Like the TextAlignment property, the Effect property has a small number of valid settings, which will be identified in the Properties window with descriptive names.

In addition to the custom properties, the FLEXLabel control should also expose the standard properties of a Label control, such as Font, Tag, BackColor, and so on. Developers expect to see standard properties in the Properties window, and you should implement them. The FLEXLabel control doesn't have any custom methods, but it should provide the standard methods of the Label control, such as the Move method. Similarly, although the control doesn't raise any special events, it must support the standard events of the Label control, such as the mouse and keyboard events.

Except for a few custom properties, this control must expose the standard functionality of the Label control. Most of the custom control's functionality exists already, and there should be a simple technique to borrow this functionality from other controls, rather than implementing it from scratch. And this is indeed the case. Visual Basic provides a Wizard, which will generate the code for implementing standard members. The ActiveX Control Interface Wizard will generate the code that implements the standard members, and you only have to provide the code for the custom properties.

Designing a Custom Control

To start a new ActiveX control project, follow these steps:

1. Choose File ➢ New Project.
2. In the New Project window, select the ActiveX Control icon. Visual Basic creates a new project named Project1, which contains a UserControl named UserControl1.

The initial setup for an ActiveX control project is shown in Figure 16.3 (the names in the project window are different, and I'll show you immediately how to change them).

Let's rename the project and the control. These two names will be used to register your control in your system, so they should be meaningful. Follow these steps:

1. Select Project1 in the Project window, and when its properties appear, change the Name property to FLEXLabel.
2. Select UserControl1 in the Project window, and when its properties appear, change the Name property to Label3D.

FIGURE

An ActiveX con
contains a User
instead of a For

TIP

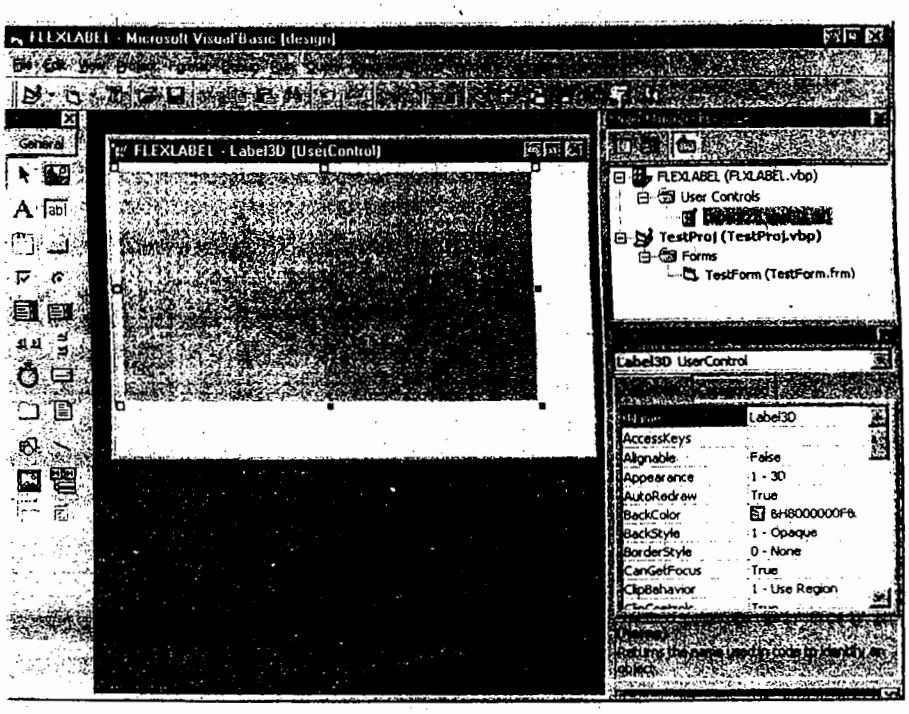


FIGURE 16.3: An ActiveX control project contains a User control instead of a Form.

TIP

Every time you place an ActiveX control on a Form, it's named according to the UserControl object's name and a sequence digit. The first instance of the custom control you place on a Form will be named Label3D1, the next one will be named Label3D2, and so on. Obviously, it's important to choose a meaningful name for your UserControl object.

