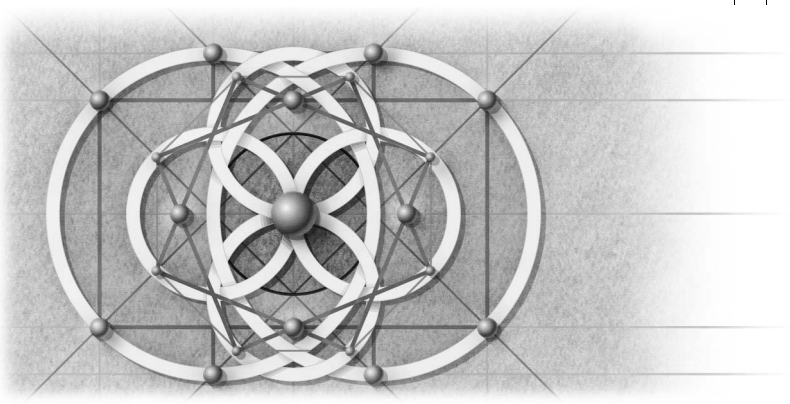

19

Replacing ActiveX Controls with Windows Forms Controls

If you've been using Microsoft Visual Basic long enough—since before Visual Basic 4, to be exact—you will have fond memories of upgrading your controls from one control model to another: from VBX to OCX. When Visual Basic pole-vaulted to 32 bits, VBX—short for Visual Basic extension—controls didn't make it over the bar and were left behind. Visual Basic 4 32-Bit Edition offered OCX—short for OLE control extension—replacements for most of the VBX controls delivered with Visual Basic 4 16-Bit Edition. In order to move your Visual Basic application to 32 bits, you were forced to upgrade your VBX controls to OCX controls. Visual Basic 4 provided a feature that would automatically replace the VBX controls in your project with equivalent OCX controls—commonly referred to as ActiveX controls today.

Visual Basic .NET offers a new control model called Windows Forms controls. Although there are many Windows Forms controls that are equivalent to the ActiveX controls you find in Visual Basic 6, you are not forced to replace all of your ActiveX controls with Windows Forms equivalents; Visual Basic .NET supports ActiveX controls as is. In fact, as we discussed in Chapter 13, the Upgrade Wizard—except in limited situations—does not replace your ActiveX controls with equivalent Windows Forms controls, even when equivalent controls exist.

This chapter discusses how you can manually replace your ActiveX controls with equivalent Windows Forms controls. In addition, it discusses the benefits of making this transition.

Benefits of Upgrading Controls

Upgrading from ActiveX controls to Windows Forms controls has certain benefits. As you'll see in this section, these benefits include 100 percent Microsoft .NET compatibility, improved versioning, and simpler deployment.

100 Percent .NET Compatibility

Windows Forms controls have an inherent advantage over ActiveX controls in that they are native to the .NET environment, and thus they are more tightly integrated into that environment. For example, a Windows Forms control can support custom property editors—such as a TreeView nodes collection editor—for use with the Property Browser.

Because ActiveX controls are based on COM and are not native to the .NET environment, you may encounter some issues with them. For example, custom property editors are not available for ActiveX controls. An ActiveX control may depend on functionality provided by the Visual Basic 6 environment, which does not exist in Windows Forms. Thus, the capabilities of the ActiveX control may degrade when placed on a Windows form. For example, the ActiveX SSTab control depends on a specific host interface in order to support child controls. Windows Forms does not support the host interface SSTab is looking for, so the control cannot accept child controls when it exists on a Windows form.

Improved Versioning

You can think of ActiveX controls and components as being global across all applications. When an ActiveX control is registered on your machine, all applications in which the control is used will share the control from the same location. If you download an update for the ActiveX control, all of your applications automatically use the updated control. This global use of controls can be good and bad. It's good if the updated control fixes bugs that affect one or more of your applications, but it's bad if the updated control introduces new bugs or incompatibilities that break your applications. Because accepting an update for an ActiveX control is an all-or-nothing proposition, the potential for doing more harm than good is high. The updated control might benefit some applications while breaking others.

Applications that are built with Windows Forms controls do not risk being broken when you download an update for a control. The reason for this is that Windows Forms controls are local to each application, meaning that each application is bound to a particular version of the control. If an updated control becomes available on a system, the application does not automatically pick it

up. Instead, it continues to use the version of the control that it was originally built against. If you want the application to pick up the latest version of the control automatically, you can change the policy for the application to cause it to do so. You can do this on an application-by-application basis to ensure that applications that will benefit from the updated control will use it, whereas ones that won't benefit or that risk being broken continue to use the version of the control known to work with the application.

