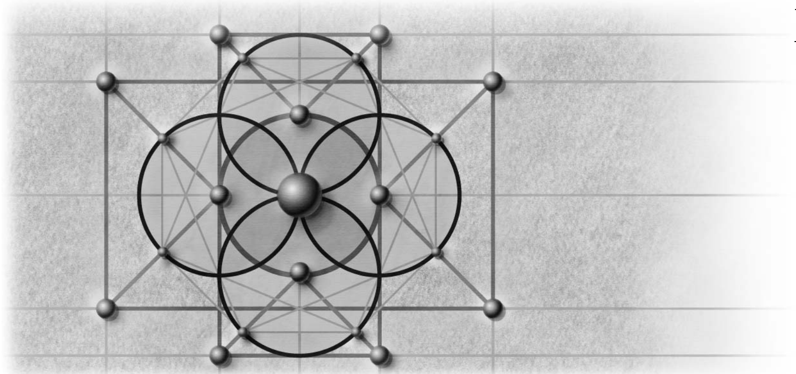

12

Resolving Issues with Forms

In 1988, Alan Cooper demonstrated a prototype called Ruby to Bill Gates. Ruby provided a form designer that allowed you to drag and drop controls, then known as gizmos, to quickly and easily create composite forms—such as dialog boxes, entry forms, and report forms. Microsoft took Cooper's Ruby product and combined it with Basic to create Microsoft Visual Basic 1. Microsoft has since shipped a version of Ruby with every version of Visual Basic, versions 1 through 6. With every version, that is, until Visual Basic .NET.

Visual Basic .NET provides a new forms package called Windows Forms. Although the Windows Forms package was designed using the same basic principle as Ruby—it is a form designer that allows you to drag and drop controls and set properties—it was never meant to be an extension of, nor to be compatible with, Ruby. Therefore, there are fundamental differences between the two forms packages that affect the way you create Visual Basic applications.

This chapter focuses on some of the fundamental differences between the Ruby and Windows Forms packages. Specifically, it discusses issues that the Upgrade Wizard does not handle for you. Before we get into the differences, however, let's look at what Windows Forms and Ruby have in common.

Similarities in Form Structure

When you create a new project in Visual Basic .NET, you will find yourself at home in the environment. The way you create and design forms is the same in Visual Basic .NET as it is in Visual Basic 6. Although the names of some of the properties, methods, and events may have changed, you should be able to

C1261587x.fm Page 266 Thursday, November 15, 2001 3:51 PM

quickly find the one you need to use. In fact, you will find that you can create the exact same form in both Visual Basic 6 and Visual Basic .NET, using largely the same actions, the same controls, and the same property settings.

The Upgrade Wizard can re-create your Visual Basic 6 forms in Visual Basic .NET. This is possible because the essential pieces of Visual Basic .NET are, for the most part, similar to those in Visual Basic 6: equivalent controls and equivalent properties, methods, and events. Figures 12-1 and 12-2 demonstrate a Visual Basic 6 form before and after being upgraded to Visual Basic .NET.

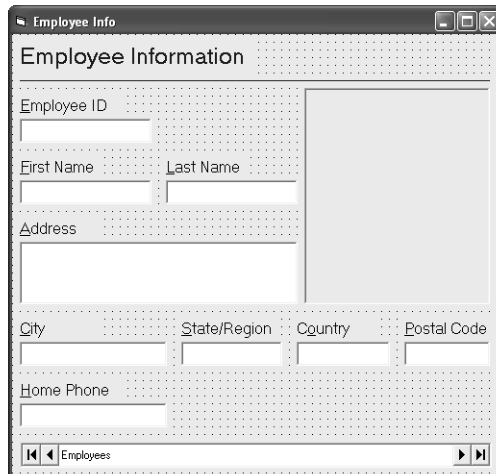

Figure 12-1 Visual Basic 6 form in design view.

Figure 12-2 Upgraded Visual Basic .NET form in design view.

C1261587x.fm Page 267 Thursday, November 15, 2001 3:51 PM

As for the differences between forms in Visual Basic 6 and Visual Basic .NET, most of them are subtle, many related to the renaming or restructuring of components. It is these subtle differences, however, that can give you the most grief. We will spend the remainder of the chapter talking about them.

General Issues

Whether you are editing an upgraded application or writing a new Visual Basic .NET application from scratch, you will encounter certain general issues. These issues apply across forms, controls, and components. For example, you need to be aware of some basic differences between the Visual Basic 6 and Visual Basic .NET property, method, and event (PME) models. Understanding these fundamental differences should give you a good base to start from when you are trying to understand issues you encounter involving a specific form or control.

