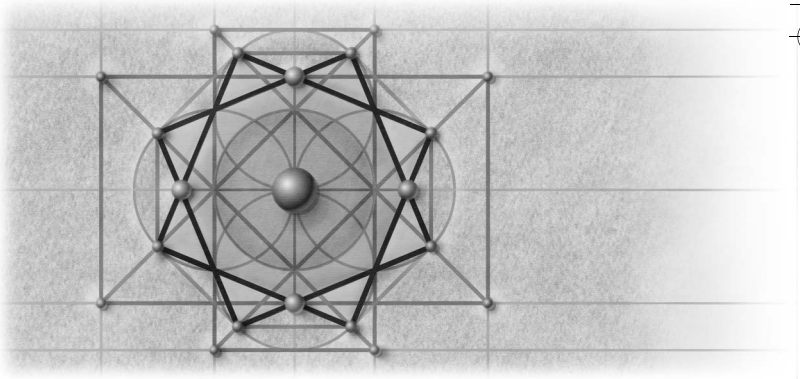


7

Upgrade Wizard Ins and Outs

This chapter focuses on the Visual Basic .NET Upgrade Wizard. Specifically, we'll look at the approach the Upgrade Wizard takes to upgrade your Visual Basic 6 application to Visual Basic .NET. This chapter will address questions such as, What will my code look like after it has been upgraded? What types of projects are upgraded? How will my forms, classes, modules, controls, and ActiveX controls be upgraded? What happens when a component, property, method, event, or language statement can't be upgraded?

Upgrade Philosophy

When the Visual Basic .NET team decided to not support 100 percent Visual Basic 6 compatibility, the challenge to create a tool to upgrade any arbitrary Visual Basic 6 project to Visual Basic .NET was enormous. There was much debate within the team about what approach to take. Ultimately, the Visual Basic team adopted two overriding goals that can be summed up with the following statements:

"It's your code."

"Just make it work."

It's Your Code

The first goal of the Upgrade Wizard is to make the changes to your Visual Basic 6 forms and code as unobtrusive as possible. You should be able to recognize your code after the wizard has upgraded it. If the wizard transforms your

code into an unintelligible pile of junk, you'll probably grab the closest person, point at your code, and say, "Look what Microsoft did to my code. I'm better off rewriting this whole thing myself." To avoid this situation, the Upgrade Wizard was designed with the goal of preserving your code as much as possible.

All of your comments should be preserved and in the same place they were before the upgrade. If you include #If…Then directives, these will also be included in your upgraded project, and you should find them exactly where you put them, relative to everything else. The wizard tries to upgrade each element of your code—language statements, controls, components, and ActiveX references—to the elements in Visual Basic .NET that most closely match.

If, for example, the *MsgBox* statement exists in both Visual Basic 6 and Visual Basic .NET, the wizard will upgrade your code to use *MsgBox*. If there is no direct match, the wizard finds a language statement or component that most closely matches the element you are using. For example, all references to a CommandButton *Caption* property in code are upgraded to the .NET Framework equivalent, *Button.Text* property. If no element in Visual Basic .NET represents the same element in Visual Basic 6, the code is left as is, an UPGRADE_ISSUE comment is inserted above the nonupgraded code, and an issue is logged to the upgrade report.

Just Make It Work

The Upgrade Wizard strives to upgrade all your code from Visual Basic 6 to Visual Basic .NET in a way that allows you to run the upgraded application immediately after upgrade. This works for simple applications or applications that make use of only Visual Basic features that the wizard can upgrade. For larger, more complex applications, you'll need to fix some compiler errors and run-time issues that the Upgrade Wizard can't resolve.

This goal fits hand in hand with the "It's your code" objective. The best way to guarantee that your application will run the same after upgrade is to upgrade the application to use language statements, controls, and objects that behave identically in both Visual Basic 6 and Visual Basic .NET. For example, the Upgrade Wizard could choose to replace all instances of the Microsoft ListView ActiveX control in your application with the .NET Framework ListView control, but it doesn't. The wizard doesn't make this upgrade because by reusing the exact same component, that component—and thus the application—has a better chance of behaving the same as before. If the wizard were to replace your 20 ActiveX controls with 20 noncompatible .NET Framework controls, you would spend more time changing code to run your application. You can get your application up and running more quickly when the upgraded code contains language statements, controls, and objects that are compatible and most familiar to you. Once you get the application running, you can choose to incrementally replace code or components with .NET equivalents, and then test each change as you go.

Compatibility Library

To help your upgraded applications work, Visual Basic .NET includes a compatibility library. This library enables support for features such as control arrays, ADO data binding, and loading resources from a .res file. You can also use some of the functions in the compatibility library to convert between twips and pixels or between System.Drawing.Image and *IPicture*. All objects and functions contained in the compatibility library can be found under the namespace Microsoft.VisualBasic.Compatibility.VB6. The Upgrade Wizard automatically adds a reference to the compatibility library for all upgraded applications.

Upgrade Wizard Capabilities and Limitations

If you own Visual Studio .NET, the best way to get a sense of what the Upgrade Wizard can do for you is to run it. To start the wizard, run Visual Studio .NET and open the .vbp file for your favorite Visual Basic 6 project. The wizard will greet you. Make your way through each step, selecting the appropriate option (defaults work fine in most cases), and then sit back and watch the status for each upgraded item. After the upgrade is complete, look through the upgraded Visual Basic .NET application. As you look over your code, see if you can match up the changes that were made as described in the following sections.

Wizard Methodology

Before we get into the thick of specific changes that the Upgrade Wizard applies to your code, let's take you up to 39,000 feet and look down across the broad landscape of what the Upgrade Wizard does. The wizard will upgrade the following items:

- Your Visual Basic 6 project to the equivalent Visual Basic .NET project type
- Your Forms to .NET Framework Windows Forms equivalents
- Your intrinsic controls to .NET Framework Windows Forms equivalent controls
- Your ActiveX control and object references to use the same ActiveX control and object references
- Your Visual Basic language statements to use the same or equivalent language statements provided by Visual Basic .NET or the .NET Framework (or in some cases the Visual Basic .NET compatibility library)

Figure 7-1 illustrates how the elements of a form are upgraded. Figure 7-2 illustrates how code is upgraded.

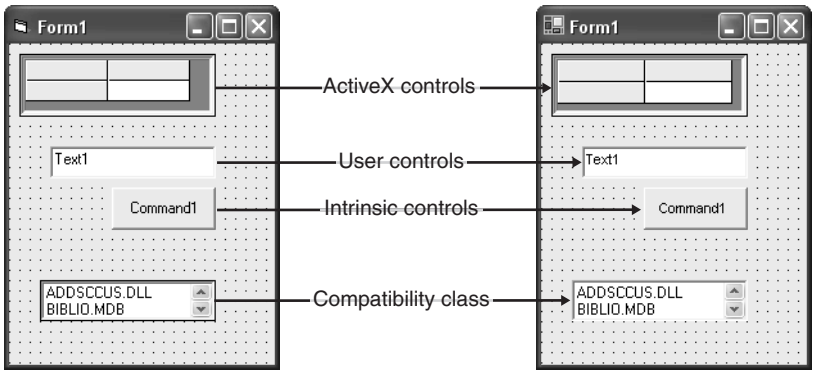


Figure 7-1 How a form is upgraded.