A new object was just introduced: the *UserControl* object. As you will soon see, the UserControl is the "Form" on which the custom control will be designed. It looks, feels, and behaves like a regular VB Form, but it's called a UserControl. UserControl objects have additional unique properties that don't apply to a regular Form, but in order to start designing new controls, think of them as regular Forms. Whereas the FLabel application aligns and displays the caption on a Form, the Label3D custom control uses the same code to draw the caption on the UserControl object (we'll simply change all instances of "Me" to "UserControl").

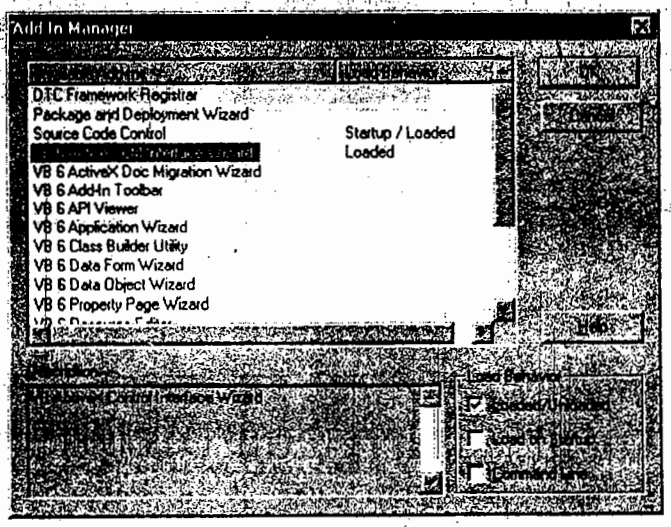
You've set the scene for a new ActiveX control. Before we insert even a single line of code, we'll let a Wizard generate as much of the custom control as it can for us. Among other things, it will design the control's structure.

The ActiveX Control Interface Wizard

Now, choose Add-Ins ▷ ActiveX Control Interface Wizard to open the Wizard. This Wizard will guide you through the steps of creating a new ActiveX control. It will create the control's interface (its properties, methods, and events) and prepare the basic code for it. You will have to provide only the code that draws the caption and a few more features that can't be automated. The Wizard will take care of the bulk of the work for you. The ActiveX Control Interface Wizard has six windows, which are explained next.

If the ActiveX Control Interface Wizard doesn't appear in the Add-Ins menu, select Add-In Manager, and in the Add-In Manager dialog box, double-click the entry VB6 ActiveX Control Interface Wizard. The indication *Loaded* will appear in the Load Behavior column next to the Wizard's name, as shown in Figure 16.4. If you want the Wizard to load every time you start Visual Basic, check the Load on Startup checkbox. Now, click the OK button, and the add-in's name will appear in the Add-In menu.

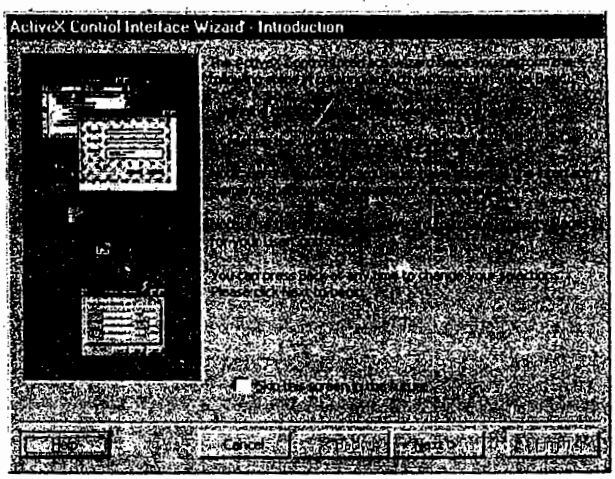
FIGURE 16.4:
The Add-in Manager dialog box



The ActiveX Control Interface Wizard will guide you through the steps of designing the skeleton of a custom ActiveX control. The Wizard will not add the code to align or even display the caption, but it will create a functional control with most of the standard members of a typical ActiveX control.