Differences in Properties, Methods, and Events

The component models for Visual Basic 6 and Visual Basic .NET are similar in the sense that objects are composed of the same pieces: properties, methods, and events. You use properties, methods, and events in exactly the same way in Visual Basic .NET as you do in Visual Basic 6. The main difference is that the specific properties, methods, and events that make up any given object do not always match up. Let's explore some of the general PME model differences.

Some Properties, Methods, and Events Have Been Renamed

One difference between the Visual Basic 6 and Visual Basic .NET PME models is that in some cases they use different names to refer to the same thing. For example, Visual Basic .NET does not contain a *Caption* property. Instead, every .NET control and component uses the *Text* property. In many other cases, properties, methods, or events have been renamed. For example, *SetFocus* in Visual Basic 6 is *Focus* in Visual Basic .NET, and the Visual Basic 6 *GotFocus* and *LostFocus* events are known as *Enter* and *Leave* in Visual Basic .NET.

If you are upgrading an application, you do not need to be too concerned about property naming differences. The Upgrade Wizard will automatically upgrade your code to use the new names. Your challenge will be to recognize and use the renamed properties when you start to modify your upgraded project or write new code. Appendix A contains a complete mapping of all Visual Basic 6 properties, events, and methods to their counterparts in Visual Basic 6.

Multiple Properties Can Be Mapped into a Single Object Property

In some cases, the Upgrade Wizard will map more than one property to constructor parameters on a object. For example, an equivalent way of setting a form's *Left* and *Top* properties is to use the *Location* property. To set the *Location*

C1261587x.fm Page 268 Thursday, November 15, 2001 3:51 PM

property, you must first create an instance of the *System.Drawing.Point* object and initialize *System.Drawing.Point* with the left and top position of the component. For example, you can replace the following Visual Basic 6 code:

```
Command1.Left = 10
Command1.Top = 20
```

with the following Visual Basic .NET code:

```
Button1.Location = New System.Drawing.Point(10, 20)
```

In a similar manner, the Visual Basic 6 *Width* and *Height* properties can be represented by the *Size* property in Visual Basic .NET.

> **Note** The *Left*, *Top*, *Width*, and *Height* properties are available in Visual Basic .NET. We use *Location* and *Size* as an example because if you look in the code generated by the Windows Forms Designer in the "Windows Form Designer generated code" #Region section of the file, you will see that the designer uses the *Location* and *Size* properties to set the position and size of controls on the form. The Upgrade Wizard will automatically map properties such as *Left* and *Top* to constructor parameters on an object such as *Point*.

Some Properties, Methods, and Events Are Not Supported

In some cases you will find that there is no equivalent Visual Basic .NET counterpart for a particular property, method, or event in Visual Basic 6. This will be the case for one of two reasons: support is provided in a different manner, or the feature is considered out of date.

Support Available in a Different Manner You won't find the Visual Basic 6 properties and methods related to drawing and graphics in Visual Basic .NET. At least, you won't find them supported in the same way. These properties and methods, such as *AutoRedraw, Line*, and *Circle*, do have equivalents in Visual Basic .NET. The equivalent functions are provided in the .NET Framework *System.Drawing.Graphics* class.

In addition, global objects such as *Clipboard* and *Printer* do not have matching objects in Visual Basic .NET. However, you will find a *Clipboard* class and a *PrintDocument* class in the .NET Framework that you can use to accomplish the same task. Table 12-1 provides a list of Visual Basic 6 statements and objects for which equivalent functionality can be found in the .NET Framework.

C1261587x.fm Page 269 Thursday, November 15, 2001 3:51 PM

Table 12-1 User Interface Components and Their Visual Basic .NET Equivalents

Visual Basic 6	Visual Basic .NET
Graphics statements (such as *Line* and *Circle)*	*System.Drawing.Graphics* class
OLE drag and drop	Windows Forms drag-and-drop properties, methods, and events to implement manual drag and drop
Clipboard object	*System.Windows.Forms.Clipboard* class
Printer object	*System.Drawing.Printing.PrintDocument* class

The Visual Basic Upgrade Wizard does not upgrade objects, properties, methods, or events for which there is no equivalent in Visual Basic .NET. It therefore will not upgrade any properties, methods, or events related to the areas listed in Table 12-1. The wizard simply preserves your Visual Basic 6 code related to these areas as is. You need to upgrade your code manually to take advantage of the equivalent functionality provided by Visual Basic .NET or the .NET Framework.

We'll discuss graphics statements in more detail shortly. For information on how to add replacements for the *Clipboard* object, see Chapter 15. OLE drag and drop was discussed in Chapter 10.

