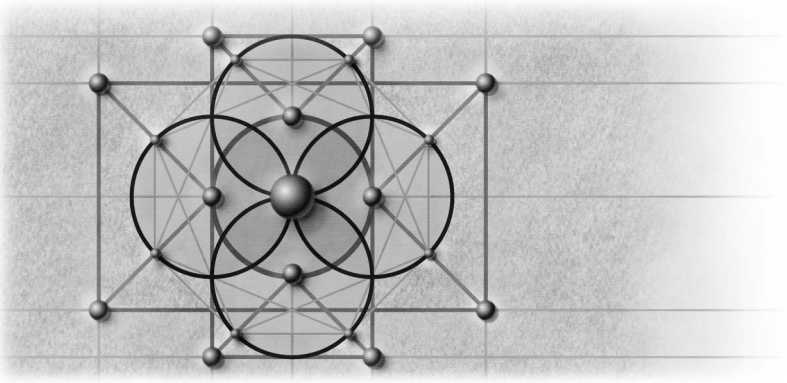

15

Problems That Require Redesign

As you get used to upgrading applications and become more familiar with the modifications needed to get them working in Microsoft Visual Basic .NET, you start noticing that some problems can be remedied with a one-line fix, whereas others require you to recode the problem area or even redesign it. This chapter looks at the most common of the redesign problems and shows you how to get each working in Visual Basic .NET.

If you give 50 different programmers the same problem to solve, you'll get 50 different solutions. We all do things differently, and any given programming problem can be solved in many different ways. For each of the problems in this chapter, we provide a sample solution that you can use as is in your own code. However, the Microsoft .NET Framework often provides several different mechanisms that have the same effect. You may find that another mechanism is more suitable for the particular problem you're trying to solve. Where possible, we try to make you aware of these mechanisms and direct you to sources where you can learn more about them.

Replacing the OLE Container Control

The OLE Container control was first introduced in Visual Basic 3 and is used to dynamically add objects to a form at run time. In Visual Basic 6, the OLE Container control is commonly used in three ways.

- To create a control that contains an embedded object, such as a Microsoft Word document. When your application is compiled, its state (the contents of the object) is compiled into your program. If the user changes the object at run time, you can programmatically save the updated state to a file using the *SaveToFile* method and subsequently reload it using the *ReadFromFile* method.
- To create a control that contains a linked object, such as a Word document. The difference between a linked object and an embedded object is that a linked object is linked to a file on disk instead of being compiled into the application. In the case of a linked document, any changes you make are immediately reflected in the document file, which can subsequently be loaded by Word.
- To bind to an OLE object in a database—typically to a picture. The OLE Container control can be data-bound to a field in a database using the DAO Data control or the RDO Remote Data control. The most common usage is to display a picture.

Visual Basic .NET does not support the OLE Container control. What happens to the control during an upgrade? Let's look at an example. Figure 15-1 shows a Visual Basic 6 form at design time, with an OLE Container control that has a link to a Word document stating, "VB Rocks!!"

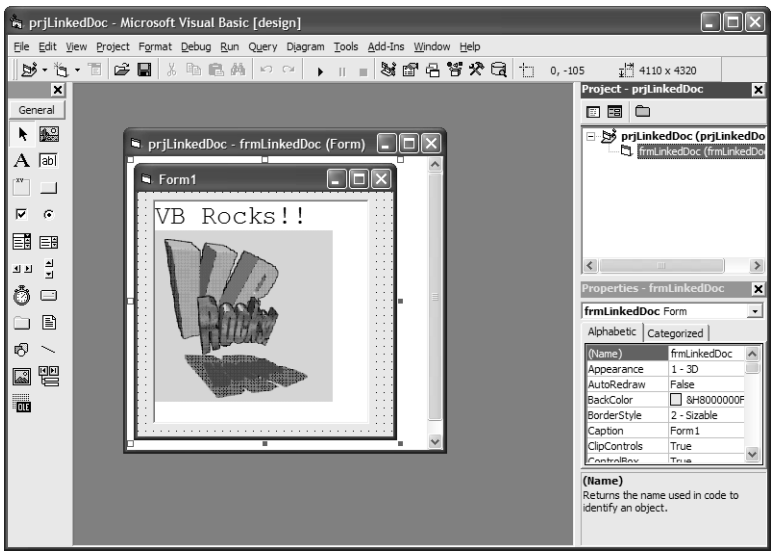

Figure 15-1 OLE Container control with linked object in Visual Basic 6.