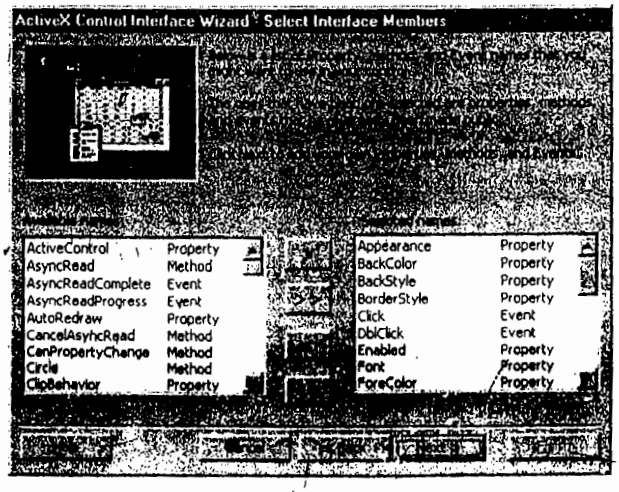
Introduction

The first screen of the Wizard is a welcome, and you can disable it in the future by checking the Skip This Screen in the Future checkbox. Click the Next button to start the design of your control's user interface.



Selecting Interface Members

The next window of the Wizard prompts you to select the standard members of the control. On the left is a list of standard properties, methods, and events that you can include in your custom control. On the right is a list of common members of the user interface, already selected for you. Visual Basic suggests that your control should support these members.

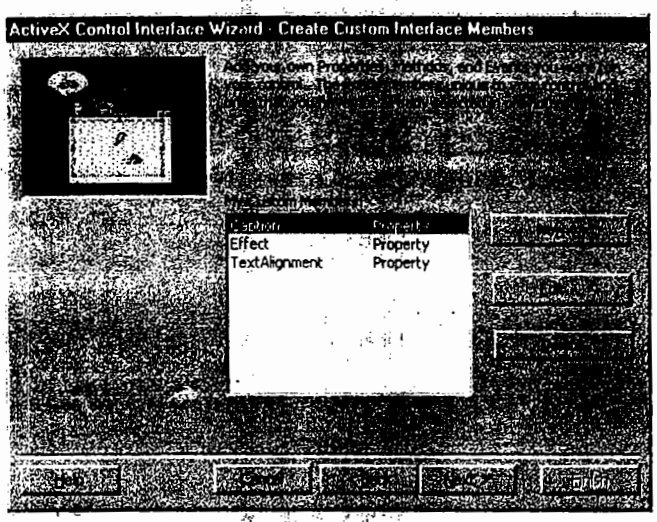


To add a new member, select it from the left list, and click the button that has a single right-pointing arrow. Add the following members to the custom control:

Appearance	Font	MouseMove	OLEGiveFeedback
BackColor	ForeColor	MousePointer	OLESetData
BackStyle	HDC	MouseUp	OLEStart Drag
BorderStyle	HWnd	OLEDrag	Picture
Click	KeyDown	OLEDragDrop	Resize
DblClick	KeyPress	OLEDragOver	
Enabled	KeyUp	OLEDropMode	

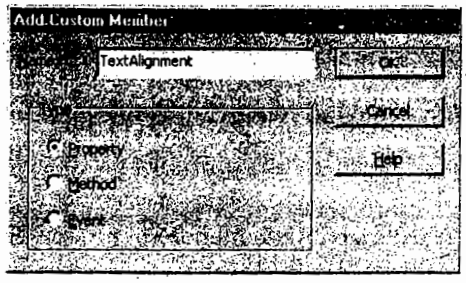
As you can see, I've included all the standard properties, events, and methods you'd expect to find in any control that's visible at runtime. I've omitted the data-bound properties, but the FLEXLabel control won't be connected to a database field. Click the Next button.