This change means that if you have an application based purely on Windows Forms controls that you have built, tested, and deployed, you can count on the application to run on a machine without disruption. No future control updates installed on the machine can affect it. The application will continue to run with the controls that it was tested against.

Simpler Deployment

When you deploy an application based entirely on Windows Forms controls and .NET components, you will find that fewer files are needed. If your application contains ActiveX controls, more files must be deployed, since an extra DLL is generated for each ActiveX control. The extra DLL is the COM interop assembly needed to facilitate the communication between Windows Forms and the ActiveX control.

If you are using Windows Forms controls that are equivalent to the ActiveX controls provided by Visual Basic 6 in your application—such as RichText, ProgressBar, StatusBar, Toolbar, TreeView, and ListView—the controls are provided as part of the Windows Forms assembly. No additional DLLs are required when you use these controls. If you are using controls provided by an outside vendor, you can deploy the control by copying the control assembly (DLL) to your application directory—the directory where your EXE file lives. No registration of the Windows Forms control is required.

Process of Replacing Controls

Because the Visual Basic .NET Upgrade Wizard does not automatically replace your ActiveX controls with Windows Forms equivalents, the process of replacing the ActiveX controls in your upgraded project with equivalent controls provided in the Windows Forms package is a manual one. This process involves the following general steps:

1. Copy the design-time property settings for the ActiveX control.
2. Delete the ActiveX control from the form.
3. Place the equivalent Windows Forms control on the form.

4. Rename the Windows Forms control to match the name of the ActiveX control.
5. Paste the design-time settings for the control.
6. Fix up the design-time settings.
7. Resolve errors in the code.
8. Remove the ActiveX control library reference.

The example in this section demonstrates each of these steps in turn.

Manually Upgrading a Control

Let's take a look at a simple example of replacing the ActiveX ProgressBar control with the Windows Forms ProgressBar control. First you need to start with a Visual Basic 6 application that uses the ProgressBar control.

The companion CD includes a sample Visual Basic 6 application called PBar that serves as the basis for demonstrating how to manually upgrade an ActiveX control to its Windows Form equivalent control. Specifically, it demonstrates how to manually upgrade the ActiveX ProgressBar control to the Windows Forms ProgressBar control. The application consists of a form containing a ProgressBar ActiveX control and a command button. The command button click event (*Command1_Click*) contains the following code, which we'll use to demonstrate how to upgrade code associated with an ActiveX control:

```
Dim i As Integer
Dim StartTime As Single

For i = ProgressBar1.Min To ProgressBar1.Max

   StartTime = Timer
   Do While Timer - StartTime < 0.01
   Loop

   ProgressBar1.Value = i
   ProgressBar1.Refresh

Next
```

The sample application also includes the following *ProgressBar Click* event for the purpose of demonstrating how to upgrade an event associated with an ActiveX control:

```
Private Sub ProgressBar1_Click()
   MsgBox "You clicked my progress bar!"
End Sub
```

First load the Visual Basic 6 sample PBar application and try it out. Then choose Start from the Run menu, and click the Show Progress button. You should see the progress automatically increment from minimum to maximum. Figure 19-1 shows an example of the application running.

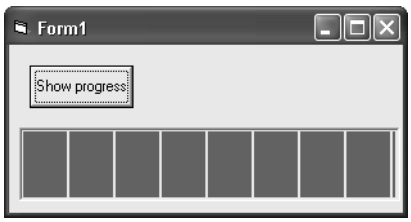

Figure 19-1 Visual Basic 6 ProgressBar application.

Now you'll upgrade the application to Visual Basic .NET. To do this, copy the PBar project from the companion CD to your hard drive, run Visual Basic .NET and open the project file for the Visual Basic 6 application—PBar.vbp. The Upgrade Wizard appears. Step through each of the wizard's pages, selecting the default options. The upgraded application will be loaded in Visual Basic .NET. Test it by choosing Start from the Debug menu and clicking the Show Progress button. The application should run as it did in Visual Basic 6.

In the next section, we will replace the ProgressBar ActiveX control, following the steps outlined earlier.