Out-of-Date Features Features such as Dynamic Data Exchange (DDE) and Visual Basic drag and drop, not to be confused with OLE drag and drop, are considered to be out of date. Neither Visual Basic .NET nor the .NET Framework supports these features. The Upgrade Wizard will preserve any Visual Basic 6 code you have written related to out-of-date objects, properties, methods, or events as is. You need to either change your code to adopt a more up-to-date technology or implement support for the technology in .NET by calling the Windows API functions that relate to the technology. For example, you can replace your DDE code with COM, or you can implement DDE support in your Visual Basic .NET application by declaring and calling the DDE-related API functions. The next section talks in more detail about ways to move code that uses out-of-date technologies to Visual Basic .NET.

Technology Differences

Since its inception in 1991, the focus of Visual Basic has been to enable Windows technologies for the developer. It allows you to quickly create Windows applications that use data, graphics, COM, and a host of other technologies. At the same time, Visual Basic has emphasized compatibility. This means that Visual Basic support—language statements and object models—for various

C1261587x.fm Page 270 Thursday, November 15, 2001 3:51 PM

technologies has accumulated over the years. Visual Basic 6 supports all technologies, regardless of their relevance in today's world. Objects and statements to support old technology coexist alongside the new: DDE with COM and the data objects DAO and RDO with ADO.

Visual Basic .NET, on the other hand, starts anew, wiping the technology slate clean. Visual Basic .NET and the .NET Framework provide first-class .NET support for the latest technologies, such as the .NET component model and ADO.NET. You can still use some of the existing technologies via COM interop, but you will not find .NET classes representing existing technologies such as ADO and DDE. Visual Basic .NET and the .NET Framework also offer renovated object models for technologies such as graphics and forms.

What does this all mean to you? It means that the Visual Basic code you write to take advantage of a particular technology is different from before. New .NET component models represent these technology areas. In many cases, you will find that you can accomplish the same tasks as before, but you will need to use different components, properties, methods, and events to do so. This section discusses changes to technology areas related to components and forms.

Graphics

Visual Basic 6 supports a number of graphics methods on the form, such as *Line* and *Circle*, and it supports PictureBox and Image controls that allow you to draw lines, circles, and text on a form or control. Visual Basic .NET does not support any methods on the form or PictureBox controls that let you draw. Instead you must respond to the *Paint* event, where you are passed a *System.Drawing.Graphics* object. You can use the *System.Drawing.Graphics* object and related objects to draw exactly as you have drawn before.

The Visual Basic .NET Upgrade Wizard does not automatically upgrade your code to respond to the *Paint* event because the model for drawing in Visual Basic 6 is dramatically different from that in Visual Basic .NET. Visual Basic 6 allows you to call a drawing method from any event or function in your code. Visual Basic .NET, on the other hand, requires any drawing code to be located in the *Paint* event or in subroutines called from the *Paint* event. It is therefore impossible for the wizard to pull calls to drawing methods, scattered about in various parts of your code, and arrange the code so that it will continue to work. Even if the wizard were able to pull off such a feat, you probably wouldn't recognize the resulting code. Therefore, the wizard leaves the calls to the Visual Basic 6 graphics methods in your upgraded Visual Basic .NET code as is. You need to manually port your graphics-related code to use the .NET equivalent graphics functions given in Table 12-2. For examples of manually upgrading Visual Basic 6 graphics statements such as *Line* and *Circle* to Visual Basic .NET, see Chapter 15.

C1261587x.fm Page 271 Thursday, November 15, 2001 3:51 PM

Table 12-2 .NET Framework Equivalents for Visual Basic 6 Graphics Statements

Visual Basic 6	Visual Basic .NET
Circle	*System.Drawing.Graphics.DrawEllipse* method
Cls	*System.Drawing.Graphics.Clear* (color) method
CurrentX	*X* parameter to various graphics methods, such as *System.Drawing.Graphics.DrawString*
CurrentY	*Y* parameter to various graphics methods, such as *System.Drawing.Graphics.DrawString*
DrawMode	Attributes of *System.Drawing.Pen* passed to a *Draw* method on *System.Drawing.Graphics*
DrawStyle	Attributes of *System.Drawing.Pen* passed to a *Draw* method on *System.Drawing.Graphics*
DrawWidth	*Width* property of *System.Drawing.Pen* passed to a *Draw* method on *System.Drawing.Graphics*
FillColor	One of the colors contained in *System.Drawing.Brushes* passed to either *System.Drawing.Graphics* method *FillEllipse* or *FillRectangle*
FillStyle	*HatchBrush* class and *HatchStyle* enumeration in the System.Drawing.Drawing2D namespace
HDC	*System.Drawing.Graphics.GetHdc* method
Image	*System.Drawing.Image* class
Line	*System.Drawing.Graphics.DrawLine* method
PaintPicture	*System.Drawing.Graphics.DrawImage* method
Point	*System.Drawing.Bitmap.GetPixel* method
Print	*System.Drawing.Graphics.DrawString* method
PSet	*System.Drawing.Bitmap.SetPixel* method
TextHeight	*System.Drawing.Graphics.MeasureString* method
TextWidth	*System.Drawing.Graphics.MeasureString* method

Dynamic Data Exchange

Dynamic Data Exchange (DDE) was at one time one of the few ways you could pass data between applications without using a shared file. Visual Basic support for DDE dates back to Visual Basic 1, which included support for DDE on Windows 3. Visual Basic support for DDE, and the underlying Windows support for DDE, has remained largely unchanged since 1990. DDE, however, has since been replaced by COM.