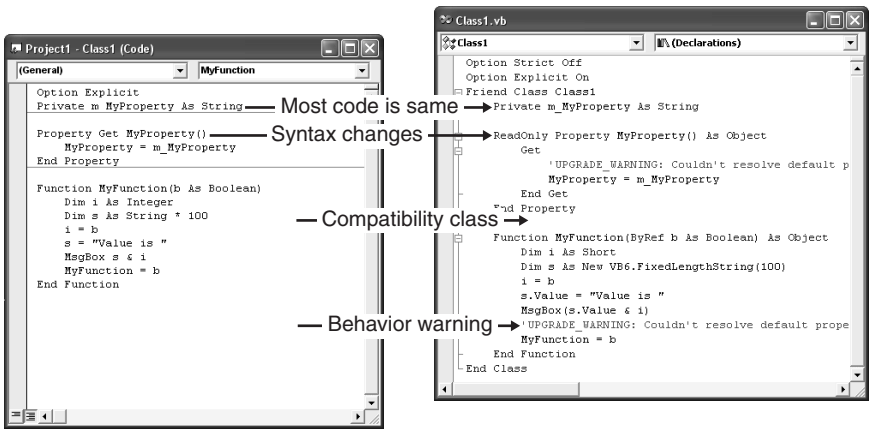


Figure 7-2 How code is upgraded.

The lack of support for certain features in Visual Basic .NET—such as Dynamic Data Exchange (DDE) and ActiveX Documents—means that certain types of Visual Basic 6 applications aren't well suited to a Visual Basic .NET upgrade. These types of applications you can consider in the "out" category when it comes to upgrade. However, just because an application is in the "out" category doesn't necessarily mean that you can't use it in Visual Basic .NET. You might still be able to communicate with your Visual Basic 6 application from Visual Basic .NET without changing it. For example, rather than upgrading your ActiveX DLL project you may want to leave it as is. You can create a Visual Basic .NET application that interoperates with the existing ActiveX DLL.

By reusing existing ActiveX components in your Visual Basic .NET applications you can create the Visual Basic .NET applications more quickly. The general ability of Visual Basic .NET applications to communicate with ActiveX components is known as **COM interop**. You may want to take advantage of COM interop when upgrading your applications so that you don't need to upgrade the application and all of its COM—generally known as ActiveX—dependencies at once. See Chapter 9 for more information on using COM interop.

The Upgrade Wizard will either not upgrade or provide only partial support for the following items:

- ActiveX EXE projects
- Designers such as Add-In Designer, DHTML Page Designer, DataReport, and DataEnvironment
- Certain ActiveX controls: SSTab and UpDown
- ActiveX document–based applications
- Extensibility model–based applications
- OLE container control
- Drag and drop
- Graphics statements and graphical controls
- Dynamic Data Exchange

Project Upgrade

Visual Basic 6 and Visual Basic .NET have the same basic concept of a project: a repository in which you can group all the needed source and resource files to create a specific output file, such as an EXE or a DLL. This section focuses on three project-related areas—types, attributes, and filenames—and discusses how the Upgrade Wizard handles each.

Project Types

Where possible, the Upgrade Wizard will upgrade your Visual Basic 6 project to an equivalent project type in Visual Basic .NET, as shown in Table 7-1. Equivalent Visual Basic .NET project types exist for common project types such as Standard EXE, ActiveX DLL, and UserControl.

> **Note** The version of the Upgrade Wizard that shipped with Visual Basic .NET doesn't include the ability to upgrade UserControl or WebClass projects. However, future versions of the Update Wizard that have this capability will be available for download from the Visual Basic Web site, *http://msdn.microsoft.com/vbasic/.*

Table 7-1 Visual Basic 6 and Equivalent Visual Basic .NET Project Types

Visual Basic 6 Project Type	Visual Basic .NET Project Type
Standard EXE	Windows application
ActiveX EXE	No equivalent (Choose between upgrade to a Windows application or a Class Library project.)
ActiveX DLL	Class Library
ActiveX control	Windows control library
WebClass-based project	XML Web Forms

Project Attributes

A Visual Basic 6 project contains a number of attributes, including Name, command-line arguments, conditional compilation constant settings, Help settings, version information, and compiler settings. Let's look at how the Upgrade Wizard handles each of these attributes.

Project name The project name for a Visual Basic 6 application is upgraded to be the Assembly Name and Root Namespace name for the Visual Basic .NET application. The Assembly Name is the name that other applications or tools use to load the EXE or DLL containing your project. The Root Namespace name is the name that other applications and tools use to refer to your Visual Basic .NET application or library. The Root Namespace name is roughly equivalent to the project name found in Visual Basic 6. Since the Visual Basic 6 project name is applied to your Visual Basic .NET project, anywhere in your code that you fully reference a type by prefacing it with the project name, the code will continue to work as is, without change.

Command-line arguments If you create a Visual Basic 6 EXE project that relies on command-line arguments, you can test your application in the Visual Basic 6 debugger by setting command-line arguments on the Make tab for Project Properties, as shown in Figure 7-3.

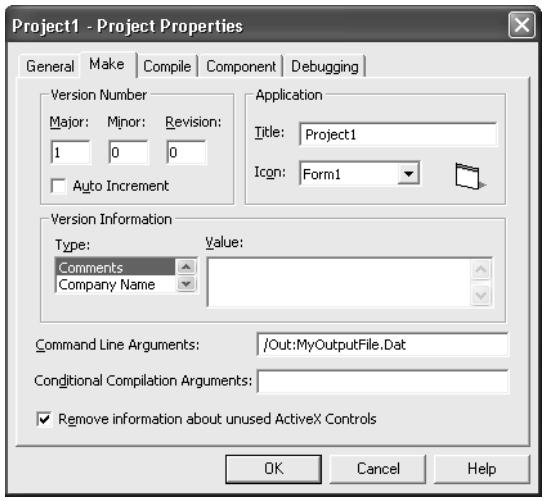


Figure 7-3 Visual Basic 6 command-line setting in the Project Properties dialog box.

The command-line settings aren't upgraded with your project. To edit them in Visual Basic .NET, select the project in the Solution Explorer, and then select Properties from the Project menu. Within the Properties dialog box, select Debugging under the Configuration Properties folder, as shown in Figure 7-4.

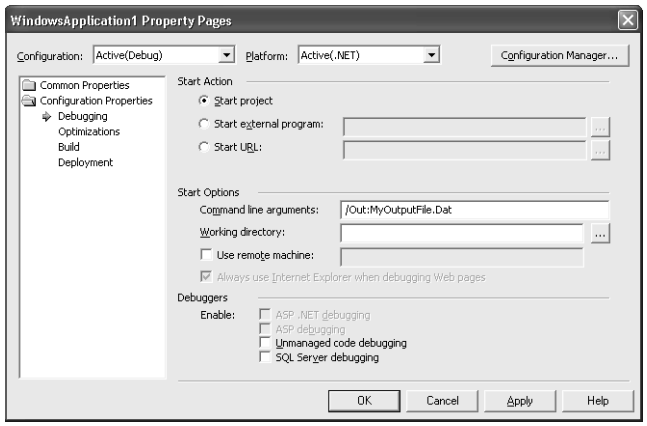


Figure 7-4 Visual Basic .NET command-line setting in Configuration Properties, Debugging.

Conditional compilation constant settings If you use #If…Then conditional statements in your code, you can set the value for the condition as a global project setting. In Visual Basic 6 the setting is located on the Make tab of the Project Properties dialog box, as shown in Figure 7-5.

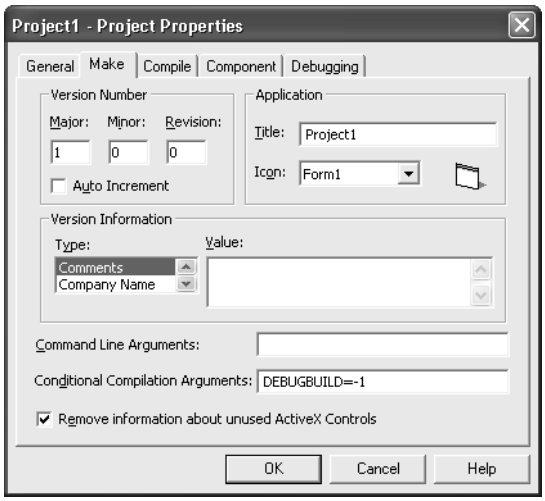


Figure 7-5 Visual Basic 6 conditional compilation statement setting in the Project Properties dialog box.