Copy the Design-Time Property Settings for the ActiveX Control

Design-time settings are stored in two places: in the upgraded form file and in the original Visual Basic 6 form file. The upgraded form file contains the extended design-time settings for the control. These are the settings—common to all controls on the form—that define attributes such as the control's name, its size and position on the form, and its tab order. The Visual Basic 6 form file contains custom property settings for the form. In the case of the ProgressBar, these settings include the *Max* and *Min* property settings.

Copy the extended property settings Let's start by copying the upgraded extended property settings found in the upgraded form file. To see these property settings, view the code for the Visual Basic .NET form file and expand the "Windows Form Designer generated code" region. The property settings for the ProgessBar control are located in the *InitializeComponent* subroutine as follows:

```
Me.ProgressBar1.Location = New System.Drawing.Point(8, 64)
Me.ProgressBar1.Name = "ProgressBar1"
```

(continued)

```
Me.ProgressBar1.OcxState = _
   CType(resources.GetObject("ProgressBar1.OcxState"), _
   System.Windows.Forms.AxHost.State)
Me.ProgressBar1.TabIndex = 0
Me.ProgressBar1.Size = New System.Drawing.Size(297, 57)
```

Select all property settings and copy the code to the Clipboard. Run Notepad and paste the code into it. Notepad will serve as a temporary holding place for this code. Delete the line containing the *OcxState* setting. *OcxState* represents the internal, saved state of the ActiveX control and cannot be applied to the ProgressBar Windows Forms control. It is easier to get the *OcxState* information from the original Visual Basic 6 form file, as we will demonstrate in the next section.

Copy the control-specific property settings Now let's copy the property settings found in the original Visual Basic 6 form file. Run another instance of Notepad, and open PBar.frm (copied previously from the companion CD). Look for the following section in the FRM file:

```
Begin MSComctlLib.ProgressBar ProgressBar1
   Height          =   855
   Left            =   120
   TabIndex        =   0
   Top             =   960
   Width           =   4455
   _ExtentX        =   7858
   _ExtentY        =   1508
   _Version        =   393216
   Appearance      =   1
   Min             =   1
   Max             =   200
End
```

Let's copy the ActiveX control-specific property settings. In this case, the control-specific property settings are *Appearance*, *Min*, and *Max*. Custom property settings appear in the FRM file's property settings block for a control, after the extended property settings. Extended property settings relate to properties that Visual Basic 6 maintains on behalf of the ActiveX control. The following list contains the extended properties supported by Visual Basic 6. You do not need to copy these settings when replacing the ActiveX control with a Windows Forms control. The reason is that the equivalent Visual Basic .NET property settings can be found in the *InitializeComponent* subroutine of the upgraded form file. It is easier to copy the upgraded extended property settings found in the

Visual Basic .NET form file than it is to copy the ones found in the original Visual Basic 6 FRM file, as we demonstrated in the previous section.

_ExtentY	DataFormat	Left
_ExtentX	DataMember	Name
_Version	DataSource	TabIndex
Extender properties	Default	TabStop
Align	DragIcon	Tag
Cancel	DragMode	TooltipText
CausesValidation	Enabled	Top
Container	Height	Visible
DataBindings	HelpContextID	WhatsThisHelpID
DataChanged	Index	Width
DataField		

Copy the following settings to the original instance of Notepad containing the extended property settings. Paste them after the extended property settings.

```
Appearance      =   1
Min             =   1
Max             =   200
```

After you've copied the settings, Notepad should contain the following:

```
Me.ProgressBar1.Location = New System.Drawing.Point(8, 64)
Me.ProgressBar1.Name = "ProgressBar1"
Me.ProgressBar1.TabIndex = 0
Me.ProgressBar1.Size = New System.Drawing.Size(297, 57)
Appearance      =   1
Min             =   1
Max             =   200
```

Modify the settings in Notepad by converting each of the settings you've just copied to Visual Basic code in the form Me.<*controlname*>.<*property-name*> = <*value*>. Your code should appear as follows after modification:

```
Me.ProgressBar1.Location = New System.Drawing.Point(8, 64)
Me.ProgressBar1.Name = "ProgressBar1"
Me.ProgressBar1.TabIndex = 0
Me.ProgressBar1.Size = New System.Drawing.Size(297, 57)
```

(continued)

```
Me.ProgressBar1.Appearance = 1
Me.ProgressBar1.Min = 1
Me.ProgressBar1.Max = 200
```

Delete the ActiveX Control from the Form

The next step is an easy one. Switch to Design view for the form, and delete the ProgressBar ActiveX control from the upgraded Visual Basic .NET form, Form1.