When we upgrade the project, the OLE control is replaced with a bright red label, indicating that the control could not be upgraded. Figure 15-2 shows the upgraded form. If you look at the upgrade report, you will find an entry saying that the control was not upgraded, and any code that manipulates the control will be marked with an upgrade warning similar to the following:

```
'UPGRADE_ISSUE: Ole method OLE1.CreateLink was not upgraded.
```

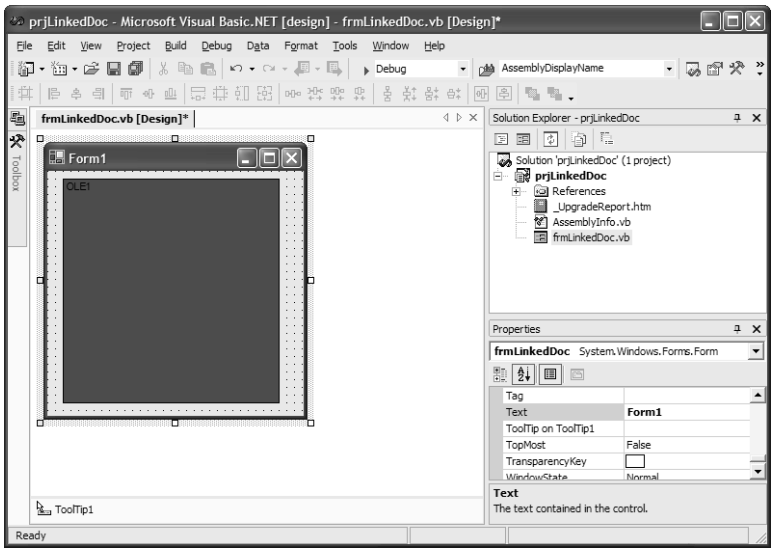

Figure 15-2 OLE Container control after upgrade.

Let's work on replacing the OLE control and getting our upgraded application to show the Word document as it did in Visual Basic 6. In many cases, we can use a WebBrowser ActiveX control to produce the same effect as a linked object within an OLE Container control. The WebBrowser resembles Microsoft Internet Explorer and can display documents, spreadsheets, and HTML pages using the *Navigate* method. It also allows you to edit these objects and save them to their original files. The WebBrowser is a lot like the linking part of OLE (which, you'll recall, is short for "object linking and embedding"). Using the WebBrowser control, we can open the Word document and display it on the form, as shown in Figure 15-3.

Figure 15-3 Word document displayed using the WebBrowser control.

Here are the steps required to replace an OLE control with the WebBrowser control:

1. Remove the red label from the form.
2. Add a WebBrowser control to the form. Finding this control can be a little confusing. For starters, it is contained in the obscurely named Shdocvw.dll. In the Customize Toolbox component picker, you need to select Microsoft WebBrowser Control on the COM Components tab. In the Toolbox it is called an Explorer control, and when you add it to your form it is called a WebBrowser control.
3. In *Form_Load* (or wherever you want to bind to the document), add the following code:

   ```
   Me.AxWebBrowser1.Navigate("C:\SampleCode\LinkedDoc.doc")
   ```

   You may want to substitute the filename in this example for the location of the object to which you are linking.
4. Press F5, and that should do the trick—the document should appear in the form. This method works for most objects that Internet Explorer can display, such as HTML files, documents, spreadsheets, GIFs, and JPEG files.

The other common use for the OLE Container control is to bind a control to an image in a database. In Visual Basic 6, this is done using the DAO and RDO Data controls. When such a project is upgraded to Visual Basic .NET, you'll see the red control mentioned previously, since neither the OLE Container control, the DAO Data control, nor the RDO Data control is supported in Visual Basic .NET.

Let's look at how you can achieve the same effect using ADO and a helper function in your Visual Basic .NET program. The following steps show how to bind a PictureBox to the Photo field of the Employees table in the Northwind database. It assumes that the NWind.mdb database is located in the root C:\ directory. If it is located somewhere else, you will need to change the file location in the following source. Here are the steps:

1. Create a new Visual Basic .NET Windows application.
2. Add a reference to ADODB.
3. Add the *adoHelper* helper class to the application. For the source code of the helper class, see the Bind OLE Picture application on the companion CD.
4. Add a PictureBox to the form.
5. In the *Form1_Load* event, add this code:

```
Dim cn As New ADODB.Connection()
Dim rs As New ADODB.Recordset()
Dim c As New adoHelper()
cn.Open("Provider=Microsoft.Jet.OLEDB.4.0;" & _
   "Data Source=C:\NWIND.MDB")
rs.Open("Employees", cn)
Me.PictureBox1.Image = c.rsFieldToBitmap(rs.Fields("photo"))
```

That's it! Press F5 and notice that the PictureBox retrieves the photo from the database, as shown in Figure 15-4.