The constant values that you set will be upgraded to Visual Basic .NET. To edit your settings, select the project in the Solution Explorer and then choose Properties from the Project menu. Within the Properties dialog box, select Build under the Configuration Properties folder, as shown in Figure 7-6.

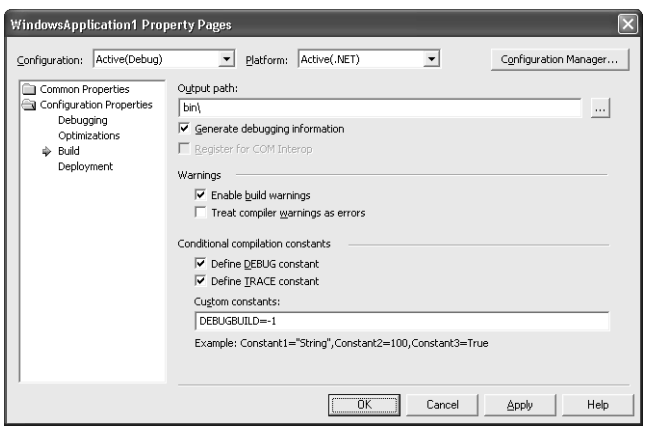


Figure 7-6 Visual Basic .NET custom constants setting under Configuration Properties, Build.

Help settings In Visual Basic 6 you were allowed to have one Help file per project. You specify the Help file name on the General tab of the Project Properties dialog box, as shown in Figure 7-7. You could also set a project-level Help context ID.

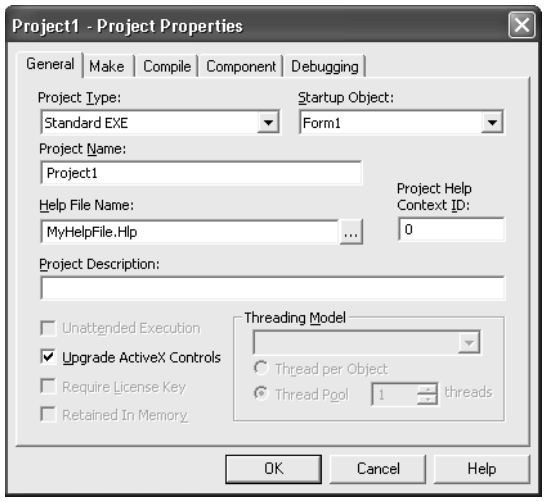


Figure 7-7 Visual Basic 6 Help file setting in the Project Properties dialog box.

Creating Help to go along with your Visual Basic .NET application is different. A Visual Basic .NET application has no concept of a project-level Help file. Instead, you request Help on a per-form basis by including a HelpProvider component on each Windows Form that you want to support Help. Because no top-level Help file is associated with a Visual Basic .NET application, the Upgrade Wizard doesn't upgrade this project setting.

> **More Info** For more information on how to add Help to your Visual Basic .NET application, refer to the following Help topic: ms-help://MS.MSDNVS/vbcon/html/vbconHelpSupportChangesInVisualBasic70.htm. You can find the Help topic by expanding the following nested Help topics:
> Visual Studio .NET; Visual Basic and Visual C#; Upgrading Applications; Upgrading from Visual Basic 6.0; Introduction to Visual Basic .NET for Visual Basic Veterans; Forms Changes in Visual Basic .NET; Help Support Changes in Visual Basic .NET.

Version information The Upgrade Wizard upgrades most version information associated with your Visual Basic 6 application. As illustrated in Figure 7-8, the following attributes are available on the Make tab of the Visual Basic 6 Project Properties dialog box:

- Comments
- Company Name
- File Description
- Legal Copyright
- Legal Trademarks
- Product Name
- Major Version Number
- Minor Version Number
- Revision

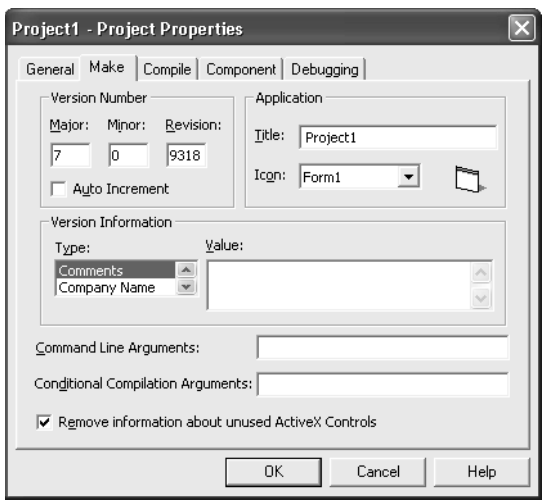


Figure 7-8 Visual Basic 6 version settings in the Project Properties dialog box.

The attributes are upgraded to a separate file in your project called AssemblyInfo.vb, which contains project-level attributes for your project. For

example, after you upgrade a Visual Basic 6 project, the version-related attributes are set in the AssemblyInfo.vb file as follows:

```
<Assembly: AssemblyTitle("Wow! My upgraded application")>
<Assembly: AssemblyDescription("I love upgrading to VB.Net")>
<Assembly: AssemblyCompany("Microsoft")>
<Assembly: AssemblyProduct("MyUpgradedApp")>
<Assembly: AssemblyCopyright("2001")>
<Assembly: AssemblyTrademark("")>
```

> **Note** The revision number that appears in the version number is not upgraded, because Visual Basic 6 and Visual Basic .NET have different revision number formats. In Visual Basic 6 the revision number is a single part of the version number. The Visual Basic .NET version number is a two-part entry composed of a revision number and a build number.

Compiler settings The Upgrade Wizard doesn't upgrade any Visual Basic compiler options to Visual Basic .NET. Because the Visual Basic 6 and Visual Basic .NET compilers generate different types of instructions, the compiler options for each product have nothing in common. The native code optimization settings, such as Remove Array Bounds Checks, don't make sense for the Visual Basic .NET compiler. The Visual Basic .NET compiler is required to generate secure code. Therefore, you don't have the option of turning off certain safety checks, such as array bound checks; all safety checks need to be in place.

Project Filenames

The filename of your Visual Basic 6 .vbp project file is applied to your new Visual Basic .NET .vbproj file. For example, if the name of your Visual Basic 6 project is MyProject.vbp, the name of your Visual Basic .NET project will be MyProject.vbproj.

Individual item filenames Each file contained within a Visual Basic 6 project retains the same base filename after upgrade. The file extension for all upgraded files is set to .vb. For example, if you have a form file in your project called Form1.frm, it will be upgraded to Form1.vb. What if you have two files with the same base filename, such as Main.frm and Main.bas? Since both files can't upgrade to the same filename (Main.vb, in this case), the Upgrade Wizard avoids the conflict by renaming one of the files with its internal name. For

example, if the name of module Main.bas is modMain, the file will be renamed modMain.vb to avoid conflicting with the form file Main.vb.

Forms and Intrinsic Controls

The Upgrade Wizard upgrades forms and intrinsic controls, such as CommandButton, TextBox, OptionButton, and CheckBox, contained on the form to the new .NET Framework Windows Forms package equivalent. After the upgrade process is complete, you'll find that the size and layout of your forms are preserved. If a control cannot be upgraded, a red label is inserted on the upgraded form to alert you of the problem. Table 7-2 shows the mapping of Visual Basic 6 intrinsic controls to their Windows Forms equivalents.