Place the Equivalent Windows Forms Control on the Form

Select the Windows Forms ProgressBar control on the Windows Forms tab of the Toolbox. Place the control on Form1. You don't need to worry about placing the control in exactly the same position as the previous control. The position and size will be restored later when you copy the code from Notepad.

Rename the Windows Forms Control to Match the Name of the ActiveX Control

Change the *Name* property of the control to *ProgressBar1* to match the name of the original ActiveX control.

Paste the Design-Time Settings for the Control

All that monkeying around you did in Notepad will now pay off. Copy the property settings you created in Notepad to the *InitializeComponent* subroutine in Form1, and replace the existing property settings for ProgressBar that you find. The *InitializeComponent* subroutine is located within the hidden block labeled "Windows Form Designer generated code."

Fix Up the Design-Time Settings

Compiler errors will display for any property settings that the compiler doesn't recognize. For example, a compiler error will occur on the following three lines of code. It occurs because the property cannot be found in the Windows Forms ProgressBar control. You will need to either eliminate property settings for properties that are no longer supported or change the property name to the equivalent property found in the Windows Forms control.

```
Me.ProgressBar1.Appearance = 1
Me.ProgressBar1.Min = 1
Me.ProgressBar1.Max = 200
```

Using IntelliSense, you can quickly get a feel for what properties the Windows Forms ProgressBar control does and doesn't support. To see a list of all properties associated with the ProgressBar control, type the name of the control

in the code window, followed by a period. A drop-down list of the available properties and methods displays. You will find that the *Appearance* property is no longer supported, *Min* maps to *Minimize*, and *Max* maps to *Maximize*. Making the appropriate changes causes the compiler errors related to design-time settings to disappear. The resulting *InitializeComponent* code is as follows:

```
Me.ProgressBar1.Minimize = 1
Me.ProgressBar1.Maximize = 200
```

Resolve Errors in the Code

Compiler errors are displayed for any code in which a property cannot be resolved. In the *Command1_Click* event, you will see three compiler errors related to the use of the properties and methods *Min*, *Max*, and *CtlRefresh*. In line with the changes you made to the design-time properties, change *Min* to *Minimum* and *Max* to *Maximum*. The *CtlRefresh* method for the ActiveX ProgressBar control corresponds to the *Refresh* method for the Windows Forms ProgressBar control. The ActiveX control property is renamed *CtlRefresh* in Windows Forms to avoid conflicts with the extended *Refresh* property applied to all controls. The Windows Forms ProgressBar control contains a single method called *Refresh*. Rename *CtlRefresh* to *Refresh*.

In addition to fixing up properties and methods, you need to update your event code. Specifically, when you delete the ActiveX control, the events for the control become disassociated and are converted to ordinary subroutines. You need to reassociate the events with the Windows Forms control. In our example, code is written in response to the ProgressBar *Click* event. When the ProgressBar ActiveX control is deleted, the following subroutine exists but is not associated with the Windows Forms ProgressBar control:

```
Private Sub ProgressBar1_ClickEvent( _
   ByVal eventSender As System.Object, _
   ByVal eventArgs As System.EventArgs)
   MsgBox("You clicked my progress bar!")
End Sub
```

The easiest way to reassociate the *Click* event with the ProgressBar control is to act as though you were writing new code for the *Click* event. Select the *Click* event for the ProgressBar from the Event drop-down list to create the *ProgressBar1_Click* event handler subroutine, as follows:

```
Private Sub ProgressBar1_Click(ByVal sender As Object, _
   ByVal e As System.EventArgs) Handles ProgressBar1.Click
End Sub
```

Cut and paste the body of the *ProgressBar1_ClickEvent* subroutine to the *ProgressBar1_Click* event subroutine, and then delete the original *ProgressBar1_Click* event subroutine, leaving the following code:

```
Private Sub ProgressBar1_Click(ByVal sender As Object, _
   ByVal e As System.EventArgs) Handles ProgressBar1.Click
   MsgBox("You clicked my progress bar!")
End Sub
```

Re-creating the event subroutines in this manner causes the correct event parameters to be created in turn. You then need to resolve any compiler errors that result. For example, you may have code that references a method of an ActiveX control's event argument object, but the method may not exist or may have a different name in the event argument object of the corresponding Windows Forms control.