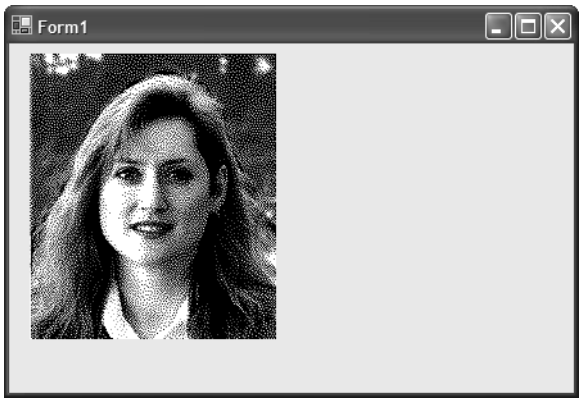

Figure 15-4 Binding a PictureBox to an OLE image in a database.

Replacing Painting Functions

Remember how, in Visual Basic 6, you could draw a circle on a form using the *Form.Circle* method? Remember also how you could make the line stay on the form by setting the *Form.AutoDraw* property to *True*? Well, things have changed a little in Visual Basic .NET. The graphics methods *Line*, *Circle*, *Print*, *PaintPicture*, and *Cls* have been removed from Form and PictureBox. Instead, Windows Forms has access to the .NET GDI+ object library, which has a much richer set of graphics functions. While all this is very exciting, you've probably guessed that the graphics methods you're used to are not supported in Windows Forms.

For example, suppose the *Click* event of a Button contains the following Visual Basic 6 code, which draws a line and a circle on the current form:

```
Me.Line (0, 200)-(2000, 200)
Me.Circle (300, 300), 100
```

The two methods cannot automatically be upgraded to Visual Basic .NET, so after upgrading, these statements become the following:

```
'UPGRADE_ISSUE: Form method Form1.Line was not upgraded
Me.Line(0, 200) To (2000, 200)
'UPGRADE_ISSUE: Form method Form1.Circle was not upgraded
Me.Circle(300, 300, 100)
```

The line and circle methods cause compile errors in Visual Basic .NET because they are not available as methods of the Form object. Where have the graphics methods gone? you may ask. They have been moved to the GDI+ graphics object. To use a GDI+ graphics object, you create a graphics object, draw onto the graphics object, and then dispose of the graphics object. For example, the following code paints a line and a circle on the form. Try it out by putting it in a *Button_Click* event in a Visual Basic .NET project.

```
Dim g As Graphics = Me.CreateGraphics
g.DrawLine(Pens.Black, 0, 20, 100, 20)
g.DrawEllipse(Pens.Black, 24, 24, 30, 30)g.Dispose()
```

Figure 15-5 shows the effect of running this code.

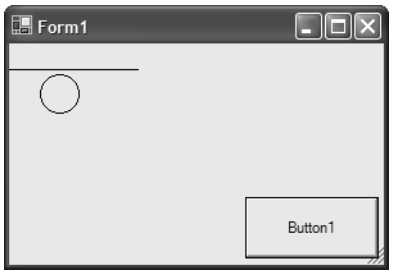

Figure 15-5 Drawing a line and a circle in Visual Basic .NET.

This code draws a line of length 100 and a circle with a diameter of 24. Notice that graphics coordinates are in pixels. In Visual Basic 6, the graphics unit was twips by default. Also notice that, as in Visual Basic 6, the origin is the top left corner. In this example, we put the graphics code in a *Click* event to ensure that the form is visible when the code is run. There is no *AutoRedraw* property in Visual Basic .NET. If the window is hidden, minimized, or moved off the screen, the line and circle you worked so hard to draw are erased. Luckily, Windows Forms gives you a way to redraw your objects every time the form is repainted. You do this using the *Form.Paint* event. This event is called every time a portion of the form's background has to be repainted. The following code shows how to paint a line and a circle using the *Paint* event. Notice that the system passes a graphics object to the event, so you don't need to create one yourself.

```
Private Sub frmLineAndCircle_Paint(ByVal sender As Object, _
ByVal e As System.Windows.Forms.PaintEventArgs) Handles MyBase.Paint
   e.Graphics.DrawLine(Pens.Black, 0, 20, 100, 20)
   e.Graphics.DrawEllipse(Pens.Black, 24, 24, 30, 30)End Sub
```

In most applications, *Form.Paint* is the best place to put graphics code, since the graphics are redrawn each time the form is repainted.

Now let's look at something else you can do in the *Paint* event using GDI+. The following *Paint* event makes the form background fade from black to white:

```
Private Sub frmGradient_Paint(ByVal sender As Object, _ByVal e As Sys-
tem.Windows.Forms.PaintEventArgs) Handles MyBase.Paint   Dim rectForm
As New Rectangle(New Point(0, 0), Me.ClientSize)   Dim l As New
Drawing2D.LinearGradientBrush(rectForm, Color.Black, _
Color.White, Drawing.Drawing2D.LinearGradientMode.Horizontal)
e.Graphics.FillRectangle(l, rectForm)End Sub
```

Figure 15-6 shows the form with the gradient background. Notice that this form also has a TextBox and a PictureBox on it, and that these controls haven't been painted with the gradient. This brings us to an important point: the graphics object passed to the *Form.Paint* event paints only the form background. Each control on the form also has a *Paint* event, which you can use to draw on that particular control.

