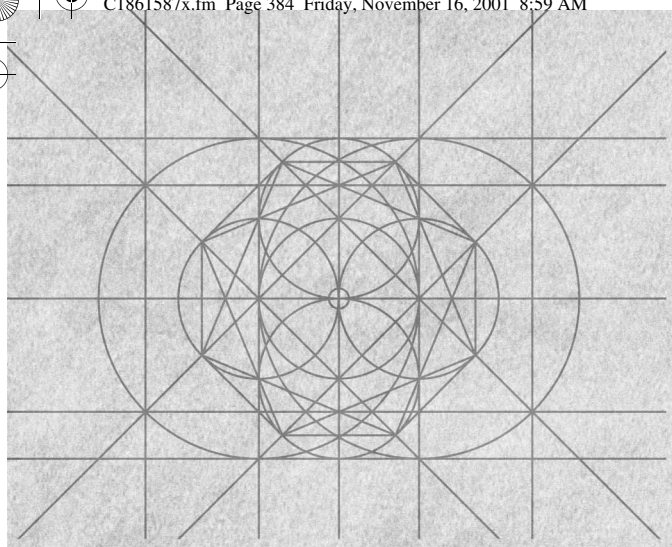
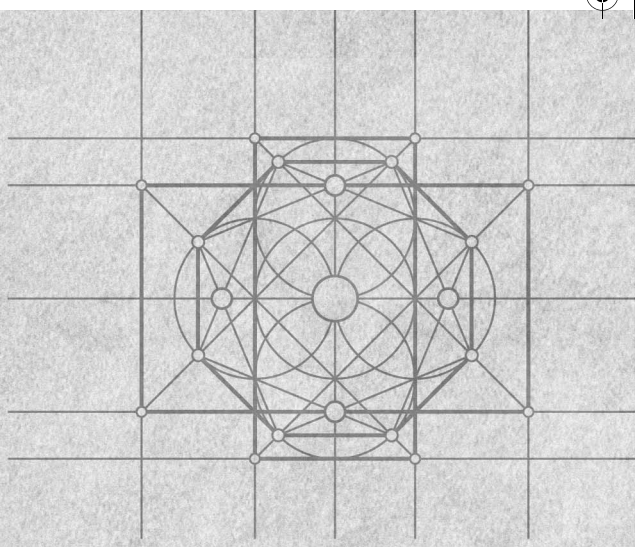
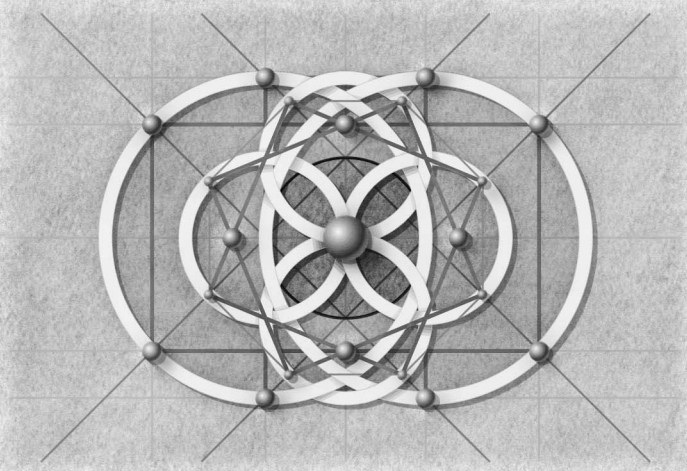
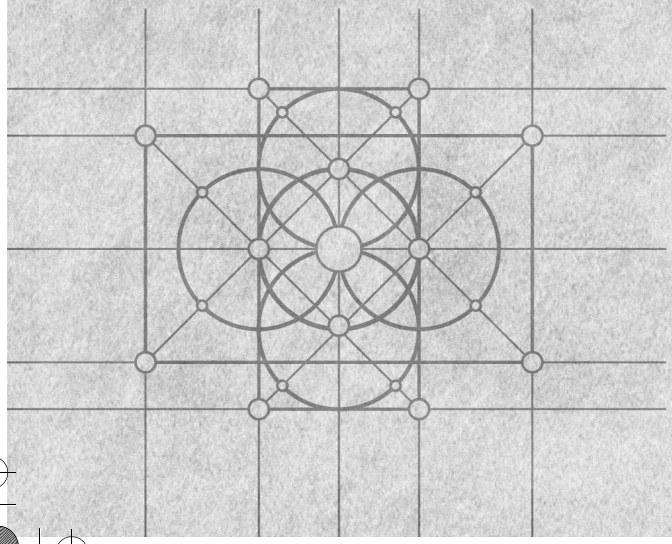
Part IV