Remove the ActiveX Control Library Reference

The final step is optional and should be performed only if you have replaced all ActiveX controls for a given library with Windows Forms equivalents. In the case of the example demonstrated here, all controls contained in the Windows common controls library—the ProgressBar control—have been replaced with Windows Forms equivalents. Therefore, you can safely remove the reference to the ActiveX control library. To remove the reference, expand the References list for the project in the Solution Explorer. You will need to remove two references for each ActiveX control. Right-click AxMSComctlLib, and choose Remove from the shortcut menu. Then right-click MSComctlLib and choose Remove. Removing an ActiveX control library reference in this manner means that fewer files need to be deployed with the application.

With these changes in place, all compiler errors should be resolved. Run the application and test it out. It should behave exactly like the original Visual Basic 6 application.

Mappings for Visual Basic 6 ActiveX Controls

Visual Basic 6 provides a number of ActiveX controls, such as the RichText, Masked Edit, and Common Dialog controls, as well as the Windows common controls—ProgressBar, Slider, StatusBar, Toolbar, TreeView, and so on. Table 19-1 lists the Windows Forms controls that make suitable replacements for Visual Basic 6 ActiveX controls.

Table 19-1 Mapping of Visual Basic 6 ActiveX Controls to Visual Basic .NET Windows Forms Controls

Visual Basic 6 ActiveX Control	ActiveX Control Library	Visual Basic .NET Windows Forms Control
ADO Data control*	MSAdodc.ocx	ADODC control provided by the Visual Basic compatibility library
Common Dialog	ComDlg32.ocx	OpenFileDialog, SaveFileDialog, FontDialog, ColorDialog, and PrintDialog
DataGrid	MSDatGrd.ocx	No equivalent control; Windows Forms DataGrid does not bind to ADO data, just to ADO.NET data
Rich TextBox	RichTx32.ocx	RichTextBox
SSTab*	TabCtl32.ocx	TabControl
ListView	MSComCtl.ocx	ListView
TreeView	MSComCtl.ocx	TreeView
ImageList	MSComCtl.ocx	ImageList
TabStrip	MSComCtl.ocx	TabControl
Toolbar	MSComCtl.ocx	ToolBar
StatusBar	MSComCtl.ocx	StatusBar
ProgressBar	MSComCtl.ocx	ProgressBar
Slider	MSComCtl.ocx	TrackBar
UpDown	MSComCt2.ocx	NumericUpDown
MonthView	MSComCt2.ocx	MonthCalendar
DTPicker	MSComCt2.ocx	DateTimePicker

* Denotes an ActiveX control that is automatically upgraded to the equivalent Windows Forms control.

ActiveX Controls vs. Windows Forms Controls

For the most part, the Windows Forms controls listed in Table 19-1 offer the same functionality as their ActiveX control counterparts. One of the main differences—as demonstrated by the ProgressBar upgrade example given earlier—is that features of a Windows Forms control are not exposed in the same way. For example, the same type of properties may exist but have different names. In the case of the ProgressBar control, the Windows Forms control's *Maximum* and *Minimum* properties equate to the *Max* and *Min* properties of the ActiveX control. For properties such as these, a simple one-to-one mapping exists, and it is

easy to find the equivalent property, method, or event when replacing an ActiveX control with a Windows Forms control.

We refer to controls that do not contain any subobjects, such as a collection of items or nodes, as having a **flat object model**. This means that all the properties and methods for the control exist directly on the control. You need to use only one dot (or period) to access any one of the properties or methods when writing code. Replacing a flat-model ActiveX control—such as a ProgressBar or Slider—with its Windows Forms equivalent is a relatively straightforward task. With both controls—the ActiveX version and the Windows Forms version—on a form side by side, you can use IntelliSense or the Object Browser to take a quick inventory of each control and compare how you map the properties, methods, and events from one control to the other.

We refer to controls that have properties that return objects such as collections or other complex structures as having a deep or **rich object model**. Controls such as TreeView and ListView fall into this category. The TreeView control, for example, has a *Nodes* property representing a collection of *Node* objects. The Nodes collection is itself an object having its own set of properties and methods. One of these methods—the *Item* method—returns a *Node* object that in turn has its own set of properties and methods. You can navigate from the control to a child object by using a dot (or period) to separate each successive object in the object hierarchy.