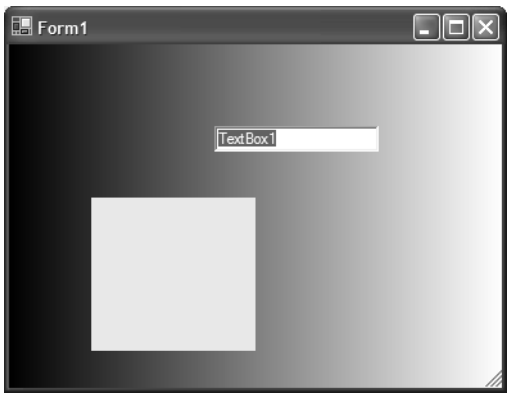

Figure 15-6 Black to white background on a form.

In addition to circles, lines, and gradient backgrounds, the graphics object also has all the methods that were available in Visual Basic 6. To erase the background, use the *Clear* method; to paint a picture, use the *PaintImage* method; and to print text onto a form, use the *DrawString* method. These methods are incredibly flexible. For example, the following code draws the string "VB Rocks" onto the background of a form, using the contents of a PictureBox as a brush:

```
Dim bitmapBrush As New Drawing.TextureBrush(PictureBox1.Image)
Dim f As New Font("Arial", 60, FontStyle.Bold, GraphicsUnit.Pixel)
e.Graphics.DrawString("VB Rocks", f, bitmapBrush, 20, 20)
```

As you can see, the GDI+ methods are quite powerful.

Learn More About GDI+

What we've shown here just scratches the surface. GDI+ is a fully featured set of graphics classes. You can draw paths, rectangles, icons, and pictures. You can rotate and transform these objects, and you can use alpha blending to fade one color into another. You will find GDI+ in the System.Drawing namespace. To learn more, search for "Introduction to GDI+" and "Using GDI+ Managed Classes" in the Visual Basic .NET Help system.

Rewriting Clipboard Code

Visual Basic 6 has a *Clipboard* object that allows you to store and retrieve text and pictures to and from the Windows Clipboard. In Visual Basic .NET, you manipulate the Clipboard using the System.Windows.Forms.Clipboard namespace. The new Clipboard classes are more flexible than those in Visual Basic 6, in that they allow you to set and retrieve data in a particular format and query the contents of the Clipboard to see what formats it supports. Unfortunately, Clipboard code cannot be upgraded automatically. However, you'll find it straightforward to implement the same functionality in Visual Basic .NET.

Let's look at an example. Suppose you have a Visual Basic 6 form with two PictureBox controls on it, sourcePictureBox and destinationPictureBox. The following code sets and retrieves the text "VB Rocks" and then copies a picture from one PictureBox to the Clipboard and into a second PictureBox:

```
'Set and retrieve text
Dim s As String
Clipboard.SetText "VB Rocks"
s = Clipboard.GetText
'Set and retrieve a picture
Clipboard.SetData Me.sourcePictureBox.Picture
Me.destinationPictureBox.Picture = Clipboard.GetData
```

When this code is upgraded, the Clipboard code is left as is and is marked with upgrade warnings:

```
'UPGRADE_ISSUE: Clipboard method Clipboard.SetText was not
upgraded.Clipboard.SetText("VB Rocks")
```

The upgraded code causes compile errors in Visual Basic .NET. To get the same functionality, we have to delete the upgraded code and replace it with code that uses the *System.Windows.Forms.Clipboard* objects:

```
'Set and retrieve text
Dim s As String
Dim stringData As IDataObject
Dim getString As New DataObject(DataFormats.StringFormat)
Clipboard.SetDataObject("VB Rocks", True)
stringData = Clipboard.GetDataObject()
If stringData.GetDataPresent(DataFormats.Text) Then
   s = stringData.GetData(DataFormats.Text, True)
End If
'Set and retrieve a picture
Dim pictureData As IDataObject
Clipboard.SetDataObject(Me.sourcePictureBox.Image)
pictureData = Clipboard.GetDataObject
If pictureData.GetDataPresent(DataFormats.Bitmap) Then
   Me.destinationPictureBox.Image = _
      pictureData.GetData(DataFormats.Bitmap, True)
End If
```

The Visual Basic .NET code improves upon the Visual Basic 6 code by checking for the existence of a compatible Clipboard format before setting the value of the string and the PictureBox image.