Table 7-2 Mapping Between Visual Basic 6 Intrinsic Controls and .NET Windows Forms Controls

Visual Basic 6 Control Class	Visual Basic .NET Control Class	Upgrade Notes
VB.CheckBox	System.Windows.Forms.Checkbox	
VB.ComboBox	System.Windows.Forms.ComboBox	
VB.CommandButton	System.Windows.Forms.Button	
VB.Data	No equivalent	Because the Data control supports DAO data binding—which is not supported by Windows Forms—the control doesn't upgrade. You can use the ADO data control in its place, update your code to use ADO data binding, and the ADO control will upgrade.
VB.DirListBox	DirListBox in Microsoft.VisualBasic.Compatibility.VB6	
VB.DriveListBox	DriveListBox in Microsoft.VisualBasic.Compatibility.VB6	

Table 7-2 Mapping Between Visual Basic 6 Intrinsic Controls and .NET Windows Forms Controls *(continued)*

Visual Basic 6 Control Class	Visual Basic .NET Control Class	Upgrade Notes
VB.FileListBox	FileListBox in Microsoft.VisualBasic.-Compatibility.VB6	
VB.Frame	System.Windows.-Forms.GroupBox or System.Windows.Forms.-Panel	If the frame has a border, the control upgrades to a GroupBox; it upgrades to a Panel if there is no border.
VB.HScrollBar	System.Windows.-Forms.HScrollBar	
VB.Image	System.Windows.-Forms.Picture	The Visual Basic 6 image control is a graphical control, which can be transparent. Windows Forms doesn't support transparent controls, so transparency will be lost after upgrade.
VB.Label	System.Windows.-Forms.Label	Visual Basic 6 labels are graphical and can be transparent. Visual Basic .NET labels have a window and cannot be transparent. Transparency will be lost after upgrade.
VB.Line	No equivalent or System.Windows.-Forms.Label	For vertical and horizontal lines a label is used. The label is set to the width of the line. The back color of the label is set to the line color. Diagonal lines do not map.
VB.ListBox	System.Windows.-Forms.ListBox	

(continued)

Table 7-2 Mapping Between Visual Basic 6 Intrinsic Controls and .NET Windows Forms Controls *(continued)*

Visual Basic 6 Control Class	Visual Basic .NET Control Class	Upgrade Notes
VB.OLE	No equivalent	In some cases you can replace the control with the Microsoft Web Browser control and use the *Navigate* property.
VB.OptionButton	System.Windows.-Forms.OptionButton	
VB.PictureBox	System.Windows.-Forms.PictureBox or System.Windows.-Forms.Panel	If the PictureBox has child controls, it upgrades to a Panel; otherwise, it upgrades to a PictureBox.
VB.Shape	No equivalent	Shape is a graphical control; Windows Forms doesn't support graphical controls. You can use the .NET Framework graphics commands in place of the control.
VB.TextBox	System.Windows.-Forms.TextBox	
VB.Timer	System.Windows.-Forms.Timer	Setting the *Interval* property of a Visual Basic .NET timer to 0 will throw an exception. You need to set the *Enabled* property of a Timer to False to disable this behavior.
VB.VScrollBar	System.Windows.-Forms.VScrollBar	

Layout and Design-Time Property Settings

The design-time layout and properties you set on a Visual Basic 6 form and the controls contained on the form are saved in a special section of the .frm file. The form and controls are saved in property description blocks, as illustrated in Figure 7-9.

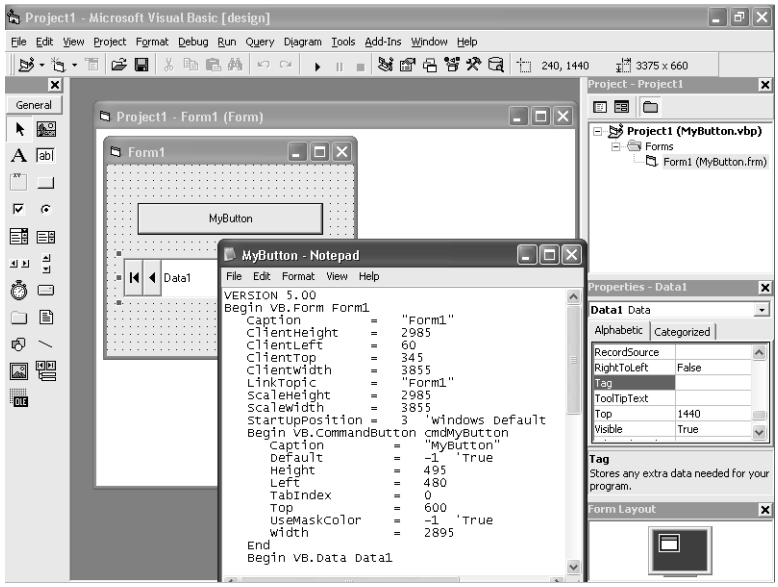


Figure 7-9 Visual Basic 6 .frm file in text view.

Windows Forms stores layout and property settings differently. All settings are saved as part of your code in a hidden block called Windows Form Designer Generated Code. The design-time settings for a Windows form can be found in the *InitializeComponent* subroutine within this hidden block.

The Upgrade Wizard maps your Visual Basic 6 design-time settings to generated code in *InitializeComponent*. For example, suppose you have a Visual Basic form containing a button named MyButton, as shown in Figure 7-10. (You can find the MyButton.vbp project on the companion CD.)

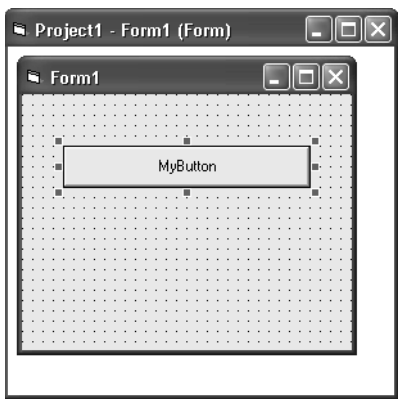


Figure 7-10 Visual Basic 6 form with a button on it.

The Visual Basic .frm file for the form will contain this description:

```
Begin VB.Form Form1
   Caption         =   "Form1"
   ClientHeight    =   2085
   ClientWidth     =   3750
   ClientLeft      =   60
   ClientTop       =   345
   Begin VB.CommandButton cmdMyButton
      Caption         =   "MyButton"
      TabIndex        =   0
      Default         =   -1  'True
      Left            =   1080
      Top             =   720
      Width           =   1455
      Height          =   495
   End
End
```

If you upgrade the preceding Visual Basic .frm, the generated Visual Basic .NET code will be

```
Private Sub InitializeComponent()
   Me.cmdMyButton = New System.Windows.Forms.Button

   Me.Text = "Form1"
   Me.ClientSize = New System.Drawing.Size(250, 139)
   Me.Location = New System.Drawing.Point(4, 23)

   Me.AcceptButton = Me.cmdMyButton

   Me.cmdMyButton.Text = "MyButton"
   Me.cmdMyButton.TabIndex = 0
   Me.AcceptButton = Me.cmdMyButton
   Me.cmdMyButton.Size = New System.Drawing.Size(97, 33)
   Me.cmdMyButton.Location = New System.Drawing.Point(72, 48)
   Me.cmdMyButton.Name = "cmdMyButton"
   Me.Controls.Add(cmdMyButton)
End Sub
```

Note We've removed superfluous settings from the .frm and *InitializeComponent* listings to focus attention on how the design-time settings of a form and control are generally upgraded. We've selected properties that demonstrate the more complex mappings that the Upgrade Wizard supports. Most other settings are one-to-one mappings from (for example) Property X to Property X.

Table 7-3 shows how the original .frm description and the generated Visual Basic .NET code match up line for line.