COM allows you to share data and invoke methods between applications in a more efficient and scalable manner. Since most Windows applications that

C1261587x.fm Page 272 Thursday, November 15, 2001 3:51 PM

expose public data and methods do so via COM, DDE is generally not the primary way that you retrieve public data and invoke public methods on a Windows application. You use COM as a means of communicating with most Windows applications.

After upgrading your application to Visual Basic .NET, it is highly recommended that you replace all DDE communication with COM. For example, if you have code that calls *LinkExecute* to invoke a method on a DDE server, look at the documentation or type library for the server and see whether there is a COM equivalent for the same method. If there is, add a COM reference to the Windows application and write Visual Basic code to call the method.

Neither Visual Basic .NET nor the .NET Framework provides support for DDE. If you absolutely must communicate with another application using DDE, you will need to implement DDE in your Visual Basic .NET application in one of two ways: by creating and interoperating with an ActiveX EXE server built in Visual Basic 6 that manages the DDE conversation or by declaring and implementing the DDE-related Windows API functions.

If you want to reuse your Visual Basic 6 DDE code, you can create a Visual Basic 6 ActiveX EXE project and add a public class to exchange the DDE-related information you need with your Visual Basic .NET application. For example, if you want to perform a DDE *LinkExecute* operation in your Visual Basic .NET code, you create a DDE helper class written in Visual Basic 6 as follows:

1. Create a Visual Basic 6 ActiveX EXE project.
2. Add a public method to *Class1* called *LinkExecute* that takes two parameters: *ServerTopic* and *ExecCommand*.
3. Add a form—called Form1—to the ActiveX EXE project.
4. Add a TextBox—called Text1—to the form.
5. Add the following code to the *LinkExecute* method in *Class1*:

```
Dim f As New Form1

f.Text1.LinkMode = 0                'None
f.Text1.LinkTopic = ServerTopic
f.Text1.LinkMode = 2                'Manual
f.Text1.LinkExecute ExecCommand
```

6. Build the ActiveX EXE server as Project1.dll.

To perform the *LinkExecute* operation in Visual Basic .NET, add a COM reference to Project1.dll to your Visual Basic .NET project References list and enter the following code:

```
Dim DDEClientHelper As New Project1.Class1Class()
DDEClientHelper.LinkExecute("MyDDEServer|MyTopic", "MyCommand")
```

C1261587x.fm Page 273 Thursday, November 15, 2001 3:51 PM

If you want to implement DDE in your Visual Basic .NET application by calling the DDE-related API functions, you will, at a minimum, need to declare and call *DdeInitialize* and *DdeUninitialize*. To call *DdeInitialize*, you need to implement *DdeCallbackProc*. However, this is just a start. You will also need to call a number of other DDE-related API functions to establish a connection and send data. To implement functionality to perform *LinkExecute*, for example, requires calling at least eight other DDE-related API functions. This task also requires a deep understanding of Windows messaging architecture and memory management, not to mention a knowledge of how to represent Windows types—such as handles and structures—in Visual Basic code. This undertaking is not for the faint of heart. To learn more about the DDE-related Windows API, search for "Dynamic Data Exchange Management Functions" in the Microsoft Developer Network (MSDN). For more information on issues related to using *Declare* statements in your code, refer to Chapter 11.

Visual Basic Drag and Drop

Visual Basic drag and drop has been available since Visual Basic 1. This feature allows you to drag and drop a control at run time onto a form or onto any other control on a form. The result of the drag-and-drop action depends on the code you write for the *DragDrop* event for a form or control. For example, you could write code for a CommandButton that would check to see whether the control being dropped was a TextBox and, if so, copy the TextBox text to the CommandButton caption. At run time, you drag and drop the TextBox onto the CommandButton, and the button caption changes to the TextBox text.

Although Visual Basic drag and drop is useful in creating a rich user interface, it has some serious limitations. Its functionality is limited to dragging and dropping forms within the same application; you cannot drag and drop between applications. In addition, if you set *DragMode* to automatic, there is no result for the drag-and-drop operation unless you write code to define the drop action. This means that you need to write code in the *DragDrop* event for the form and all controls that are to support drag and drop. Finally, when a control is dragged over a particular form or control, there is no concept of a no-drop icon. You need to write code in the *DragOver* event and set the *DragIcon* property to a no-drop icon for all controls that you do not want to allow to be dropped.