Using the Controls Collection

To add and remove controls at run time in Visual Basic 6, you can use the controls collection. For example, the following code adds a TextBox dynamically at run time:

```
Dim myControl As Control
Set myControl = Me.Controls.Add("VB.TextBox", "myControl")
myControl.Visible = True
```

This code adds a TextBox to the form and makes it visible. Figure 15-7 shows the result.

Figure 15-7 Adding a TextBox to a Visual Basic 6 application dynamically at run time.

It is also easy in Visual Basic 6 to remove a control dynamically. The following line of code removes the TextBox we just added:

```
Me.Controls.Remove "myControl"
```

One of the annoying things about the Visual Basic 6 model is that the controls collection has no IntelliSense, so you have to remember the methods, parameters, and ProgID of the control to add. There is no automatic upgrade

for the controls collection, but it is easy to add and remove intrinsic controls in Visual Basic .NET. Here is the code for adding a TextBox to a form:

```
Dim c As Control
c = New TextBox()
c.Name = "myControl"
Me.Controls.Add(c)
```

In Windows Forms, controls are indexed by number, not by name. To remove a control by name, you have to iterate through the controls collection, find the control, and remove it. The following code shows how to remove the newly added control by name:

```
Dim c As Control
For Each c In Me.Controls
   If c.Name = "myControl" Then
      Me.Controls.Remove(c)
      Exit For
   End If
Next
```

Adding ActiveX controls dynamically at run time is a bit more work. As we discussed in Chapter 13, Visual Basic .NET creates wrappers for ActiveX controls. These wrappers must exist before a control can be added. In addition, most ActiveX controls have design-time licenses that must be present before the control can be created on the form. Visual Basic .NET compiles the license into the executable. These factors mean that the control must already be present in the project before it can be added dynamically at run time. One way to do this is to add a dummy form to the project and put all ActiveX controls that will be added dynamically onto this form. After you've created this dummy form, you can add the ActiveX control dynamically to any form in your project. The following code shows how to add a Windows Common Controls TreeView control dynamically to a form (the project already has a dummy form with a TreeView added):

```
Dim c As New AxMSComctlLib.AxTreeView()
c.Name = "myTreeView"
Me.Controls.Add(c)
```

The control can be removed dynamically in a manner similar to removing an intrinsic control:

```
Dim c As Control
For Each c In Me.Controls
   If c.Name = "myTreeView" Then
      Me.Controls.Remove(c)
      Exit For
   End If
Next
```

Using the Forms Collection

The forms collection in Visual Basic 6 is a collection of all open forms in the project. The most common uses of the forms collection are to determine whether a form is open, to iterate through the forms collection, and to close a form by name. The following Visual Basic 6 example shows how to iterate through the forms collection, determine whether a form called Form1 is open, and then unload Form1 if it is open:

```
Dim f As Form
For Each f In Forms
   If f.Name = "Form1" Then
      Unload f
      MsgBox "Form unloaded"
   End If
Next
```

The forms collection cannot be upgraded automatically, so if the previous code is upgraded, the collection is marked with an upgrade issue:

```
'UPGRADE_ISSUE: Forms collection was not upgraded.
For Each f In Forms
```

The only way to achieve the same effect in Visual Basic .NET is to implement your own forms collection. Luckily, this is easier than it sounds. First create a new Visual Basic .NET Windows application. Add a new module, and in the module put the following code (you can find this code on the companion CD in the file FormCollectionClass.vb):

```
Module FormsCollection
Public Forms As New FormsCollectionClass()
End Module
```

(continued)

```
Class FormsCollectionClass : Implements IEnumerable
   Private c As New Collection()
   Sub Add(ByVal f As Form)
      c.Add(f)
   End Sub
   Sub Remove(ByVal f As Form)
      Dim itemCount As Integer
      For itemCount = 1 To c.Count
         If f Is c.Item(itemCount) Then
            c.Remove(itemCount)
            Exit For
         End If
      Next
   End Sub
   ReadOnly Property Item(ByVal index) As Form
      Get
         Return c.Item(index)
      End Get
   End Property
   Overridable Function GetEnumerator() As _
      IEnumerator Implements IEnumerable.GetEnumerator
      Return c.GetEnumerator
   End Function
End Class
```

This is the collection that will maintain your collection of forms. You need to make sure that each form is added to the collection when it is created and removed from the collection when it is disposed of. In the *New* event for Form1, insert the following line:

```
Forms.Add(Me)
```

In the *Disposed* event, insert this line:

```
Forms.Remove(Me)
```

That's all you need to do to implement your forms collection. If your application has many forms, you'll have to put the *New* and *Dispose* code in every one.

Let's see how to use this collection in code. The following Visual Basic .NET example does exactly what the first Visual Basic 6 example in this chapter did: loop through the forms collection, determine whether a form called Form1 is open, and unload it if it is open.

```
Dim f As Form
For Each f In Forms
   If f.Name = "Form1" Then
      f.Close()
   End If
Next
```

You can use this forms collection to unload forms, loop through the collection of open forms, or simply check whether a particular form is open.