Table 7-3 Mapping Between Visual Basic 6 .frm Settings and Visual Basic .NET Code

Visual Basic 6 .frm Setting	Visual Basic .NET Code	Upgrade Notes
Caption = "Form1"	Me.Text = "Form1"	*Caption* property is upgraded to *Text* property in Windows.-Forms. Generally you'll see *Caption* properties mapped to *Text* properties in your code.
ClientHeight = 2085 ClientWidth = 3750	Me.ClientSize = New System.-Drawing.Size(250, 139)	Form *Height* and *Width* settings are mapped to the Visual Basic .NET form's *ClientSize* property. Sizes are automatically converted from twips to pixels.
ClientLeft = 60 ClientTop = 345	Me.Location = New System.Drawing.-Point(4, 23)	*Left* and *Top* settings are mapped to the *Location* property. Positions are automatically converted from twips to pixels.
Begin VB.Command-Button cmdMyButton	Me.cmdMyButton = New System.Windows.Forms.-Button	In Visual Basic 6, the run time takes care of creating the form and controls spelled out in the .frm file. In Visual Basic .NET, declaration and allocation of each control is explicit.
Caption = "MyButton"	Me.cmdMyButton.-Text = "MyButton"	*Caption* is upgraded to *Text*.
Default = -1 "True"	Me.AcceptButton = Me.cmdMyButton	*Default* and *Cancel* are no longer properties of individual buttons in Visual Basic .NET. Instead, you set the *AcceptButton* and *CancelButton* properties of the form to point to the respective *Default* and *Cancel* buttons. In this case the *Default* design-time property setting is automatically upgraded to the *AcceptButton* setting on the current form.

After the form is upgraded, it will look nearly identical to its Visual Basic 6 counterpart, as shown in Figure 7-11.

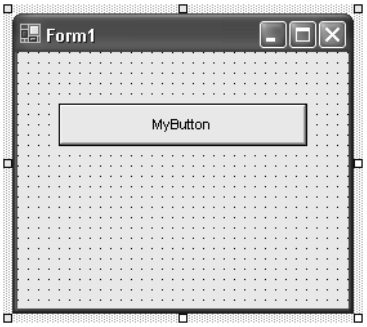
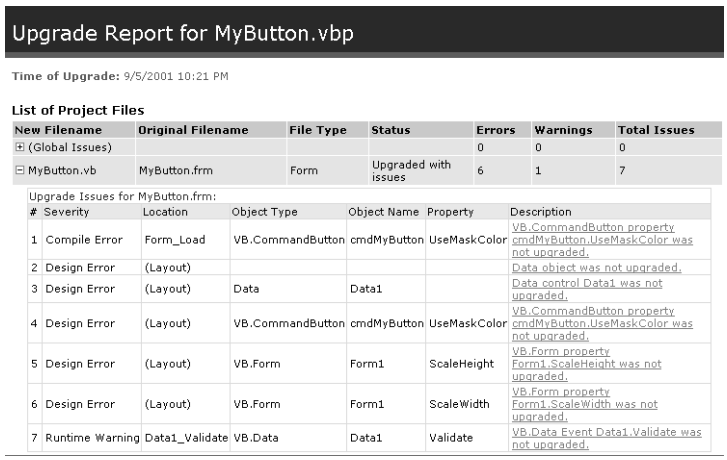


Figure 7-11 Upgraded Visual Basic .NET form.

Let's take a look at some cases in which the design-time settings don't upgrade. We'll place a Data control on the Visual Basic 6 form with the MyButton control. We'll also set the *UseMaskColor* property of the MyButton control to *True*, resave the Visual Basic 6 application, and upgrade it again. The first issue is that the Data control (as listed in Table 7-2 on page 128) can't be upgraded. A red label on the form represents the Data control. Red labels on your form indicate that the form didn't fully upgrade.

An interesting characteristic of this upgraded project is that if you build it, you won't get any compiler errors. How can this be true if there is no property equivalent to the CommandButton's *UseMaskColor* property? If the Upgrade Wizard emitted the design-time property setting in the *InitializeComponent* section, it would lead to a compiler error. Any compiler errors that occur within *InitializeComponent* will prevent you from viewing the design-time form. Since it is important that you be able to view your form after upgrade, the wizard intentionally omits any design-time properties that don't map. Instead, it logs an error to the upgrade report.

If you open the _UpgradeReport.htm file included with your upgraded project, you'll see a summary of the properties that weren't upgraded for each control. In this example, you'd see an entry for the CommandButton's *UseMaskColor* property. Figure 7-12 shows the upgrade report that is produced. The report that documents the issue provides a link to Help, and, where possible, it offers suggestions on how to work around or solve the issue.



Upgrade Report for MyButton.vbp

Time of Upgrade: 9/5/2001 10:21 PM

List of Project Files

New Filename	Original Filename	File Type	Status	Errors	Warnings	Total Issues
⊞ (Global Issues)				0	0	0
⊟ MyButton.vb	MyButton.frm	Form	Upgraded with issues	6	1	7

Upgrade Issues for MyButton.frm:

#	Severity	Location	Object Type	Object Name	Property	Description
1	Compile Error	Form_Load	VB.CommandButton	cmdMyButton	UseMaskColor	VB.CommandButton property cmdMyButton.UseMaskColor was not upgraded.
2	Design Error	(Layout)				Data object was not upgraded.
3	Design Error	(Layout)	Data	Data1		Data control Data1 was not upgraded.
4	Design Error	(Layout)	VB.CommandButton	cmdMyButton	UseMaskColor	VB.CommandButton property cmdMyButton.UseMaskColor was not upgraded.
5	Design Error	(Layout)	VB.Form	Form1	ScaleHeight	VB.Form property Form1.ScaleHeight was not upgraded.
6	Design Error	(Layout)	VB.Form	Form1	ScaleWidth	VB.Form property Form1.ScaleWidth was not upgraded.
7	Runtime Warning	Data1_Validate	VB.Data	Data1	Validate	VB.Data Event Data1.Validate was not upgraded.

Figure 7-12 Visual Basic .NET upgrade report.

Control Arrays

Control arrays are a long-standing Visual Basic feature. A control array allows several controls to share the same events and initial property settings, allowing you to apply the same behavior to the controls in the control array. You can also create control arrays in Visual Basic .NET. However, the way you do so is very different from the way you create them in Visual Basic 6. Think of a control array in Visual Basic .NET as a standard array containing controls. To create a control array in Visual Basic .NET, you need to write code to declare an array of the particular control type, and then add the controls to the array. You write additional code to hook the same set of events to each member of the control array. If you're dealing with a dozen or more control array elements on your form, writing all that code is a time-consuming process.

The Visual Basic .NET compatibility library introduced earlier in this chapter includes support for control arrays. The compatibility library contains an array class for each control type. For example, the control array class for Button is called *ButtonArray*. The control array class includes methods such as *Item* and *Load* that allow you to access control array elements or add new members to the control array. In addition the control array class includes all of the events for the underlying control. The events are shared for all elements of the Visual Basic .NET control array, as you would expect based on how Visual Basic 6 control arrays behave.

The Upgrade Wizard upgrades your Visual Basic 6 control arrays to target control array classes defined in the Visual Basic .NET compatibility library. Control arrays and all associated code are upgraded as is, with minor syntactical changes in some cases, as illustrated in the following example.