Visual Basic OLE drag and drop was introduced in Visual Basic 5. It provides several advantages over Visual Basic drag and drop. First, OLE drag and drop is a COM standard, meaning that you can drag and drop data between any applications that support OLE drag and drop and the data that you are dragging. In addition, OLE drag and drop is automatic. You can enable it by setting the *OLEDragMode* and *OLEDropMode* properties of the form and controls that you want to have support OLE drag and drop. You are not required to write any code to enable basic drag-and-drop operations. Finally, in OLE drag and drop,

C1261587x.fm Page 274 Thursday, November 15, 2001 3:51 PM

the concept of a no-drop icon is built in. This means that by default the form and controls do not advertise themselves as targets of a drag-and-drop operation. You set the *OLEDropMode* property or write code for the controls that you want to have support OLE drag and drop.

Because Visual Basic drag and drop is considered an outdated feature, the Upgrade Wizard does not upgrade your *DragDrop*-related code. Instead, it preserves the Visual Basic 6 drag-and-drop code as is in the upgraded Visual Basic .NET project. The code will not compile, so you need to either delete it and remove the drag-and-drop functionality or change the code to use OLE drag and drop. For information on how to implement drag and drop in Windows Forms, see Chapter 10.

Issues Involving Forms

Let's start our discussion of forms by looking at the differences between a Visual Basic 6 form and a Visual Basic .NET form. Fortunately, these differences are relatively minor. The basic concept of a form, for example, is the same between Visual Basic 6 and Visual Basic .NET. The form serves as a surface where you can place controls to create a composite form. The way in which you create a form, set properties, and add controls to it using the Visual Basic .NET designer is also essentially the same as in Visual Basic 6. You should find yourself at home when you create and edit Visual Basic .NET forms.

Event Firing Differences

A subtle difference between Visual Basic 6 and Visual Basic .NET is that some events do not fire in the same relative order. For example, in Visual Basic 6 the *Load* event is the first event you expect to fire. This is not true for Visual Basic .NET, however, where the *Resize* event occurs before the *Load* event. Also, depending on the control properties you set at design time, a number of other events can occur before the *Load* event. Table 12-3 lists the events that can occur before the *Form.Load* event.

Table 12-3 Visual Basic .NET Events That Can Occur Before the *Form.Load* Event

Object	Event
Form	Move
Form	Resize
CheckBox	Click
ComboBox	Change
OptionButton	Click
TextBox	TextChanged

C1261587x.fm Page 275 Thursday, November 15, 2001 3:51 PM

Problems generally occur when you write your code in such a way that code in one event relies on values or objects being initialized or set in another event. For example, suppose that you create an element of a control array in the *Form.Load* event and write code to position the control array element in the *Form.Resize* event. If the *Resize* event fires after the *Load* event, everything works as planned. If, on the other hand, the *Resize* event fires before the *Load* event, an error will occur in your *Resize* code. The reason is that you will be attempting to access a control array element that does not yet exist.

To illustrate this problem, consider the following code for a Visual Basic 6 standard EXE application, in which Form1 contains a CommandButton with the *Index* property set to 0.

```
Private Sub Form_Load()
   ' Load a new element of the Command1 control array
   Load Command1(1)
   Command1(1).Visible = True
End Sub

Private Sub Form_Resize()
   ' Position control array element 1 below control array element 0
   Command1(1).Top = Command1(0).Top + Command1(0).Height + 3
End Sub
```

If you use the Upgrade Wizard to upgrade the application to Visual Basic .NET, the upgraded code will be as follows:

```
Private Sub Form1_Load(ByVal eventSender As System.Object, _
                       ByVal eventArgs As System.EventArgs) _
                       Handles MyBase.Load
   ' Load a new element of the Command1 control array
   Command1.Load(1)
   Command1(1).Visible = True
End Sub

'UPGRADE_WARNING: Event Form1.Resize may fire when form is initialized
.
Private Sub Form1_Resize(ByVal eventSender As System.Object, _
                         ByVal eventArgs As System.EventArgs) _
                         Handles MyBase.Resize
   ' Position control array element 1 below control array element 0
   Command1(1).Top = _
      VB6.TwipsToPixelsY(VB6.PixelsToTwipsY(Command1(0).Top) + _
      VB6.PixelsToTwipsY(Command1(0).Height) + 3)
End Sub
```

When you run the upgraded Visual Basic .NET application, an error occurs on the assignment of *Command1(1).Top*. The error is "Control array element 1 doesn't exist." It occurs because the *Resize* event takes place before the *Load* event.