Creating Custom Interface Members



In this window, you add the properties, events, and methods that are unique to your custom control. Follow these steps:

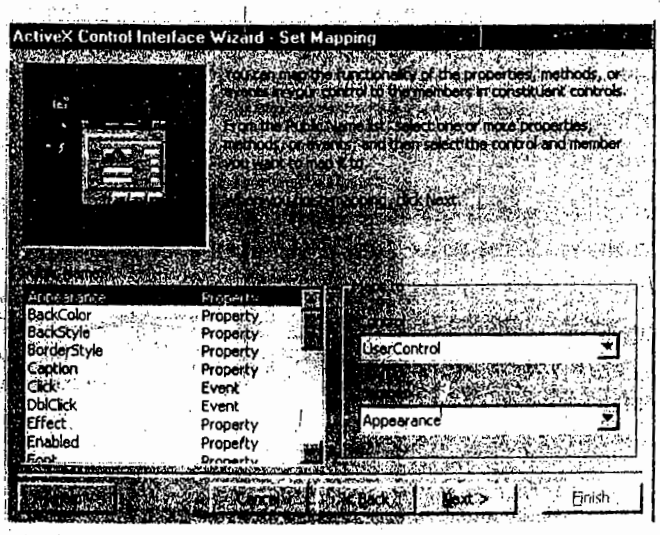
1. Click the New button to display the Add Custom Member dialog box.



2. In the Name box, enter **Caption** (which is the name of a property), and select the Property radio button.
3. Click OK. The name of the first custom property is now displayed in the My Custom Members box of the Create Custom Interface Members window.
4. Repeat steps 1 through 3 to add the TextAlignment and Effect properties.

If you have misspelled any of the property names or if you want to delete one and re-enter it, use the Edit and Delete buttons. When you are done, click the Next button.

Setting Member Mapping



You use this window to map certain properties of your custom control to properties of the so-called constituent controls. A *constituent control* is a regular VB control that your custom control uses. Suppose your custom control contains a Label

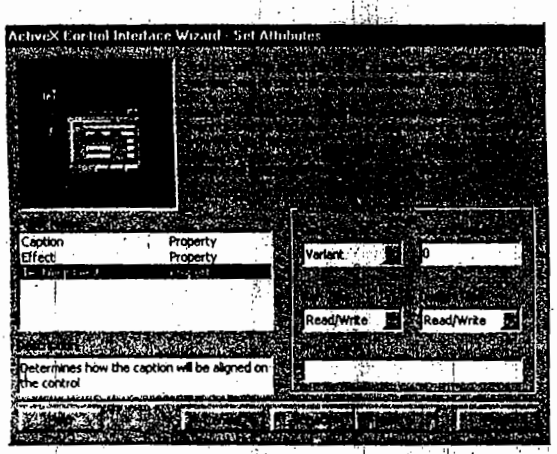
control, and you want the label's background color to become your control's background color (or foreground color, for that matter). In this window, you map the properties of the controls used by your custom control to the properties of any of the controls it uses. When calling one of the mapped members, the user thinks he or she is setting a control's properties, but in reality is setting a constituent control's member.

You must map all the members (properties, methods, and events) to the UserControl object (except for the custom members, of course). When the user clicks the control, for instance, the Click event is passed to the host application, as if it were generated by the custom control. Any properties, methods, or events that you don't want to handle with your own code must be mapped to the UserControl object. The Click event is a typical example. There's no action you want to take from within your control's code when it's clicked with the mouse. The event must be reported to the host application to handle it accordingly.

To map properties, follow these steps:

1. From the Public Name list, select a property or an event.
2. Click the Control drop-down list's down arrow, and select UserControl. The Wizard immediately selects the UserControl member with the same name.
3. Map all members of the new control (except for custom members) to the equivalent members of the UserControl object.
4. Click the Next button.

Setting Attributes



In this window, you set the attributes of the new members (or change the attributes of some default members, if you want, but this isn't recommended). Members that have already been mapped will not appear in this list. The Wizard has declared all new properties as variants, because it can't decipher their types from their names. Our TextAlignment and Effect properties, however, are integers, and the Caption property is a string.

To set attributes, follow these steps:

1. In the Public Name box, select TextAlignment.
2. Click the Data Type drop-down arrow, and select Integer.

Notice that you can set an initial value too. Set the TextAlignment property's initial value to 4 (which displays the caption in the center of the control).