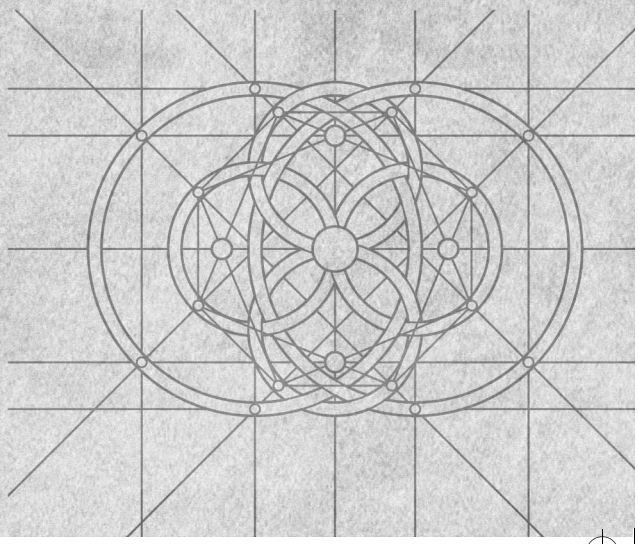
Techniques for Adding Value

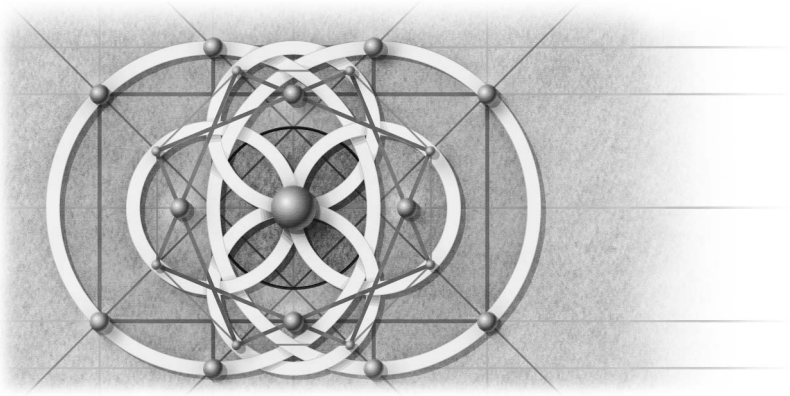












18

Adding Value to Your Applications

Parts I, II, and III of this book discussed how to prepare applications for upgrading, introduced the Microsoft Visual Basic .NET upgrade technologies, and looked at how to fix problems in your upgraded application. The final four chapters of this book depart from the subject of upgrading and instead look at how you can continue developing your application in Visual Basic .NET and how you can add value by incorporating new features of Visual Basic .NET into your application.

In this chapter, we'll demonstrate a number of new features that you may want to add to your upgraded applications. You'll see how to load objects dynamically from other DLLs, create rich graphics that were out of the reach of Visual Basic 6 programming, access the registry with less effort than ever before, and work with the powerful new Microsoft .NET file classes.

Note that the features we discuss in this chapter are all optional additions to your projects. We introduce some new ways of doing familiar things, plus some techniques that were not possible in Visual Basic 6. This chapter does not provide an exhaustive list of new capabilities in Visual Basic .NET; instead, it simply describes some cool features you can add to upgraded projects.

Throughout this chapter we'll refer to the sample application DynamicApp.sln, located on the companion CD.

Overview of the Sample Application

DynamicApp is a Microsoft Windows application with three forms. The primary form, Dashboard, is a dashboard from which you can launch the other features of the application: a registry editing form and a snappy graphics form, as seen in Figure 18-1.

