuccess = True
           End Select
        End If

        If Not fSuccess Then
           RaiseEvent TranslationError( _
              "Translation not implemented for " & SentenceFrom)
        End If
     End Function

End Class
```

1. Choose Add New Item from the Project menu and select COM Class. This gives you a template class for creating binary compatible objects.
2. Copy and paste the public event *TranslationError* and the public method *Translate* to the new COM class. Paste the code after *Sub New*.
3. Right-click Class1.vb in the Solution Explorer and delete it. We no longer need it.
4. View the code for *ComClass1* and expand the #Region "COM GUIDs," since you will need to use these values later.
5. Change the *Public Class* declaration for *ComClass1* to *Translator* as follows:

   ```
   Public Class Translator
   ```

6. Add the following *Imports* clause to the top of the file as follows. COM attributes such as *Guid* and *ComVisible* will come from this namespace.

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

7. Replace the *ComClass* attribute in the *Translator* class with the following attributes:

```
<Guid(ClassId), ComVisible(True)> _
Public Class Translator
```

8. Define the programmable interface containing the *Translate* method. Define the interface after the #End Region for "COM GUIDs" as follows:

```
<Guid(InterfaceId)> _
Interface ITranslate
   <DispId(1)> _
   Function Translate(ByVal SentenceFrom As String, _
                      ByVal LanguageFrom As String, _
                      ByVal LanguageTo As String) As String
End Interface
```

9. Immediately following the programmable interface, insert the declaration for the event interface as follows:

```
<Guid(EventsId)> _
Interface ITranslateEvents
   Event TranslationError(ByVal ErrorMessage As String)
End Interface
```

10. Add an *Implements* clause after the *Public Class* declaration to implement the programmable and event interfaces you just declared as follows:

```
Public Class Translator
   Implements ITranslate, ITranslateEvents
```

11. Modify the *Public Event* declaration to implement the interface event as follows:

```
Public Event TranslationError(ByVal ErrorMessage As String) _
   Implements ITranslateEvents.TranslationError
```

12. Modify the public *Translate* method to implement the *ITranslate.Translate* interface method as follows:

```
Public Function Translate(ByVal SentenceFrom As String, _
                          ByVal LanguageFrom As String, _
                          ByVal LanguageTo As String) _
                             As String _
                          Implements ITranslate.Translate
```

When completed, you should have the following class with all the necessary COM attributes defined. You can rebuild the class without breaking compatibility with existing COM clients. The only way you will break compatibility is if you consciously change GUIDs, change a method or event signature, or change the order of events or methods defined in an interface. To maintain compatibility from this point on, you must restrict yourself to adding new methods to the end of an interface. You can never remove or change an existing method, nor can you can change the order of the methods.

```
Imports System.Runtime.InteropServices

<Guid(ClassId), ComVisible(True)> _
Public Class Translator
   Implements ITranslate, ITranslateEvents

#Region "COM GUIDs"
   ' These  GUIDs provide the COM identity for this class
   ' and its COM interfaces. If you change them, existing
   ' clients will no longer be able to access the class.
   Const ClassId As String = "4B33F612-69C0-4270-A7F3-1B460EBDE978"
   Const InterfaceId As String = _
      "D1C5A219-A6C9-46CA-939F-8CF2D22FE441"
   Const EventsId As String = "86215D08-D4AD-4CA8-9BAC-01E8A592812D"
#End Region

   <Guid(InterfaceId)> _
   Interface ITranslate
      <DispId(1)> _
      Function Translate(ByVal SentenceFrom As String, _
                         ByVal LanguageFrom As String, _
                         ByVal LanguageTo As String) As String
   End Interface

   <Guid(EventsId)> _
   Interface ITranslateEvents
      Event TranslationError(ByVal ErrorMessage As String)
   End Interface

   ' A creatable COM class must have a Public Sub New()
   ' with no parameters; otherwise, the class will not be
   ' registered in the COM registry and cannot be created
   ' via CreateObject.
   Public Sub New()
      MyBase.New()
   End Sub

   Public Event TranslationError(ByVal ErrorMessage As String) _
      Implements ITranslateEvents.TranslationError
   Public Function Translate(ByVal SentenceFrom As String, _
                             ByVal LanguageFrom As String, _
                             ByVal LanguageTo As String) As String _
                             Implements ITranslate.Translate

      Dim fSuccess As Boolean

      ' Note to self: Find someone who speaks Spanish and VB who
      ' is willing to expand this component to translate any common
      ' English phrase
      If LCase(LanguageFrom) = "english" And _
         LCase(LanguageTo) = "spanish" Then
         Select Case LCase(SentenceFrom)
            Case "hello world"
               Translate = "hola mundo"
               fSuccess = True
         End Select
      End If

      If Not fSuccess Then
         RaiseEvent TranslationError( _
            "Translation not implemented for " & SentenceFrom)
      End If

   End Function

End Class
```

Note In order to control the binary compatibility of components, many of you asked for a Visual Basic feature to expose COM attributes such as *Guid* and method IDs in a Visual Basic class. Visual Basic 6 offers a simple way to enable binary compatibility—the Binary Compatibility option—but does not allow you to define or edit the COM attributes that it automatically generates for you. Visual Basic .NET does not offer any simple solution to binary compatibility; there is no binary compatibility option that you can turn on. Instead, you must write code manually defining the interfaces and COM attributes to enforce binary compatibility. The ability to specify COM attributes comes at a cost: more code. After you have “beautified” your code by manually adding these attributes, you may ask what you have gotten yourself into. Be careful what you ask for—particularly if you’ve requested more control in specifying COM attributes; you just might get it.

Conclusion

You can mix and match COM with .NET in any number of ways, depending on the needs of your business. There are economic and practical reasons for choosing to have one part of your application based on .NET and the other part based on COM. The beauty of COM interop is that it gives you a choice. When upgrading your applications, you don't have to standardize on .NET immediately to take advantage of the new features. You can start using those new features now and continue to use your existing Visual Basic 6 code and components as needed to accomplish your goals more quickly and effectively.