You can fix this problem, and others like it, by using a class-level Boolean variable to keep track of whether the *Load* event has been executed. If it has not executed and is not in the process of executing, you should not execute any code in the current event. In the example just given, you can declare a new form-level variable called *m_Loaded* as follows:

```
Private m_Loaded As Boolean = False
```

In the *Form1_Resize* event procedure, check *m_Loaded* and do not execute any code unless it is *True*:

```
If m_Loaded Then
   ' Position control array element 1 below control array element 0
   Command1(1).Top = _
      VB6.TwipsToPixelsY(VB6.PixelsToTwipsY(Command1(0).Top) + _
      VB6.PixelsToTwipsY(Command1(0).Height) + 3)
End If
```

In the *Form1_Load* event procedure, set *m_Loaded* to *True* as follows:

```
m_Loaded = True

' Load a new element of the Command1 control array
Command1.Load(1)
Command1(1).Visible = True
```

Because the *Resize* event fires after the *Load* event in Visual Basic 6, you need to emulate this behavior in Visual Basic .NET. You do this by calling the *Resize* event before exiting the *Form1.Load* event procedure, as follows:

```
m_Loaded = True

' Load a new element of the Command1 control array
Command1.Load(1)
Command1(1).Visible = True

' Emulate Visual Basic 6 behavior by invoking the Resize event
Form1_Resize(Me, New System.EventArgs())
```

Heed Upgrade Warnings

The *Form1_Resize* event procedure for the upgraded Visual Basic .NET code includes the comment "UPGRADE_WARNING: Event Form1.Resize may fire when form is initialized." The Upgrade Wizard includes this type of warning for any upgraded event that is known to fire before the *Load* event. To be safe in situations such as this, you may want to implement the same types of changes that we applied to the *Resize* event in the previous example.

C1261587x.fm Page 277 Thursday, November 15, 2001 3:51 PM

The Default Form: *DefInstance*

Visual Basic 6 supports the notion of a default form. A default form is the default instance of the form that Visual Basic creates for you automatically. Having a default form instance allows you to write code such as the following:

```
Form1.Caption = "Caption for my default form"
```

Visual Basic .NET does not support a default form instance. If you write the equivalent code in Visual Basic .NET, as follows:

```
Form1.Text = "Caption for my default form"
```

you will run smack into a compiler error informing you that there is no shared member called *Text* on Form1. One way to solve this problem is to add a shared method to your form that returns the default instance—in other words, the first created instance—of the form. For example, for a default Visual Basic .NET Windows application project, you could add the following code to Form1:

```
Public Shared m_DefInstance As Form1
Public Shared ReadOnly Property DefInstance() As Form1
   'If no form instance exists, create one
   'Note: if the form is the startup form it will
   'already exist
   If m_DefInstance Is Nothing Then
      m_DefInstance = New Form1()
   End If
   Return m_DefInstance
End Property
```

This code sets up the *DefInstance* property but does not initialize the *m_DefInstance* variable to the first form instance created. To accomplish this initialization, you need to initialize the variable in the *Sub New* constructor for the form. The *Sub New* constructor exists on every form and can be found in the #Region code block labeled "Windows Form Designer generated code." Expand the #Region code block and insert the following assignment in *Sub New* at the end of the subroutine:

```
If m_DefInstance Is Nothing Then
   ' Initialize m_DefInstance to the first form created
   m_DefInstance = Me
End If
```

Now you can write code that accesses the default instance of a form as follows:

```
Form1.DefInstance.Text
```

C1261587x.fm Page 278 Thursday, November 15, 2001 3:51 PM

The default form is an example of a Visual Basic 6 semantic that does not exist in Visual Basic .NET. However, as we have demonstrated here, you can write code to cause your Visual Basic .NET forms to adopt the same semantic.

It is important to understand the purpose of the *DefInstance* property because you will likely see it used in your upgraded application. It is used when you access properties or controls using the name of the form. The Upgrade Wizard automatically implements the *DefInstance* shared method on all upgraded forms. However, the implementation that the wizard applies to your upgraded form is much more complicated than the simple example given here. The generated code takes into account whether the form is a startup or not. It also contains logic to initialize the default instance variable correctly when the form is a multiple document interface (MDI) form or an MDI child form. You can examine and even modify the code that the wizard generates. The *DefInstance*-related code is generated within the #Region blocks "Windows Form Designer generated code" and "Upgrade Support."

Application Lifetime and Forms

Chapter 10 introduced differences in application lifetime between Visual Basic 6 and Visual Basic .NET. Let's look into this area in more detail. An application in Visual Basic 6 application terminates when the last form is unloaded. A Visual Basic .NET application, on the other hand, terminates when the startup form associated with the application is unloaded. This means that your Visual Basic .NET application can terminate when one or more forms is still showing.