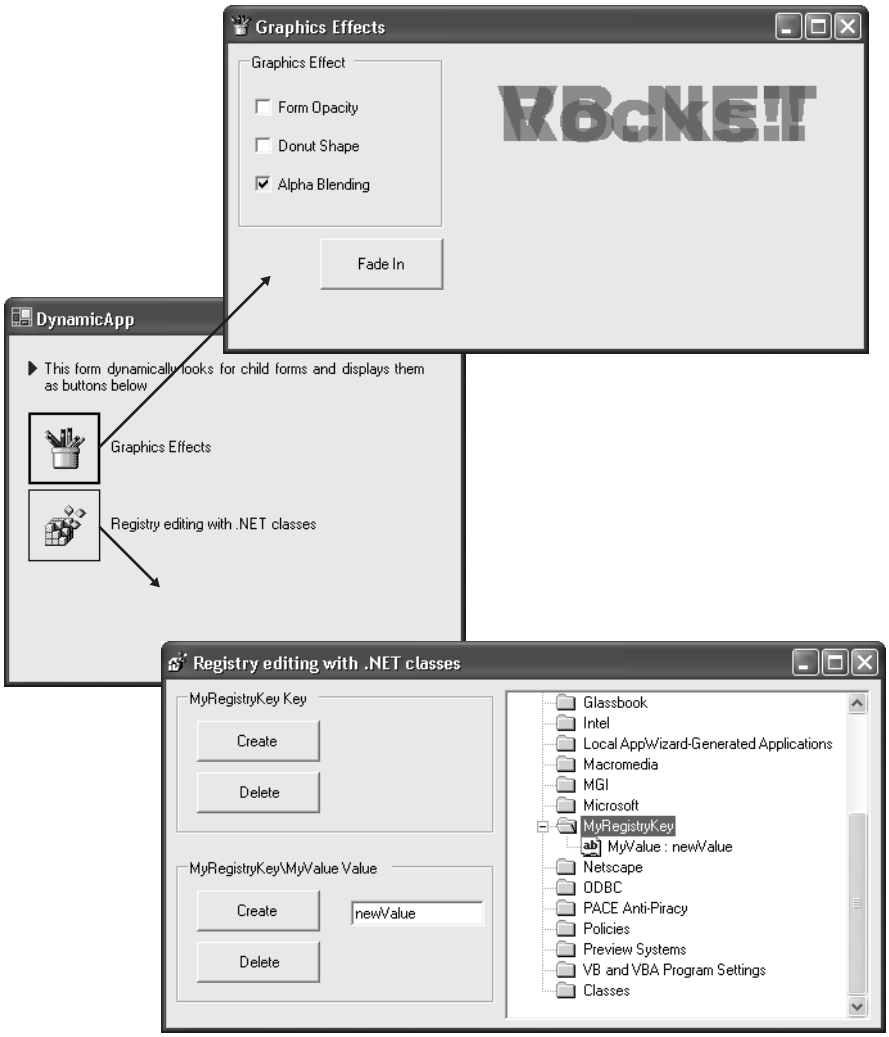


Figure 18-1 Dashboard opens the registry editing form and the snappy graphics form.

What is interesting is the architecture of the application. Each form is in a different application; the dashboard is in the main EXE, while the graphics form and registry editing form are in two separate DLLs. When the dashboard opens, it searches its directory for DLLs and then examines each DLL it finds for Windows forms. It dynamically builds a list of available Windows forms and shows them as buttons on the dashboard. When you click a button on the Dashboard form, the appropriate DLL is loaded dynamically and the form is shown. These actions are performed using some of the new file functions in Visual Basic .NET.

New File Functions

As we mentioned before, the Dashboard form handles three tasks of interest:

- It obtains a list of DLLs in a directory.
- It examines each of the DLLs for Windows forms.
- It dynamically loads and shows the form.

Let's see how it performs each of these actions.

Reading the Contents of a Directory

In Visual Basic 6, it is awkward to read the contents of a directory: you have to call the *Dir$* function repeatedly. This function returns the name of the first file in the directory, and then the next, and so on. You can't examine two directories at the same time because *Dir$* tracks only the file it has enumerated for the current directory. In Visual Basic .NET, getting the list of files in a directory is much simpler. Use the *System.IO.Directory.GetFiles* method to return an array of filenames. The following sample code is from the *Dashboard.GetSubForms* method in the DynamicApp project:

```
Dim filePathArray() As String
filePathArray = System.IO.Directory.GetFiles(myDirectory(), "*.dll")
```

After this code runs, the *filePathArray* variable is filled with an array of file paths, such as

```
C:\DynamicShell\bin\GraphicsFeatures.dll
```

The System.IO namespace contains many other useful functions as well. For example, *System.IO.Path.GetDirectoryName* returns the directory name from a file path. The code

```
MsgBox(System.IO.Path.GetDirectoryName( _
   "C:\DynamicShell\bin\GraphicsFeatures.dll"))
```

shows C:\DynamicShell in the message box. In a similar vein, the *System.IO.Path.GetFileName* method returns the filename of a file path. For example, the code

```
MsgBox(System.IO.Path.GetFileName( _
  "C:\DynamicShell\bin\GraphicsFeatures.dll"))
```

shows GraphicsFeatures.dll in the message box. Another useful function is *System.Reflection.Assembly.GetExecutingAssembly*. This last method returns the file path of the application. DynamicApp passes the file path to *System.IO.Path.GetDirectoryName* to get the application's directory, which is how it knows where to look for DLLs.

Finding All the Forms in a DLL

Once an application has found a DLL, how can it dynamically determine what forms it contains? The enabling technology is called **reflection**. Reflection encompasses a set of classes that allow you to examine the contents of a DLL at run time. Let's see how it works, using a simplified version of the *Dashboard.GetSubForms* method in DynamicApp. The following sample shows how to list all of the types (*type* is the .NET term for class, module, or form) in the current application. Try it out. Create a new Windows application, and enter the following code in the *Form_Load* event:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
   Dim asm As Reflection.Assembly
   Dim typeArray() As Type
   Dim typeCounter As Integer
   asm = asm.LoadFrom( _
      System.Reflection.Assembly.GetExecutingAssembly.Location)
   typeArray = asm.GetTypes()
   For typeCounter = 0 To typeArray.Length - 1
      MsgBox(typeArray(typeCounter).Name)
   Next
End Sub
```

There is only one type in this application: Form1. So when you run this code, Form1 is shown in the message box. What's happening here? Remember that *assembly* is the .NET term for an EXE or a DLL. We declare a variable of type *Assembly* and then use it to load the current application. Next we retrieve all of the types in the assembly into an array. We then loop through the array and show the name of each type in a message box. Simple, isn't it?

DynamicApp uses this mechanism to get the list of forms inside every DLL in the bin directory where the application is located. Instead of showing the form name in a message box, it stores the list of forms in an array and uses that array to create a set of buttons on the Dashboard form. When the user clicks a button, DynamicApp shows the appropriate form.

Loading Forms Dynamically

Once you know the name of the DLL and the name of the form you want to show, the rest is quite simple. You create an instance of the form, using the *Assembly.CreateInstance* method, and then call the *Show* method on the form. The following code snippet is a simplified version of the code in the DynamicApp *Dashboard.Button_Click_Handler* method. It demonstrates how to load a DLL called MyDll.dll, create an instance of a form within the DLL Form1, and show the form.