Consider a Visual Basic 6 form that has two command buttons in a control array called *Command1*. The following Visual Basic 6 code loads *Command1(2)* as a new control array element:

```
Private Sub Command1_Click(Index As Integer)
   MsgBox "Command1 (" & Index & ") clicked"
End Sub

Private Sub Form_Load()
   Load Command1(2)
   Command1(2).Left = Command1(1).Left + _
      Command1(1).Width
   Command1(2).Visible = True
End Sub
```

Here's the Visual Basic .NET code after the Upgrade Wizard has upgraded it:

```
Private Sub Command1_Click(ByVal eventSender As System.Object, _
ByVal eventArgs As System.EventArgs) _
Handles Command1.Click
   Dim Index As Short = Command1.GetIndex(eventSender)
   MsgBox("Command1 (" & Index & ") clicked")
End Sub

Private Sub Form1_Load(ByVal eventSender As System.Object, _
ByVal eventArgs As System.EventArgs) _
Handles MyBase.Load
   Command1.Load(2)
   Command1(2).Left = _
      VB6.TwipsToPixelsX(VB6.PixelsToTwipsX(Command1(1).Left) + _
      VB6.PixelsToTwipsX(Command1(1).Width))

   Command1(2).Visible = True
End Sub
```

Focus your attention on the usage of *Command1*, and ignore the pixel-to-twips conversions; you'll see that the code you need to use control arrays is nearly identical, with a couple of notable differences:

- The *Load* statement becomes a *Load* method on the base control array element *Command1*—for example, *Command1.Load(2)*.
- The control array index is not passed directly as a parameter to the *Command1_Click* event. The event parameter is instead obtained from the base control array object *Command1* by calling *GetIndex* and passing *GetIndex* the source of the event. The event source will be the member of the control array that you clicked.

Despite these differences, the good news is that Visual Basic .NET and the Upgrade Wizard both support control arrays.

ActiveX Controls and ActiveX References

The general approach to upgrading ActiveX controls and references is to reuse the exact same controls and references in your upgraded Visual Basic .NET application. Although in theory your ActiveX components should work the same as they did prior to the upgrade, you might encounter some issues. These issues stem from differences between the Visual Basic 6 and Visual Basic .NET ActiveX control hosting environments. Issues can also stem from the fact that .NET supports ActiveX components by communicating with them through a COM interop layer. (These issues are discussed in Chapter 9 and Chapter 13.) This section focuses on some of the exceptions to the rule that ActiveX controls and references are upgraded to use the same component. In some cases, an ActiveX control or reference is upgraded to use a native .NET component replacement.

ADO Data Control

You can place the ActiveX ADO Data control (ADODC) shipping with Visual Basic 6 on a Windows form, but you can't use it to bind to Windows Forms controls. For example, you can't place an ADODC control on a Windows form and hook a TextBox up to it because the ADODC control has no way to communicate the bound data to the Windows Forms TextBox control. The ADODC control can communicate only with other ActiveX controls, not with .NET Windows Forms controls.

To resolve this issue, the Visual Basic .NET compatibility library contains a .NET version of the ADODC control. You can use the ADODC .NET control to bind data to Windows Forms controls and other ActiveX controls. The Upgrade Wizard automatically upgrades any instances of ADODC ActiveX controls to ADODC .NET controls. The ADODC .NET control was designed to be a compatible replacement for the ADODC ActiveX control.

How does this help you? If you use the ADODC ActiveX control in your Visual Basic 6 applications, you should expect your ADO data-bound applications to continue working after you upgrade to Visual Basic .NET.

Microsoft Data Binding Collection

The Visual Basic .NET compatibility library provides an MSBind .NET class as a compatible replacement for the ActiveX MSBind object found in Visual Basic 6.

Visual Basic 6 allows you to link a control such as a TextBox with the ADODC control by setting the *DataSource* property of the TextBox control. Although the *DataSource* property appears on the TextBox control, the property is added and managed by Visual Basic and not by the control itself. In fact the control does not even know that it has a *DataSource* property. When you set the *DataSource* property, Visual Basic creates an instance of *MSBind* and associates the TextBox with a field in a database. The database field is specified by the *DataField* property, which is also added to the TextBox control and managed by Visual Basic.

The problem is that Windows Forms doesn't provide any support for the *DataSource* and *DataField* properties. In fact, when you place a TextBox (or any other control) on a Windows form, you won't find a *DataSource* or *DataField* property. How do you get data binding to work in Visual Basic .NET? You use an instance of a binding collection (MSBind).

You can add items to the binding collection to associate a property on a control with a field in a database. To bind a control's property to a field in a database, you add the control and the property where you want the database data to be stored to the MSBind collection. If you want to implement custom data binding or binding to nontraditional properties on a control, use MSBind in your Visual Basic 6 application.

Because there is no *DataSource* or *DataField* property, the Upgrade Wizard translates the binding information stored in the *DataSource* and *DataField* properties to binding information set into an MSBind .NET binding collection. Suppose you're upgrading an application with a TextBox (Text1) bound to an ADODC (Adodc1) control. The TextBox displays the CompanyName contained in the Customers table. The Upgrade Wizard automatically generates the following code to hook up data binding:

```
Private ADOBind_Adodc1 As VB6.MBindingCollection

Public Sub VB6_AddADODataBinding()
   ADOBind_Adodc1 = New VB6.MBindingCollection()
   ADOBind_Adodc1.DataSource = CType(Adodc1, msdatasrc.DataSource)
   ADOBind_Adodc1.Add(Text1, "Text", "CompanyName", Nothing, "Text1")
   ADOBind_Adodc1.UpdateMode = _
      VB6.UpdateMode.vbUpdateWhenPropertyChanges
   ADOBind_Adodc1.UpdateControls()
End Sub

Public Sub VB6_RemoveADODataBinding()
   ADOBind_Adodc1.Clear()
   ADOBind_Adodc1.Dispose()
   ADOBind_Adodc1 = Nothing
End Sub
```

> **Note** In Visual Basic 6, the binding collection is called BindingCollection. In Visual Basic .NET, the binding collection is called MBindingCollection. The *M* stands for managed, meaning the managed code version of the BindingCollection.

VB6_AddADODataBinding is called from *Sub New* so that the data bindings are set up as soon as the form is created. VB6_RemoveADODataBinding is called when the form is disposed. Data bindings are removed when the form is destroyed.

In general, if you have an ADODC-based data-bound application in which all the data binding is set up at design time, the Upgrade Wizard takes care of generating the necessary Visual Basic .NET code to make it work. Because there are no properties you can set in the Property Browser, updating the application after upgrade requires considerably more work. You'll need to update VB6_AddADODataBinding to add or remove bound controls and associated database fields.

SSTab Control

The Upgrade Wizard treats the SSTab control as a special case because the SSTab ActiveX control requires a special interface called *ISimpleFrame*. This interface enables the control to contain other controls. Without this interface, you can't place other controls within the SSTab. If you've used the SSTab control, you know that it would be pretty useless if you couldn't place other controls inside it. The control reaches the height of uselessness when you place it on a Windows form. Windows Forms doesn't support the *ISimpleFrame* interface, so you can't place any controls on the tabs within the SSTab.

The Upgrade Wizard recognizes this situation and rectifies it by automatically upgrading your instances of SSTab controls to instances of Windows Forms TabControls. The Windows Forms TabControl is similar—but not identical—in behavior to the SSTab control. You'll find that after upgrade, the TabControl has a similar appearance to the SSTab control. All the controls contained in the SSTab control will be found on their respective tabs in the Windows Forms TabControl. However, you'll notice differences when you write code to operate the TabControl. Most of these differences stem from the TabControl having a TabPage collection object. You need to manually update the code that sets array properties, such as TabCaption, to instead set the caption on the specific tab within the TabPage collection.

UpDown Control

The UpDown control uses internal interfaces supported by Visual Basic 6 to buddy up with another control. These interfaces aren't supported in Windows Forms. Furthermore, because the control is an ActiveX component, it can't link to a non-ActiveX component such as a Windows Forms Button control. If you place an ActiveX UpDown control on a Windows form, you should set the *BuddyControl* property to another control on the form, and then run the application. The UpDown control doesn't update the buddy control.