> **Note** The way you specify the startup form in Visual Basic .NET is similar to the way you do so in a Visual Basic 6 application. To set the startup form in Visual Basic .NET, right-click the project name in the Solution Explorer and select Properties from the shortcut menu. You are presented with the project settings dialog box, which groups various settings under a number of categories. Select the General category under Common Properties, and choose the desired startup form from the Startup Object list. Figure 12-3 shows where you can find the Startup Object list.

C1261587x.fm Page 279 Thursday, November 15, 2001 3:51 PM

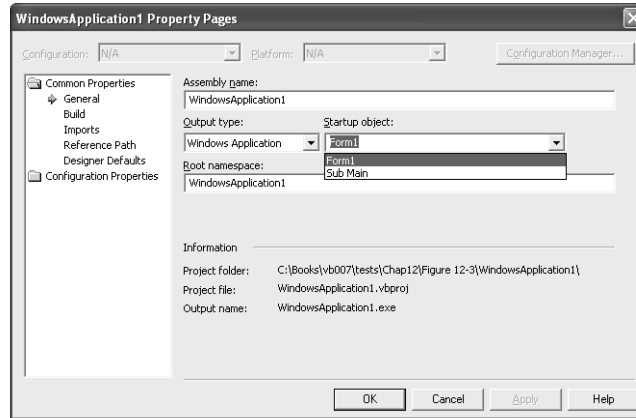

Figure 12-3. Startup Object list in the Visual Basic .NET project settings dialog box.

When you create a new Visual Basic .NET application or upgrade a Visual Basic 6 application, the lifetime of the startup form determines the lifetime of the application. If you unload the startup form, the application terminates no matter what other forms are currently open. If the startup form is your main form, this termination will most likely not be an issue. If, on the other hand, the form that determines the application lifetime is a secondary form, such as an MDI child form, you will probably need to make some changes to your application.

The simplest way to deal with this issue is to identify a form in your application that you consider to be the main form. For example, if your application is composed of an MDI form with a number of MDI child forms, you would identify the MDI form as the main form. You should set this form as the startup form for the application.

If you want your application to terminate when the last form is closed, you need to write code that manages the lifetime of your application. Let's look at a Visual Basic 6 application that has two forms: Form1 and Form2, in which Form2 is shown from Form1's *Load* event. In Visual Basic 6, the application terminates when you close both forms, regardless of the order in which you close them. Form1 contains the following code in its *Form_Load* event procedure to show Form2:

```
Private Sub Form_Load()
   Form2.Show
End Sub
```

C1261587x.fm Page 280 Thursday, November 15, 2001 3:51 PM

If you upgrade the project to Visual Basic .NET, the application will terminate as soon as you close Form1, even if Form2 is still showing. Suppose you want to change the behavior of the Visual Basic .NET application so that it behaves like the Visual Basic 6 application. The reason Form1 determines the lifetime for the Visual Basic .NET application is that the compiler includes an invisible piece of code that starts up your application and shows the startup form. The code the compiler includes is

```
System.Windows.Forms.Application.Run(New Form1())
```

Application.Run is a .NET Framework function that loops internally processing messages and waits for the form instance that is passed to it to terminate. If you do not want your application to be associated with any particular form, you can substitute a version of *Application.Run* that takes no arguments as follows:

```
System.Windows.Forms.Application.Run()
```

This version loops until you call *Application.Exit* from somewhere in your program. You can control the execution of your application by calling this version of *Application.Run* from *Sub Main*. For example, you could show Form1 and then call *Application.Run* from *Sub Main* as follows:

```
Sub Main()
   Form1.DefInstance.Show()   ' Equivalent to Form1.Show
                              ' in Visual Basic 6
   System.Windows.Forms.Application.Run()
End Sub
```

Create *Sub Main* in a new code module called Module1, and then set Module1 as the startup object for the project.

Since the application is no longer tied to the lifetime of any particular form, you need to add code that calls *Application.Exit* when the last form is unloaded. One way to do this is to have each form increment a global form count every time a form instance is created and then decrement the form count when the instance is destroyed. When the application form count is decremented and reaches 0, meaning that no more forms are loaded, call *Application.Exit*. To increment and decrement the global form count, you can add two functions to Module1 called *AddForm* and *RemoveForm*. You also need to declare a module-level variable that will keep track of the form count. Here is an example of the code you need to add:

```
Dim FormCount As Integer

Sub AddForm()
   FormCount += 1  'Equivalent to FormCount = FormCount + 1
End Sub
```

```
Sub RemoveForm()

   FormCount -= 1  'Equivalent to FormCount = FormCount - 1

   'At least one form remains, keep going
   If FormCount > 0 Then
      Exit Sub
   End If

   'No more forms - end the application
   'Note: Make sure this call is the last statement in the Sub
   'System.Windows.Forms.Application.Exit()

End Sub
```