Replacing a rich-model ActiveX control with the equivalent Windows Forms control presents a unique challenge. The challenge stems from differences in how the object models allow you to interact with subobjects of ActiveX and Windows Forms controls. ActiveX controls take a more property-centered approach to adding or navigating a control's subobjects. Windows Forms controls take a more object-oriented approach. This difference affects the way you write code to add or find a subobject for a control. The challenge is to figure out the code that you need to write to achieve the same results in a Windows Forms control as are achieved in its ActiveX counterpart.

Let's take a look at adding child nodes to a TreeView control as an example. This example highlights the general differences between ActiveX and Windows Forms controls having rich object hierarchies, such as the ListView, ToolBar, StatusBar, and tabbed dialog controls.

The following is an example of code written to add a single parent and child node to a TreeView control. The code is associated with a Visual Basic 6 form containing a TreeView control and an ImageList control named TreeView1 and ImageList1, respectively. Assume that the ImageList control contains at least one image added at design time.

```
Set TreeView1.ImageList = ImageList1
TreeView1.Nodes.Add , , "Parent", "Parent node", 1
TreeView1.Nodes.Add 1, tvwChild, "Child", "Child node", 1
```

The ActiveX controls provided with Visual Basic 6 do not allow you to create a subobject directly. For example, you cannot dimension a variable of type *Node* using the *New* qualifier. Instead, you create a node indirectly by calling the *Add* method on the Nodes collection and passing in a number of arguments representing property settings for the newly created node. Because this type of object model directs you to specify the properties of the *Node* object rather than the *Node* object itself, it is a property-centered model for creating nodes.

To achieve the same result using the Windows Forms TreeView control, you must think in a more object-oriented fashion. In the object model used in Windows Forms, it is preferable to create *Node* objects and then add them to the *Nodes* collection. We should note that the Windows Forms TreeView control, and other Windows Forms controls for that matter, do allow you to create collection member objects in a property-centered way. For example, you can create a *Node* object by calling the *Add* method on the *Nodes* collection and passing the *Text* value for the node. However, the *Add* method on collections generally does not support a large number of arguments—unlike collections related to ActiveX controls, where the arguments represent properties for the new item. You must get back into the object-oriented mindset and set the property values on the object after it is created.

The following code is the equivalent of the previous example; it adds a single parent and child node to a Windows Forms TreeView control:

```
Dim nodeChild As TreeNode = New TreeNode("Child node", 0, 0)
Dim childNodes() As TreeNode = {nodeChild}
Dim nodeParent As TreeNode = New TreeNode("Parent node", _
   0, 0, childNodes)

TreeView1.ImageList = ImageList1
TreeView1.Nodes.Add(nodeParent)
```

Compared to the ActiveX TreeView control, the Windows Forms TreeView control requires you to take a more structured, bottom-up approach to creating the nodes within the TreeView. In the Windows Forms version of TreeView, you cannot add individual nodes relative to other nodes in an ad hoc fashion. Instead, you must add the child nodes for a parent node at the time you add the parent node. You do this by passing in an array of *TreeNode* child objects as an argument to the *TreeNode* constructor for the parent node. As this code demonstrates, you add nodes in a bottom-up manner by first allocating the child nodes and then creating the parent node and passing it an array of its children. The code has a more object-oriented feel to it, as you construct the *Node* objects and then pass the objects as arguments to other methods—such as *Add*—to be added to the Nodes collection.

We intentionally used the ImageList control in this example to point out another difference between ActiveX controls and Windows Forms controls that use the rich object model. The ActiveX controls provided in Visual Basic 6 assume the value of 1 as the index for the first item in a collection. Windows Forms controls, on the other hand, always assume an index value of 0 for all collections. That is why the code for the ActiveX TreeView control passes a value of 1 as the index for the first image in the ImageList ListImages collection, whereas the Windows Forms code example uses 0.

Conclusion

Although the Upgrade Wizard makes life easier by having your upgraded Visual Basic 6 applications use the same ActiveX controls, keeping the ActiveX controls means that you do not reap any of the benefits of an application that is based 100 percent on .NET controls and components. Unfortunately, the Upgrade Wizard does not give you the choice of upgrading the ActiveX controls in your project to the equivalent Windows Forms controls.

This chapter has demonstrated how you can manually replace ActiveX controls with equivalent Windows Forms controls after your project has been upgraded. If the ActiveX control you need to replace has a flat object model, it is a relatively straightforward process to map its properties, methods, and events to those of the equivalent Windows Forms control. If, on the other hand, you want to replace an ActiveX control that has a rich object model with the equivalent Windows Forms control, you will need to restructure your code so that it fits the object model defined by the Windows Forms control.