```
Dim asm As Reflection.Assembly
Dim f As Form
asm = asm.LoadFrom("MyDll.dll")
f = CType(asm.CreateInstance("MyDll.Form1"), Form)
f.Show()
```

Notice with the *CreateInstance* method that we have to refer to the form as MyDll.Form1, using the *<namespace>.<formname>* format. Also be aware that when you write code that loads a DLL dynamically, all names, including the filename and type name, are case sensitive.

Reading and Writing to Files

While we're on the subject of the new file functions, let's take a short digression and quickly discuss how to write to and read from files using the new .NET Framework methods. You can still use the existing Visual Basic file functions, but the .NET Framework file methods, although a little more complicated, offer more flexibility when working with files.

We'll create a simple console application that creates a file, writes "Hello World" to it, closes the file, and then reopens it and shows the contents in a

message box. This sample is on the companion CD, and it is called FileReadAndWrite.sln. The project contains one module. Here are the contents of that module:

```
Sub Main()
   '* Create a file and write to it
   Dim outFile As System.IO.FileStream
   outFile = New System.IO.FileStream("C:\tempFile.txt", _
      IO.FileMode.Create, IO.FileAccess.Write)
   Dim fileWriter As New System.IO.StreamWriter(outFile)
   fileWriter.WriteLine("Hello World")
   fileWriter.Close()
   outFile.Close()

   '* Open a file and read from it
   Dim inFile As System.IO.FileStream
   inFile = New System.IO.FileStream("C:\tempFile.txt", _
      IO.FileMode.Open, IO.FileAccess.Read)
   Dim fileReader As New System.IO.StreamReader(inFile)
   While fileReader.Peek > -1
      MsgBox(fileReader.ReadLine)
   End While
   fileReader.Close()
   inFile.Close()
 End Sub
```

To open a file for writing, you have to perform two steps: create a stream object and then create a *StreamWriter* object to write to the stream. You can then write to the file using the *StreamWriter* object's *Write* and *WriteLine* methods. Reading from a file involves a similar process: create a stream object, and then create a *StreamReader* to read from the stream. To determine whether there is anything to read from the file, use the *StreamReader* object's *Peek* method, which returns the value of the next byte in the file, or –1 if there is nothing left to read. The *StreamReader* object's *Read*, *ReadBlock*, *ReadLine*, and *ReadToEnd* methods are used to read the contents of the file.

Using Dynamic Properties

Let's take one more digression and look at dynamic properties. With Windows Forms you have the ability to change the properties of the form at run time, using a configuration file. Figure 18-2 shows the DynamicApp solution open in Visual Basic .NET. Notice that the solution has a file called App.config, and notice also that in the Property Browser, the *Form.Text* property has a small blue icon next to the name. This indicates that the *Form.Text* property is retrieved at run time from the App.config file.

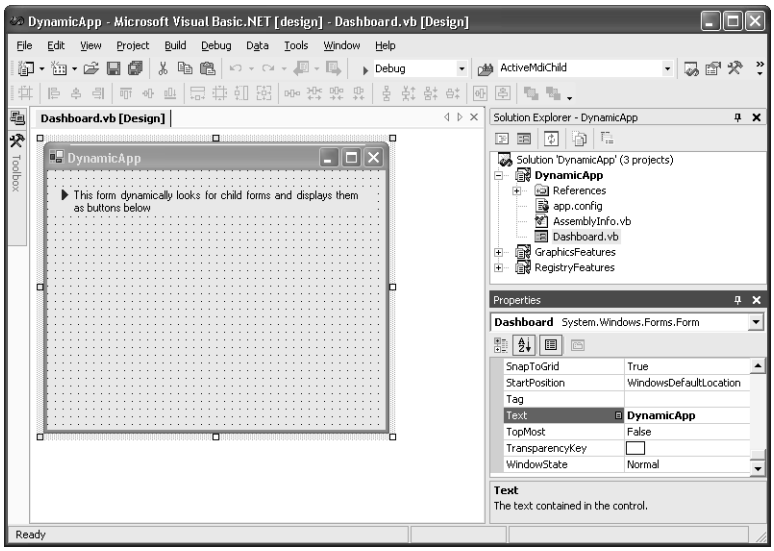


Figure 18-2 The App.config file and the Form.Text dynamic property.

If you double-click the App.config file to open it, you will see that it is an XML file and that it has the following line in the body:

```
<add key="Dashboard.Text" value="DynamicApp" />
```

This is the value of the *Text* property, which is used to set the title bar text of the application. When the project is compiled, this file is deployed alongside the EXE, with the name DynamicApp.exe.config. If you open this file in Notepad and change the setting to something like