The Upgrade Wizard won't upgrade UpDown controls found on a form. Instead, you must replace each UpDown control with the Windows Forms NumericUpDown control.

Visual Basic Code

The Visual Basic .NET language is largely a superset of the Visual Basic 6 language. You use the same keywords, conditional statements, and expressions that you use when writing a Visual Basic 6 application. The Upgrade Wizard will upgrade most of your code that relates to the core Visual Basic language without making any changes to it. Chapter 11 covers issues related to upgrading code in more detail. This section focuses on how the Upgrade Wizard upgrades your Visual Basic 6 code.

General Types of Code Conversions

The Upgrade Wizard upgrades your Visual Basic 6 language statements using a variety of means, which are explained in the following sections.

Mapping to native language statement with the same name Most Visual Basic core language statements, such as If…Then, For…Next, Select Case, And, Or, Do…Loop, On Error GoTo, and MsgBox fall into this category. The code you see in Visual Basic 6 is the code you get in Visual Basic .NET.

Mapping to native language statement with a different name The functions IsEmpty, IsNothing, and IsObject fall into this category. Each of these functions operates on Variants, which aren't supported in Visual Basic .NET. These functions are upgraded to functions that operate on *Object* types. The following Visual Basic 6 code:

```
Dim b As Boolean
Dim v As Variant

b = IsEmpty(v)
b = IsObject(v)
b = IsMissing(v)
b = IsNull(v)
```

upgrades to the following Visual Basic .NET code:

```
Dim b As Boolean
Dim v As Object
'UPGRADE_WARNING: IsEmpty was upgraded to IsNothing and has
'a new behavior.
b = IsNothing(v)
'UPGRADE_WARNING: IsObject has a new behavior.
b = IsReference(v)
```

```
'UPGRADE_NOTE: IsMissing() was changed to IsNothing()
b = IsNothing(v)
'UPGRADE_WARNING: Use of Null/IsNull() detected.
b = IsDbNull(v)
```

Mapping with coercion Visual Basic .NET disallows direct assignments of incompatible types to each other without an explicit conversion. For example, you can't directly assign an integer a *String* value or a *Date* to a *Double*. You need to do an explicit conversion using a function such as *CInt* or *CShort*. Some types contain conversion methods, such as *ToString* or *ToOADate*. As an example of how conversion functions are added after upgrade, consider the following Visual Basic 6 code:

```
Dim i As Integer
Dim TodaysDate As Date
Dim d As Double
i = "7"
TodaysDate = Now
d = TodaysDate
```

This code upgrades to the following Visual Basic .NET code:

```
Dim i As Short
Dim TodaysDate As Date
Dim d As Double
i = CShort("7")
TodaysDate = Now
d = TodaysDate.ToOADate   'Converts to an OLE Automation compatible
                          'Date value
```

> **Note** If you need to convert between types in your Visual Basic .NET code, look at the methods implemented by the .NET *System.Convert* class.

Mapping to an equivalent .NET Framework function Math functions such as *Abs*, *Atn*, *Cos*, and *Sqrt* fall into this category. Certain properties on the *App* object, such as *PrevInstance*, upgrade to .NET equivalents as well. For example, take a look at the following Visual Basic 6 code:

```
Dim i As Integer
i = Abs(-1)
```

This code upgrades to the following Visual Basic .NET code, using the equivalent function from the *System.Math* class:

```
Dim i As Short
i = System.Math.Abs(-1)
```

Mapping to a function provided by the Visual Basic .NET compatibility library

The *LoadResString*, *LoadResData*, *LoadResPicture*, *SendKeys*, and some *App* object methods fall into this category. The following Visual Basic 6 code:

```
Dim strPath As String
strPath = App.Path
```

upgrades to

```
Dim strPath As String
strPath = VB6.GetPath
```

No mapping available This category includes cases that can't be automatically upgraded. Instead, the wizard generates an upgrade issue and possibly an UPGRADE_TODO item. The binary string functions, such as *AscB*, *LeftB*, *MidB,* and *RightB* fall into this category. For example, the following Visual Basic 6 code:

```
Dim iCharCode As Integer
Dim BinaryString() As Byte
BinaryString = StrConv("This is my binary string", vbFromUnicode)
iCharCode = AscB(BinaryString)
```

upgrades to the following Visual Basic .NET code:

```
Dim iCharCode As Short
Dim BinaryString() As Byte
'UPGRADE_ISSUE: Constant vbFromUnicode was not upgraded.
'UPGRADE_TODO: Code was upgraded to use
'System.Text.UnicodeEncoding.Unicode.GetBytes() which may not have the
'same behavior.
BinaryString = System.Text.UnicodeEncoding.Unicode.GetBytes(StrConv( _
   "This is my binary string", vbFromUnicode))
'UPGRADE_ISSUE: AscB function is not supported.
iCharCode = AscB(BinaryString)
```

Several UPGRADE_ messages are emitted in this case. The UPGRADE_ISSUE messages relate to byte-related support not being available. The UPGRADE_TODO message warns you that the .NET conversion function used might not have the behavior you desire.

Option Explicit On Included Automatically

When you use the Option Explicit statement in Visual Basic 6, the compiler forces you to declare all variables with a Dim, Private, Public, or Static statement (for example). If you're not using the Option Explicit statement in Visual Basic 6, the Upgrade Wizard includes the Option Explicit On statement in all forms and modules and takes care of declaring any undeclared variables for you. If you've been planning to go through and explicitly declare all of your variables, or you've avoided doing it because of time constraints, your wait has been rewarded. The Upgrade Wizard takes care of this for you.

DefType Is Upgraded

If you use the DefType statement to determine the variable type for unqualified variable declarations, your DefType declarations will be upgraded to the appropriate type. For example, the following Visual Basic 6 code:

```
DefStr A-Z      'All unqualified variable declarations will be Strings
Sub Main
   Dim s
   s = "I'm a string"
End Sub
```

upgrades to the following Visual Basic .NET code:

```
Sub Main
   Dim s As String
   s = "I'm a String"
End Sub
```

Data Types Are Upgraded to Fit

The storage size for Visual Basic intrinsic types such as *Integer* and *Long* has changed in Visual Basic .NET. In Visual Basic 6 an *Integer* is 16 bits and a *Long* is 32 bits. In Visual Basic .NET an *Integer* is 32 bits and a *Long* is 64 bits. Most of the time it doesn't matter whether you use an *Integer* instead of a *Long*. If you have a loop index that goes from 0 to 10, you can nitpick over the performance implications of using one size or another, but generally it makes no difference. The program works the same way.

Sometimes, however, the size difference matters. For example, if you're calling Windows API functions that require a 32-bit integer argument, you had better use a *Declare* statement that declares the argument as 32 bits.

To keep your application running smoothly after upgrade, the Upgrade Wizard declares all of your numeric types to use the correct Visual Basic .NET equivalent based on size. This means that an *Integer* variable is upgraded to *Short*. A variable of type *Long* is upgraded to type *Integer*. In the case of the *Variant* type, the Upgrade Wizard maps to the closest equivalent type found in

Visual Basic .NET: *Object*. Table 7-4 gives a mapping of types between Visual Basic 6 and Visual Basic .NET. The table provides mappings where the name of the type is different between Visual Basic 6 and Visual Basic .NET. All other types, such as *Byte*, *Single*, *Double*, and *String,* map as is.