3. In the Default Value box, enter the value 4. This is the value that is displayed by default in the Properties window for the FLEXLabel control. You will see later in the chapter how to assign custom data types to your properties.
4. Repeat steps 1 through 3 for the Effect property. Set its data type to Integer and its initial value to 2 (carved).

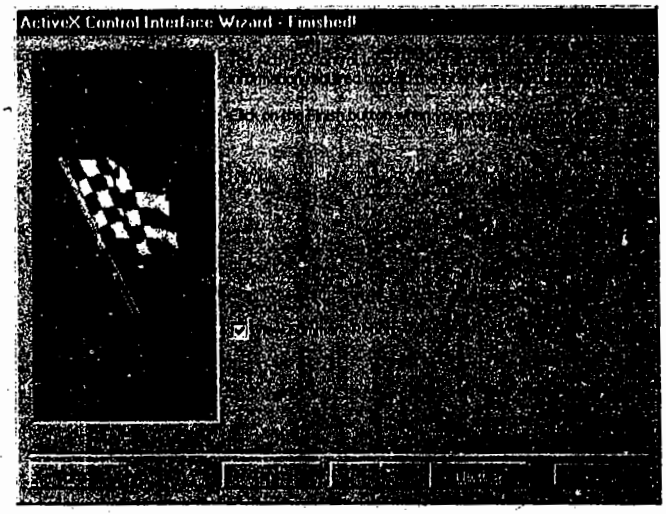
TIP

Don't forget to supply a short description of each property in the Description box. These descriptions are displayed below the Properties window when the user selects a property. The standard members have a description already, but you must supply descriptions for your custom properties.

In the Arguments box, you supply the arguments of the events. This project doesn't have any custom events, but you'll see examples of custom events later in the chapter.

Notice that the properties can have different design-time and runtime properties, which is the first really unique characteristic of a user control. If you think that the developer has no reason to change the alignment or the effect at runtime, make these properties Read-Only at runtime.

Finishing



In this window, you are prompted to click Finish to generate the control. Leave View Summary Report checked.

Your hard disk will spin for a few seconds, and you will see the summary report. Read the summary report (which basically tells you how to proceed), and close or save the editor window.

You have created the FLEXLabel control. Not that it does much (you haven't entered the code for displaying the caption yet), but you are ready to test its behavior as a control. At this point, your control has its own Properties page, and you can place it on your Forms.

There's nothing on the screen except a Form. This is the UserControl object, which for all intents and purposes is a Form. There's also a new icon in the toolbox. This is your new control's icon. The icon of a matrix with a pen on top is the default icon for all ActiveX controls. To change the icon, design a new bitmap and assign it to the UserControl object's ToolboxBitmap property. Let's first make sure the control works, and then we'll customize it. The ActiveX control's icon is disabled because you are still designing the control. The control won't be activated while it's being designed.

Testing Your New ActiveX Control

To test your new control, you would normally have to create an OCX file, start a new project, and then place the new custom control on a Form. To simplify development, Visual Basic lets you add a new project to the existing project. In this way, you can test

```
Public Property Let Caption(ByVal New_Caption As String)
    m_Caption = New_Caption
    PropertyChanged "Caption"
End Property
```

Each property is defined by two Public procedures:

- Property Let
- Property Get

The Property Let procedure is invoked every time the property is changed, either via the Properties window (at design time) or via code (at runtime). The code that's executed when a property changes value consists of two lines. The first line gets the value supplied by the procedure's argument (which is the new value of the property) and assigns it to the private variable that represents the property in the control. The rest of the code sees only the *m_Caption* local property, not the actual property. The second line notifies Visual Basic that the property has changed value. The PropertyChanged method is important and must be included in the Property Let procedure, because this is how Visual Basic saves any changes made to the property at design time so that it will take effect at runtime.

The Property Get procedure is invoked every time the program recalls the value of the property. This procedure reads the value of the *m_Caption* private variable and assigns it to the Caption property. There must be a Property Let and a Property Get procedure for each property, and they must include the lines shown here. They represent the minimum functionality of the mechanism for setting and reading property values.