Do You Really Need a Forms Collection?

Before adding a forms collection to your project, you may want to ask yourself whether you really need it. One drawback is that you need to remember to add the code to every form in your application. If you forget to add it to a single form, that form will not be added to the collection. If you do use the forms collection, and you use it a lot, consider adding the *New* and *Dispose* code to the template from which every new form is created.

Some applications created with the Visual Basic Application Wizard contain code that loops through the forms collection and closes every form in the application. In many cases you can safely remove this code from the project and avoid having to create your own forms collection.

Upgrading *PrintForm* Code

Visual Basic 6 offers a simple way to print forms using the *Form.PrintForm* method. *PrintForm* is not perfect: some controls don't print well, and the quality is not always picture-perfect. But for many people, it's good enough. Unfortunately, Visual Basic .NET has no direct equivalent of *PrintForm*, and thus the

Upgrade Wizard can't upgrade *PrintForm* code. Don't worry, though; we've found a replacement for you, using a *PrintForm* helper class. (For the source code to this class, see the PrintForm application on the companion CD.) Here are the steps to implement *PrintForm* behavior in Windows Forms:

1. Create a new Visual Basic .NET Windows application.
2. Add the *PrintForm* helper class.
3. Add a Button to the default form, and in the *Button_Click* event insert the following code:

```
Dim c As New PrintForm()
c.Print(Me)
```

4. Run the application, and click the button. Bingo! Your form will be sent to the printer.

You should be aware of one limitation: the form to be printed must be the topmost form and must be entirely visible, since the class takes a "snapshot" of the form and prints it. In many cases this technique will be good enough, especially when you simply want to print the visible layout of the current form. As a bonus, the helper class also contains another function, *getImageFromForm*. This method copies the image of the form to an image object. You can use this to, say, copy the image from one form to another. Let's see how this works by extending the previous example:

1. Add a second form, Form2, to your application.
2. Add a second Button to Form1, and in the *Button2_Click* event insert the following code:

```
Dim c As New PrintForm()
Dim i As Image
i = c.getImageFromForm(Me)
Dim f2 As New Form2()
f2.Show()
f2.CreateGraphics.DrawImage(i, New Point(0, 0))
```

3. Run the application, and click Button2. The application paints the image of Form1 onto Form2, as shown in Figure 15-8. The controls painted onto Form2 won't work, since we've simply painted the form image. Nevertheless, it's a convincing illusion.

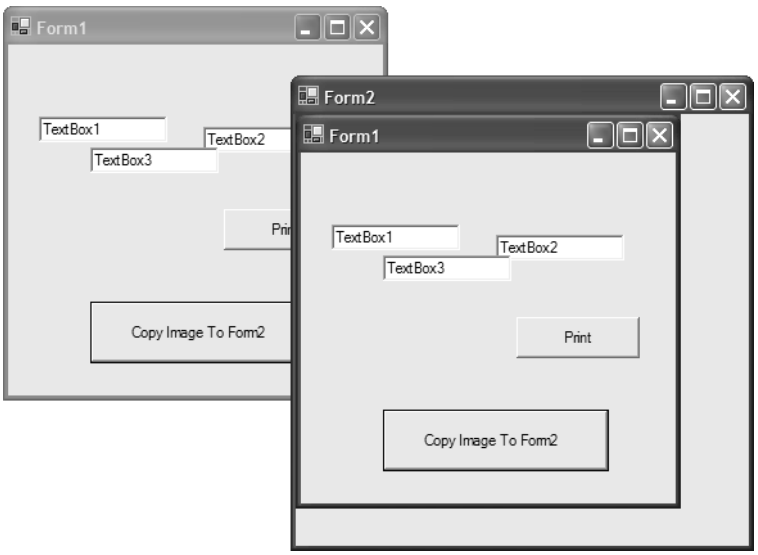

Figure 15-8 Which is the real Form1?

Where Are the Print Functions?

The printing model in the .NET Framework is quite different from that in Visual Basic 6. If you're looking for the printing functions, you'll find them in the System.Drawing.Printing namespace. For help with printing programmatically from your application, search for the topic "Printing with the PrintDocument Component" in the Visual Basic .NET Help system.

Replacing Property Pages

Suppose you're writing an image list user control, and you want to give users a way to add a collection of images. How do you do it? In Visual Basic 6, you would add a property page to your control. In the property page, you would write code that manages the collection. Property pages are useful because they allow you to work around the limitations of the Visual Basic 6 Property Browser, which can edit only strings, enums, numbers, colors, pictures, and fonts. Editing a collection of bitmaps is out of the question for the poor old Visual Basic 6 Property Browser. You'll be pleased to know that Visual Basic .NET has a Property Browser that can edit any .NET variable type or class—property pages are no longer needed.