Table 7-4 Mapping of Types Between Visual Basic 6 and Visual Basic .NET

Visual Basic 6 Type	Upgrades to Visual Basic .NET Type
Integer	Short
Long	Integer
Variant	Object
Currency	Decimal

ByRef Added to Unqualified Parameters

The default calling convention in Visual Basic 6 is to pass all unqualified parameters as *ByRef*. Visual Basic .NET, on the other hand, requires you to qualify all parameters as *ByVal* or *ByRef*. Because the Visual Basic 6 default calling convention is *ByRef*, the Upgrade Wizard upgrades and qualifies all unqualified parameters with the *ByRef* keyword.

For example, the following Visual Basic 6 code:

```
Sub Test(MyParam As Long)
   MyParam = 7
End Sub
```

upgrades to the following Visual Basic .NET code:

```
Sub Test(ByRef MyParam As Integer)
   MyParam = 7
End Sub
```

Code-Behind Forms

Code-behind forms refers to code that makes reference to a form, control, property, method, or event. Although this is normally code you write behind a form, you can also reference controls from other project items, such as a class module or .bas module. We will limit this discussion to code that references a property, method, or event for any form or control in the project.

Code that references a form or control property is upgraded in a similar fashion to a design-time property setting, as discussed earlier in the section "Layout and Design-Time Property Settings." The main difference is that for properties you refer to in code, the Upgrade Wizard will leave any code that cannot be upgraded as is. In many cases this will lead to a compiler error. In

addition, a comment that begins with UPGRADE_ (such as UPGRADE_ISSUE) will be inserted on the line immediately above the nontranslated statement. For example, let's say you attempt to upgrade the following code:

```
cmdMyButton.UseMaskColor = True
```

The upgraded code will appear as follows:

```
'UPGRADE_ISSUE: CommandButton property cmdMyButton.UseMaskColor
'was not_upgraded. Click for more:
''ms-help://MS.VSCC/commoner/redir/redirect.htm?keyword="vbup2064"'
cmdMyButton.UseMaskColor = True
```

The Upgrade Wizard leaves the code in as is—without omitting it or commenting it out—because doing so makes the issue stand out like a sore thumb. You are forced to recognize and deal with the problem before you can run your upgraded application. The philosophy here is that it's best to get all the bad news up front. It's generally much easier to resolve a problem as soon as you discover it, rather than finding it down the road after you've made a bunch of code changes. If the Upgrade Wizard were simply to emit the UPGRADE_ISSUE command and avoid the compiler error by commenting out the offending line, you might never pay attention to the issue. If a problem related to the issue surfaced months later, you might end up spending a significant amount of time tracking down the line of code that the Upgrade Wizard conveniently commented out for you.

Global Objects

The Upgrade Wizard provides minimal support for upgrading global objects such as *App*, *Printer*, *Clipboard*, and *Screen*. In fact, the wizard upgrades some methods for the *App* and *Screen* objects, and that's all. You will need to manually upgrade the code related to the other global objects. For example, to upgrade your *Printer* object–related code, use the .NET *System.Drawing.Printing.PrintDocument* class. Chapter 12 contains more information on upgrading global objects.

Class Modules and User Controls

Each class module and user control upgrades to an equivalent class module and user control in Visual Basic .NET. Table 7-5 lists the upgrades for class attributes of class modules and user controls.

Table 7-5 Class Attribute Mappings

Class Attribute	Upgrades To
Private	Friend
Public noncreatable	Public
Single use	Public
Global single use	Public
Multiuse	Public
Global multiuse	Public

Persistable Classes: *ReadProperties*, *WriteProperties*

The *ReadProperties* and *WriteProperties* events for persistable classes and user controls are not upgraded because the .NET Framework doesn't support the same notion of data persistence that ActiveX supports. The .NET Framework requires that clients of a .NET server persist the properties of the server by generating code to set server properties to initial property values. Clients persist ActiveX servers, on the other hand, by passing either a PropertyBag or a binary stream to the ActiveX server, requesting that the server save the state of its properties to the PropertyBag or stream. When the server is a Visual Basic–authored ActiveX component, the client will invoke the *ReadProperties* and *WriteProperties* events of the component. After the server has been upgraded to a .NET server, the client directly obtains the property values for the ActiveX component without invoking events such as *ReadProperties* and *WriteProperties*.

The bottom line is that you don't need *ReadProperties* and *WriteProperties* to create a persistable Visual Basic .NET server class. You can remove this code after upgrade.

DataSource Classes

For a Visual Basic 6 class in which the *DataSourceBehavior* property is set to vbDataSource, the class is upgraded to implement the *DataSource* interface. Your code contained within the *GetDataMember* event is upgraded to the GetDataMember method of the implemented *DataSource* interface. You can test your upgraded class by assigning an instance of it to the DataSource property of a control such as the Visual Basic 6 DataGrid. The DataGrid should display the data returned by your upgraded Visual Basic .NET *DataSource* class.

Objects for Accessing Data

If you've written a Visual Basic 6 application based on ADO, it will upgrade as is, including code related to ADO data binding. In addition, when upgrading your ADO-based application, the Upgrade Wizard will promote all references to

ADO versions 2.6 or earlier to ADO version 2.7. If you based your application on DAO or RDO, the DAO- and RDO-related elements of your application will also upgrade as is as long as you are not using DAO or RDO data binding.

Data Binding

To support data binding, any Microsoft ADODC ActiveX controls you use will automatically be upgraded to use a .NET ADODC control. The .NET version is designed to be compatible with the ActiveX version.

Visual Basic 6 provides internal support for DAO and RDO data binding. There are neither internal supports nor public classes in Visual Basic .NET to support DAO or RDO data binding. If your application requires data binding, you should manually replace your DAO- or RDO-related elements with ADO-related elements. You can make the changes to your Visual Basic 6 application, test the changes, then upgrade your ADO-based Visual Basic 6 application to Visual Basic .NET.

DataEnvironment

If you use a DataEnvironment designer in your Visual Basic 6 application, the wizard will create a Visual Basic .NET *DataEnvironment* class—contained in its own .vb file—and add it to your upgraded Visual Basic .NET project. The *DataEnvironment* class will contain all settings and objects found in the original Visual Basic 6 DataEnvironment. Any code that references the DataEnvironment will continue to work. You should be able to get your upgraded application to work with the generated *DataEnvironment* class at run time.

Visual Basic .NET doesn't provide a way for you to edit *DataEnvironment* properties at design time. To change property settings, you need to edit the code contained in the generated *DataEnvironment* class file.

Designers

Visual Basic .NET doesn't include designers such as the DHTML, WebClass, DataEnvironment, and DataReport. In fact, none of the designers are included with Visual Basic .NET, nor is there any support to load a designer in Visual Studio .NET.

In some cases the code and design-time settings related to a designer are upgraded to work in the upgraded Visual Basic .NET application. For example, code associated with your WebClass is upgraded to Web Forms; DataEnvironment-related settings and code are upgraded to a generated *DataEnvironment* class. The CrystalReports package in Visual Studio .NET supports reading reports stored in DataReport designer format.

Conclusion

In terms of upgrade potential, is your application in or out? As discussed in this chapter, the situation really depends on the type of Visual Basic application you've written. If your application drives the Visual Basic 6 extensibility model, uses an ActiveX designer such as the DHTML page designer, or relies on Dynamic Data Exchange (DDE), it's in the out category. You'll need to do a significant amount of rewriting to make your application work in the .NET environment. In some cases, it might make more sense to leave your application running in Visual Basic 6 and interoperate with it from Visual Basic .NET.

If, on the other hand, your application is a more typical three-tier application with a presentation layer that communicates with a server component, it's most definitely in the in category. You'll find that the Upgrade Wizard does a good job of upgrading applications that fall into this category. The Upgrade Wizard will strive to upgrade your application in the least intrusive way possible that will allow you to get it up and running quickly. Using the Upgrade Wizard, you will be able to get your Visual Basic 6 application up and running in Visual Basic .NET more quickly than by porting your Visual Basic 6 application by hand.