```
<add key="Dashboard.Text" value="Hello World" />
```

"Hello World" will show in the title bar the next time the application is run. It is called a **dynamic property** because the value is not stored in the compiled application but instead in an editable XML file that the application reads at run time.

Why is the *Text* property dynamic? Are all properties dynamic? When you create a form, by default none of the properties are dynamic. To make a particular property dynamic, you use the *(Dynamic Properties)* property in the Property Browser; you use the *(Advanced)* builder to select the properties that you want to make dynamic, as shown in Figure 18-3.

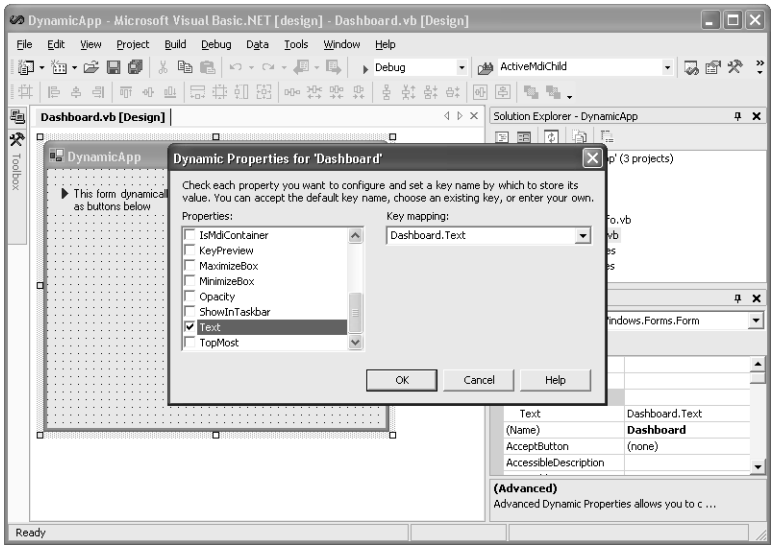


Figure 18-3 Making properties dynamic.

Dynamic properties are valuable for database connection strings, Internet addresses, directory names, and any other property that might need to be changed after the application is compiled.

New Windows Capabilities

Now let's look at what each of the subforms in the sample application does.

Accessing the Registry

Accessing the registry is awkward in Visual Basic 6. You have to use APIs like RegCreateKey and RegDeleteKey, which are cumbersome. The .NET Framework provides a much easier mechanism for accessing the registry. The registry editing form in the RegistryFeatures project (included on the companion CD) shows how to use the .NET Framework classes to create and delete keys and values. Figure 18-4 shows the registry editing form.

Clicking the Create button in the MyRegistryKey Key group box creates the registry key HKEY_CURRENT_USER\Software\MyRegistryKey. Clicking the Delete button deletes this key. Clicking the Create button in the MyRegistryKey\MyValue Value box adds a new string value to the key called MyValue. Clicking the Delete button deletes the value. The contents of the HKEY_CURRENT_USER\Software key are shown in the tree view on the right of the form.

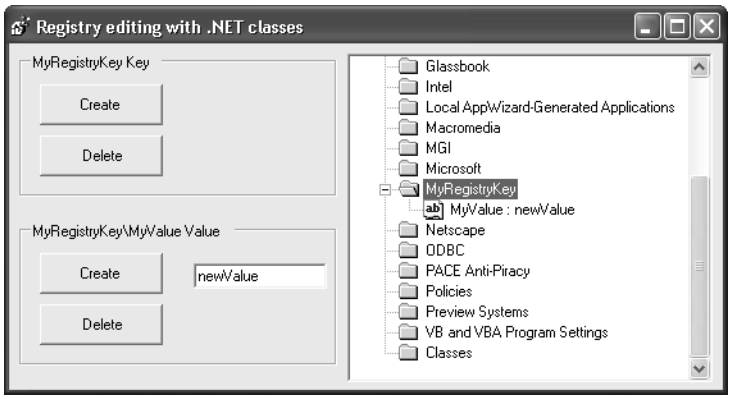


Figure 18-4 Registry editing form.

The following code shows how the form adds the registry key. First it tries to open the registry key, and if the registry key doesn't already exist, the code creates it.

```
Dim rk As Microsoft.Win32.RegistryKey
rk = Registry.CurrentUser.OpenSubKey("Software\MyRegistryKey", True)
If rk Is Nothing Then
   Registry.CurrentUser.CreateSubKey("Software\MyRegistryKey")
Else
   rk.Close()
End If
```

To create a registry key, we first create a Microsoft.Win32.RegistryKey variable called *rk*. We then use *rk* to attempt to open a registry key within *CurrentUser*. The *CurrentUser* property maps to the HKEY_CURRENT_USER registry hive. If the registry key doesn't exist, it is created using the *CreateSubKey* method.

Deleting a key is even simpler. It involves only one line of code.

```
Registry.CurrentUser.DeleteSubKeyTree("Software\MyRegistryKey")
```

This line deletes the MyRegistryKey registry key and any registry subkeys and values it may have.

To add a value, we open the registry key and use the *SetValue* method. In the following example, we add a string value *"MyValue"* with the contents of the newValue.Text TextBox.