Property pages in user controls are not upgraded automatically. To enable your control to edit the values that used to be exposed in the control's property pages, you need to add properties to your user control. The good news is that the Visual Basic .NET Property Browser automatically recognizes the types of your control's properties and allows users to edit them in the Property Browser. In the case of a bitmap collection, you simply add the property to your user control, and the Visual Basic .NET Property Browser provides a bitmap collection editor. Let's try this out. Add the following code to a Windows Forms user control:

```
Dim m_bitmapCollection() As Bitmap
Property BitmapCollection() As Bitmap()
   Get
      Return m_bitmapCollection
   End Get
   Set(ByVal Value As Bitmap())
      m_bitmapCollection = Value.Clone
   End Set
End Property
```

If you add the control to a form, you will notice that it has a new property, *bitmapCollection*, and that you can edit this property using a collection editor. Figure 15-9 shows this editor. It's that's simple.

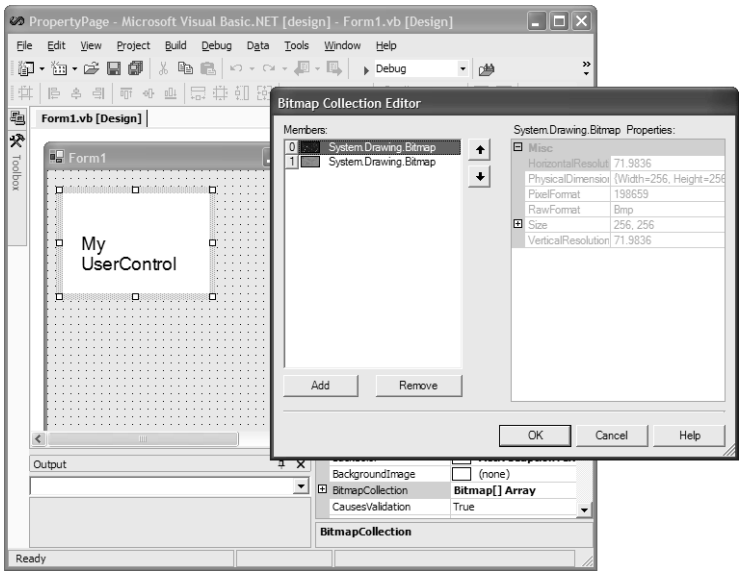

Figure 15-9 Adding a bitmap collection to your user control.

There are property page editors for every .NET class. For example, if you add a *targetDate* property of type *Date* to your user control, the Property Browser automatically provides a date editor. Try putting this property code into your user control:

```
Private m_targetDate As Date
Public Property targetDate() As Date
   Get
      Return m_targetDate
   End Get
   Set(ByVal Value As Date)
      m_targetDate = Value
   End Set
End Property
```

The Property Browser automatically uses the date editor for the property, as shown in Figure 15-10.

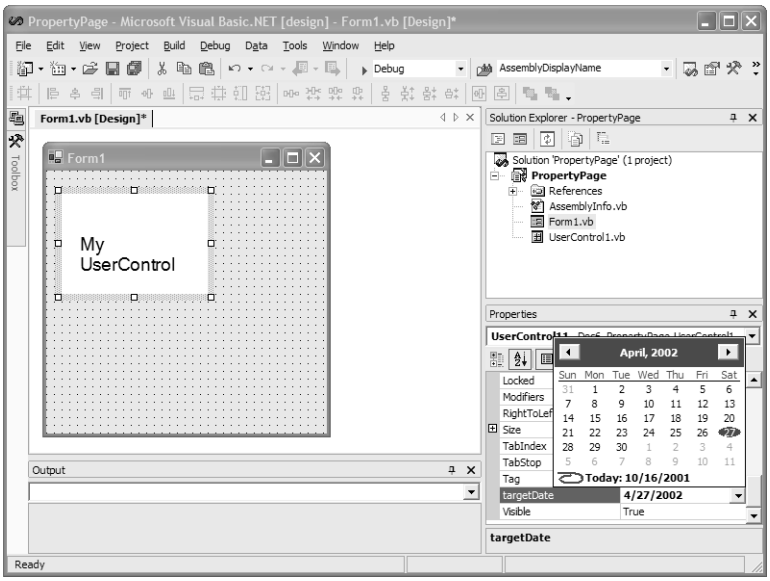

Figure 15-10 Date editor provided by the Property Browser.

The advantage of showing properties in the Property Browser instead of in property pages is that in the Property Browser all of the properties for your control are visible and easily discoverable. Compare this with Visual Basic 6 property pages, which hide functionality from people who use your controls.