You then need to add calls to *AddForm* and *RemoveForm* to both Form1 and Form2. You can add the call to *AddForm* to the *Form_Load* event procedure and a call to *RemoveForm* to the *Form_Closed* event procedure. Here is the resulting code for Form1:

```
Private Sub Form1_Load(ByVal eventSender As System.Object, _
                       ByVal eventArgs As System.EventArgs) _
                       Handles MyBase.Load
   AddForm()
   Form2.DefInstance.Show()
End Sub

Private Sub Form1_Closed(ByVal sender As Object, _
                         ByVal e As System.EventArgs) _
                         Handles MyBase.Closed
   RemoveForm()
End Sub
```

Here is the code you need to add to Form2:

```
Private Sub Form2_Load(ByVal sender As System.Object, _
                       ByVal e As System.EventArgs) _
                       Handles MyBase.Load
   AddForm()
End Sub

Private Sub Form2_Closed(ByVal sender As Object, _
                         ByVal e As System.EventArgs) _
                         Handles MyBase.Closed
   RemoveForm()
End Sub
```

C1261587x.fm Page 282 Thursday, November 15, 2001 3:51 PM

As long as Module1 is set as your startup object, the application will behave like the Visual Basic 6 application. You can close Form1 and Form2 in any order. When the last form is closed, *Application.Exit* is called and the application terminates.

MDI Forms

Both Visual Basic 6 and Visual Basic .NET support MDI parent and child forms. The difference between the two environments lies in how you create the MDI parent form and show the MDI child form. To create an MDI application in Visual Basic 6, you add a special *MDIForm* to your application to act as the MDI parent form. Visual Basic .NET has no concept of an *MDIForm* as a separate form type. Instead, you set the *IsMdiContainer* property of a Visual Basic .NET form to *True*.

In Visual Basic 6, you specify MDI child forms by setting the *MDIChild* property of a form to *True*. When the MDI child form is shown, it always displays within the MDI parent form. Visual Basic .NET does not work this way. In Visual Basic .NET, there is no *MDIChild* property. Instead, you must write code to set the *MdiParent* property of a form to the MDI parent form. For example, if you want Form2 to be an MDI child form of MDI parent Form1, you need to write the following code:

```
Dim MyForm2 As New Form2
MyForm2.MdiParent = Me
MyForm2.Show
```

In the code example given here, assume that the code is written in the module for Form1. For example, we can include this code in the *Form1_Load* event procedure to show the MDI child Form2 from Form1.

Lifetime of MDI Form Applications

As we discussed in the previous section, the lifetime of a Visual Basic .NET application is tied to the startup form. The same is true of MDI form applications, but with an interesting twist. Visual Basic 6 allows you to specify an MDI child form as the startup form in an MDI form application. Specifying an MDI child form as the startup form has the advantage that you don't have to write any code to show the MDI parent and MDI child form by default. When the MDI startup child form is shown, it checks to see whether the MDI parent form is showing. If it is not, the MDI parent is shown automatically and the MDI child form is displayed within the MDI parent.

If you upgrade an application in which an MDI child form is the startup form, you will find that it will terminate when you close the MDI child form. To prevent this, you need to specify the MDI parent form, not the child form, as the startup form.

Conclusion

This chapter has covered the general types of differences between Visual Basic 6 and Visual Basic .NET that you will encounter. For example, it looked at how property model differences and differences in technology play out in your upgraded Visual Basic .NET application. In addition, it looked at how form-related issues such as event ordering and lifetime can lead to subtle differences in the execution of your upgraded application. We have offered solutions to help you move your Visual Basic 6 code to Visual Basic .NET. Solutions are available even when the technology is no longer supported or the components and language statements that provide the support have changed drastically.

The characteristics that make up a Visual Basic form have remained largely constant since the inception of Visual Basic in 1988. The form is based on a Windows window in which a form designer allows you to drag and drop controls and set properties. The way you create a form and the behavior you expect from your form in a compiled application have also remained unchanged over the years. The Visual Basic .NET form is no exception. You create and interact with Visual Basic .NET forms in pretty much the same way as you do Visual Basic 6 forms. Differences become noticeable when you start using Visual Basic .NET forms on a more granular level. For example, properties do not always jive between Visual Basic 6 and Visual Basic .NET.

Figuring out the mapping between Visual Basic 6 and Visual Basic .NET properties is a matter of learning such details as that *Caption* always maps to *Text* or that *OLEStartDrag* maps to *DoDragDrop*. Most of these mappings can be looked up or memorized from a table. Appendix A of this book provides such a table. It should serve as your technical reference for mapping Visual Basic 6 objects and statements to their Visual Basic .NET equivalents.

C1261587x.fm Page 284 Thursday, November 15, 2001 3:51 PM