```
Dim rk As Microsoft.Win32.RegistryKey
rk = Registry.CurrentUser.OpenSubKey("Software\MyRegistryKey", True)
rk.SetValue("MyValue", CStr(Me.newValue.Text))
rk.Close()
```

If the value already exists, it is overwritten with this new value. To delete the value, use the *DeleteValue* method, as in the following example:

```
Dim rk As Microsoft.Win32.RegistryKey
rk = Registry.CurrentUser.OpenSubKey("Software\MyRegistryKey", True)
rk.DeleteValue("MyValue")
rk.Close()
```

Notice in all these examples that each time we open a registry key, we also close it using the *Close* method. The registry editing form acts upon the HKEY_CURRENT_USER hive. In addition, you can access the HKEY_LOCAL_MACHINE, HKEY_CLASSES_ROOT, HKEY_USERS, and HKEY_CURRENT_CONFIG registry hives using the *Registry* object's *LocalMachine*, *ClassesRoot*, *Users*, and *CurrentConfig* properties. This is certainly much easier than working with the Visual Basic 6 registry APIs.

Control Anchoring

The registry editing form also demonstrates automatic resizing using control anchoring. When you resize the form, the tree view also resizes—without your writing a single line of resize code! Using the *Anchor* property of the TreeView control, you can bind the edges of the control to the edges of the form. The TreeView control is anchored to each side of the form so that when the form resizes, the control also resizes, keeping the top, bottom, left, and right edges the same distance from the edges of the form. This method is much easier than in Visual Basic 6, where you have to write lines and lines of code in the *Form_Resize* event to adjust the size of the control.

Graphics Features

If you enjoy adding some "sharpness" to your applications with graphics, you'll be pleased to know that the .NET Framework has a set of dedicated graphics classes called GDI+, making it easier to add graphics than in any previous version of Visual Basic. We discussed creating gradient backgrounds and drawing shapes in Chapter 15; now we'll look at three other features: opacity, regions, and alpha blending. These features are all demonstrated in the Graphics Effects form in the Graphics Features project on the companion CD. This form has three check boxes and a button. Each one applies a different graphics effect.

The first check box is Form Opacity. Setting this option makes the form semitransparent, as shown in Figure 18-5.

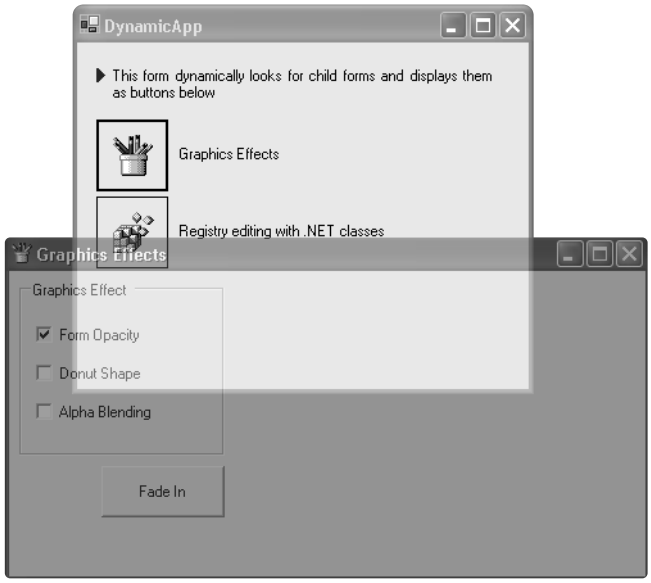


Figure 18-5 Semitransparent form.

The opacity property is a measure of opaqueness. It can take any value between 0 and 1. Setting the property to 1 makes the form solid, setting it to 0 makes the form invisible, and setting it to 0.5 makes the form semitransparent. A fun use of this property is to fade a form from invisible to fully solid. Click the Fade In button on the Graphics Effects Form to see a demonstration of this. Here is the code that fades the form from invisible to solid:

```
Private Sub FadeInButton_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles FadeInButton.Click
   Dim d As Double
   For d = 0 To 1 Step 0.01
      Me.Opacity = d
      Me.Refresh()
      System.Threading.Thread.CurrentThread.Sleep(5)
   Next
   Me.HasOpacity.Checked = False
End Sub
```

The *For…Next* loop increments the variable *d* from 0 to 1. The code then sets the opacity to the value of *d*: 0.00, 0.01, 0.02, 0.03, and so on. The *Refresh* method forces the form to completely repaint, and the *System.Threading.Thread.CurrentThread.Sleep* method waits 5 milliseconds before continuing to fade the form in. While opacity may not be instrumental in solving any real-world business problems, it's a fun effect to add.

The second graphics feature this form demonstrates is regions. Setting the Donut Shape option changes the form from rectangular to a doughnut shape, as shown in Figure 18-6.

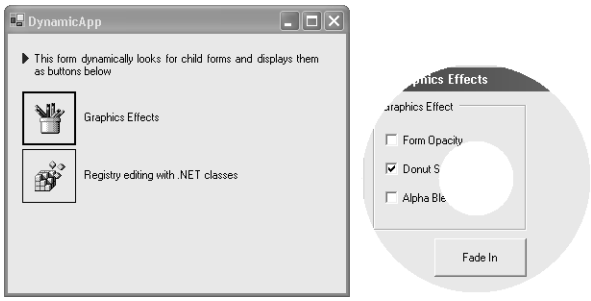


Figure 18-6 Form displayed using the Donut Shape option.

To change the region of a form, create a region from a set of graphics paths and apply this to the form's *Region* property. The following code shows how to do this:

```
Dim myPath As New Drawing2D.GraphicsPath()
Dim bigDiameter As Integer = Me.ClientSize.Height
Dim donutWidth As Integer = 80
myPath.AddEllipse(0, 0, bigDiameter, bigDiameter)
myPath.AddEllipse(donutWidth, donutWidth, _
   bigDiameter - (donutWidth * 2), _
   bigDiameter - (donutWidth * 2))
Me.Region = New Region(myPath)
```

This code creates a *GraphicsPath* variable *myPath*. This variable is used to store one or more vector shapes. The sample above adds two ellipses to the variable. The first ellipsis is the outer ring of the doughnut. The second is the inner ring, the hole of the doughnut. Finally, the *Region* property of the form is set to the newly created path. This shape is a true doughnut—if you click in the middle of the hole, the mouse clicks are sent to the window behind the doughnut form. This technique is very flexible. The number of different graphics paths and different shapes you can create are limited only by your imagination. For example, the following code changes the form's shape to the string "VB.NET Rocks."

```
Dim myPath As New Drawing2D.GraphicsPath()
myPath.AddString("VB.NET Rocks", Me.Font.FontFamily, Font.Bold, _
80, New Point(0, 0), StringFormat.GenericDefault)
Me.Region = New Region(myPath)
```

The third graphics feature this form demonstrates is alpha blending. This technique allows you to paint text or pictures semitransparently. Figure 18-7 shows the effect of setting the Alpha Blending option on the Graphics Effects form.

Figure 18-7 Alpha blending.

Setting the check box enables a timer on the form. In the *Timer.Tick* event, two strings are painted onto the form: a blue *VB.NET* is painted first, and a red *Rocks!!* is painted over the top. Each is painted with a different alpha value that sets the transparency. The alpha value cycles from 0 to 256 and back to 0 each time the *Timer.Tick* event fires. The net effect is that the text on the form fades from *VB.NET* to *Rocks!!* to *VB.NET* again. Here is the code from the *Timer.Tick* event that does the alpha blending:

```
Static alpha As Integer = 0, upDown As Integer = 2
Dim g As Graphics
Dim f As New Font("Arial Black", 60, FontStyle.Regular, _
   GraphicsUnit.Pixel, 0)
Dim tempBitmap As New Bitmap(300, 100)
alpha += upDown
If alpha <= 0 Then
   upDown = -upDown
   alpha = 0
ElseIf alpha >= 255 Then
   upDown = -upDown
   alpha = 255
End If
g = Graphics.FromImage(tempBitmap)
g.Clear(Me.BackColor)
g.DrawString("VB.NET", f, New SolidBrush(Color.FromArgb(255 - alpha, _
   Color.DarkBlue)), New RectangleF(0, 0, 300, 100))
```

(continued)

```
g.DrawString("Rocks!!", f, New SolidBrush(Color.FromArgb(alpha, _
   Color.Red)), 0, 0)
g.Dispose()
g = Me.CreateGraphics
g.DrawImage(tempBitmap, 200, 10)
g.Dispose()
tempBitmap.Dispose()
f.Dispose()
```

The *alpha* and *upDown* variables are static—they retain their old value each time the event is called. Each time the event is called, *alpha* increases by 2 until it reaches 256, and then it decreases by 2 until it reaches 0 and starts all over again. A temporary bitmap called tempBitmap is created for drawing the string. The reason for drawing to a temporary bitmap first is to avoid screen flicker. The two strings are drawn with the opposite alpha values—when *VB.NET* is drawn as solid, *Rocks!!* is drawn as transparent. Valid *alpha* values range from 0 (transparent) to 255 (opaque). Finally, the bitmap is drawn onto the form.

Notice that to draw onto the bitmap or form, we first have to create a graphics object:

```
g = Me.CreateGraphics
g.DrawImage(tempBitmap, 200, 10)
```

This graphics object is the GDI+ graphics drawing surface that supports all of the advanced graphics methods. A new graphics object has to be created for each bitmap or form you want to draw onto. When you finish using the graphics object, it is important to destroy it using the *Dispose* method:

```
g.Dispose()
```

Windows XP–Style Controls

The Windows XP user interface has a new look. The Windows common controls, such as buttons, group boxes, and tabbed dialog boxes, are softer in appearance—they have curved edges and are highlighted with a pleasant orange outline when the mouse moves over them.

Although you can't add or edit Windows XP–style controls using the Visual Basic .NET Windows Forms designer, you can configure your application to use Windows XP controls when it runs. In effect, these are normal controls that simply display differently at run time. The XPTheme sample on

the companion CD shows how to use Windows XP–themed controls, and the following steps describe how to add the Windows XP theme to a new application:

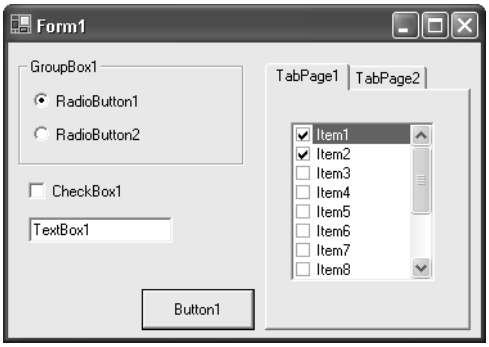
1. Create a new Windows application. Add some controls.
2. For every control that has a *FlatStyle* property, set the *FlatStyle* property to *System*, and Windows XP will use the default system style for the control.
3. Compile your application as usual.
4. Create a manifest file in the same directory as the application's executable (usually the bin directory). A **manifest file** is an XML file that contains run-time information about the application. Using Notepad, create a text file with the following constants:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1"
manifestVersion="1.0">
   <dependency>
      <dependentAssembly>
         <assemblyIdentity
            type="win32"
            name="Microsoft.Windows.Common-Controls"
            version="6.0.0.0"
            processorArchitecture="X86"
            publicKeyToken="6595b64144ccf1df"
            language="*"
         />
      </dependentAssembly>
   </dependency>
</assembly>
```

5. Save the file with the name <*exename*>.manifest. For example, if your application name is MyApp.exe, this file should be called MyApp.exe.manifest. The manifest works on an application-by-application basis rather than a file-by-file basis. Thus, you need to create only one manifest for the main EXE of the application, even if it contains many DLLs. When your application runs, Windows XP looks at the application manifest and uses the information it finds there to alter the application's execution. In this case, the manifest instructs Windows XP to use the latest version of the common controls library—the version with the Windows XP–style controls in it.

Figure 18-8 shows a form with standard controls on the left and the same forms with Windows XP controls on the right.

Default controls



Windows XP themed controls

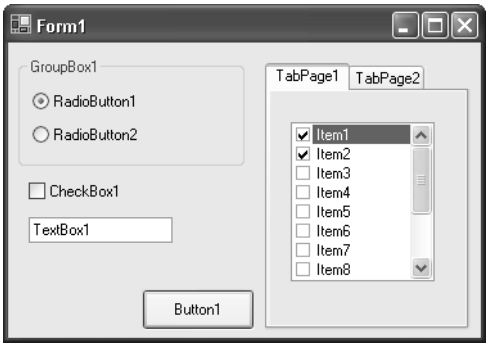


Figure 18-8 Normal controls (above) and Windows XP controls (below).

XCopy Deployment

Do upgraded applications have any limitations? Most upgraded applications contain COM controls and ActiveX references. Most of these controls have to be registered before they can be used. Visual Basic .NET components don't have this limitation—they are self-describing. The information that would go into the registry for a COM component is compiled into the .NET application.

It is often said that .NET components support XCopy deployment (referring to the DOS multiple copy XCopy command). What this means is that you can install any Visual Basic .NET application simply by copying the EXE and any dependent DLLs from a directory on one machine to a directory on another machine (assuming that the .NET Software Development Kit [SDK] is already

installed on the client machine). No registration, no extra installation steps. Plus, the application is isolated—the DLLs it installs do not overwrite DLLs used by other applications.

XCopy deployment is extremely useful for client applications that are to be installed on perhaps thousands of client machines. With Visual Basic 6 applications, it is usually during one of these installations that something goes wrong—your application installs a DLL that overwrites another DLL or another DLL's registration, which then breaks another application. These types of DLL conflicts are frustrating for both you, the developer, and your users.

Visual Basic .NET applications don't have this problem because an installation doesn't overwrite other files (unless you explicitly decide to overwrite an existing file), and other installations don't affect your application's files. However, if your application uses COM components—ActiveX controls or COM libraries—these components must be installed and registered, just as they must be in Visual Basic 6. This means two things. First, for each Visual Basic .NET application that contains COM components you must register the COM components, and second, if one of these COM components is overwritten, your application will stop working. If easy XCopy deployment is important to you, it's a good idea to remove the COM components from your application and replace them with .NET components. This step is especially relevant for applications that will be installed to multiple client machines.

How do you replace COM components with .NET components? In the next chapter, you'll see how you can do this by replacing common ActiveX controls with Windows Forms controls.

Conclusion

This chapter has looked at several ways you can extend your upgraded applications. These are important techniques because most of them would be either impossible or very difficult to achieve in Visual Basic 6. This point underscores one of the important concepts of upgrading: when you move your application to Visual Basic .NET, the real benefit comes when you add value with some of the great features of Visual Basic .NET and the .NET Framework.

The next three chapters look at more ways to add value to your applications. In addition to learning how to replace ActiveX controls with Windows Forms controls, you'll see how to integrate your ADO data access code with ADO.NET data access code, and you'll learn techniques for redesigning distributed applications.