Writing Your Own Property Editors

The Windows Forms Property Browser can edit any .NET class or variable type. But what happens if you write a new class, say a *Customer* class, and you want users to be able to edit it in the Property Browser? Windows Forms gives you a way to provide this option by allowing you to write your own custom property editor that will be used whenever people edit properties of type *Customer*. To learn more about writing custom property editors, search for the following two topics in the Visual Basic .NET Help system: "Attributes and Design-Time Support" and "Implementing a UI Type Editor."

Eliminating *ObjPtr*, *VarPtr*, and *StrPtr*

Visual Basic 6 supports the undocumented functions *ObjPtr*, *StrPtr*, and *VarPtr*. These functions return the memory address of the location where the corresponding object, variable, or string is stored. This memory address, also known as a pointer, can then be passed to Windows APIs that accept memory addresses as parameters. Not surprisingly, these functions are rarely used, and also not surprisingly, programmers can get into real trouble using them, since they bypass the memory management and safety features of Visual Basic 6.

There is no automatic upgrade for *ObjPtr*, *VarPtr*, and *StrPtr*, but there is a way to achieve the same functionality in Visual Basic .NET. Before continuing, we must warn you that manipulating memory is not a good practice. You shouldn't need to do it; there is almost always a more appropriate workaround that uses safer techniques.

Okay, okay, if you really want to know how to get the memory address in Visual Basic .NET, we'll show you. But remember, this stuff is dangerous! Let's look at some examples. Create a new Visual Basic .NET console application. Our examples will use the default module. We will be using the System.Runtime.InteropServices namespace, so at the top of the module, add the following line of code:

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

Memory pointer functions are commonly used with the CopyMemory API, so let's add a *Declare* statement for the API in the declarations section of our module:

```
Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" _
(ByVal lpDest As Integer, ByVal lpSource As Integer, _
ByVal cbCopy As Integer)
```

After adding this statement, your module should look like the following:

```
Imports System.Runtime.InteropServices
Module Module1
    Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" _
    (ByVal lpDest As Integer, ByVal lpSource As Integer, _
    ByVal cbCopy As Integer)

    Sub Main()

    End SubEnd Module
```

We will be working with our examples inside the *Sub Main* method. In Visual Basic .NET, variables are garbage-collected and can in theory be moved around in memory arbitrarily by the Garbage Collector. As a result, getting the memory address of a variable poses a problem, since if the Garbage Collector decides to move the variable in memory, the memory address we just retrieved becomes invalid. Luckily, the Garbage Collector provides a method to "pin" the object in memory so that it won't be moved around. The *GCHandle.Alloc* method gets a handle to the variable and pins it in memory. The handle's *AddrOfPinnedObject* method then returns the memory address. Let's see this in action. The following code creates an integer, pins it in memory, displays the memory address of the variable, and then unpins the handle. (You should always unpin the handle when you're finished playing around with the memory address.)

```
Sub Main()
   Dim myInteger As Integer
   Dim handle As GCHandle = _
      GCHandle.Alloc(myInteger, GCHandleType.Pinned)
   Dim address As Integer = handle.AddrOfPinnedObject.ToInt32
   MsgBox("The memory address of myInteger is " & address)
   handle.Free()End Sub
```

Let's look at another example. This code creates two strings, gets the memory addresses of each, and then uses the CopyMemory API to copy one string into another:

```
Sub Main()
   Dim src, dest As String
   src = "VB Rocks"
   dest = "            "
   Dim srcHandle As GCHandle = _
      GCHandle.Alloc(src, GCHandleType.Pinned)
   Dim srcAddr As Integer = srcHandle.AddrOfPinnedObject.ToInt32
   Dim destHandle As GCHandle = _
      GCHandle.Alloc(dest, GCHandleType.Pinned)
   Dim destAddr As Integer = destHandle.AddrOfPinnedObject.ToInt32
   CopyMemory(destAddr, srcAddr, Len(src) * 2)
   srcHandle.Free()
   destHandle.Free()
   MsgBox(dest)
End Sub
```

If you run this code, you'll see that the source string is copied into the destination string, which is then shown in a message box. Something to be aware of is that variables are stored differently in Visual Basic .NET than in Visual Basic 6. Notice that when you use the CopyMemory API, the third parameter is the number of bytes to copy, which is double the length of the string. In Visual Basic .NET, strings are stored as Unicode, so each character is stored in 2 bytes. Thus, when copying a string using APIs, you have to specify a number of bytes that is double the string length.

Conclusion

This chapter has looked at eight problems that require some redesign work when moving from Visual Basic 6 to Visual Basic .NET, and it has shown you how to make the necessary changes to your code. In all of the cases, the redesign is straightforward—you just have to know how to do it. The next chapter looks at more situations that require redesign, all related to upgrading MTS and COM+ services applications.