Of course, you can add validation code here too. The TextAlignment property's value must be in the range 0 through 9. As is, the custom control allows the user to enter any value in the Properties window for this property. Let's add some validation code in the Property Let procedure of the TextAlignment property. The validation code is simple: It rejects any values that are smaller than 0 or larger than 8.

Code 16.4: Validation Code for the Property Let Procedure

```
Public Property Let TextAlignment(ByVal New_TextAlignment As Integer)
    If m_TextAlignment >=0 And m_TextAlignment<=8 Then
        m_TextAlignment = New_TextAlignment
        PropertyChanged "TextAlignment"
    Else
        MsgBox "Invalid value for this property"
    End If
End Property
```

3. Check the FLEXLABEL checkbox, and click OK. The custom control's icon will appear on the project's toolbox. Notice that the name of the control is the same as the project's name.

If you use this icon to place a FLEXLabel control on a Form, Visual Basic automatically names it Label3D1 (if it's the first control on the Form; the second will be named Label3D2, and so on).

The custom control has been registered with the system Registry on your computer, but how about other computers? When you distribute an application that uses the custom control, the new control must be installed on the host computer before your application can use it. To install a custom control to another system, use the Regsvr32 utility, passing the name of the OCX file as an argument. Assuming you have copied the FLXLABEL.OCX file to the Windows\System folder, use the following command from the DOS prompt to install it on the host system:

```
REGSVR32 FLXLABEL.OCX
```

If the OCX file resides in another folder, switch to its folder and issue this command:

```
C:\WINDOWS\SYSTEM\REGSVR32 FLXLABEL.OCX
```

To uninstall a custom control, use the REGSVR32 program with the /U option. The following command will uninstall the FLXLABEL control:

```
C:\WINDOWS\SYSTEM\REGSVR32 FLXLABEL.OCX /U
```

Interacting with the Container

ActiveX controls are meant to be used as building blocks in application development. As such, they are sited on Forms or other controls that can act as containers. As an ActiveX control designer, you should be able to access the properties of a control's container and adjust the control's appearance according to the properties of the container. You will find two objects useful in designing custom controls: Extender and Ambient.

The Extender Object

The Extender object provides some of your control's basic properties, such as its Name, Width, and Height. These are the properties that are maintained by Windows itself, and they don't trigger a Property Let procedure when they are set. As you know, the control's Name property can be changed at any time while the control is used in design mode, but there are no Property procedures for this property. To find out the name assigned to the control by the user, you must call the Extender

object's Name property. The Extender object is also your gateway to certain properties of the *parent control*, the control on which the custom control is sited.

The Name Property You can find out the Name of the container control and its dimensions. To access the Extender object, use the following expression:

```
UserControl.Extender.extProperty
```

The *extProperty* entry is an Extender property name. The name of the custom control is returned by the following expression:

```
UserControl.Extender.Name
```

NOTE But do I really have to invoke the Extender object to find out the custom control's name from within the custom control? Isn't this overkill? If you think about it, the control doesn't know its own name! The user can set the control's Name property at any time during the control's design, and to read this name from within the control's code, you must indeed call upon the services of the Extender object. There are no Property Let and Property Get procedures for the Name property.

The Width and Height Properties These properties return the control's dimensions, as specified by the user. To find out the control's dimensions, use the expressions:

```
UserControl.Extender.Width
```

and

```
UserControl.Extender.Height
```

The Tag and Index Properties These are two more properties that Visual Basic maintains for you. Tag and Index aren't properties of the Extender object (although the syntax indicates that they are). They are properties of your custom control that can't be accessed directly. I didn't include any code for maintaining these properties, but they appear in the control's Properties window anyway. Because their values are maintained by Visual Basic, you can't access them directly; you must go through the Extender object.

The Parent Property This property returns the object on which your control is sited. The UserControl.Extender.Parent object is one way of accessing the container control's properties. To find out the container control's dimensions, you can use the following statements:

```
PWidth = UserControl.Extender.Parent.Width
PHeight = UserControl.Extender.Parent.Height
```

You can use similar statements to read the container control's name (UserControl.Extender.Parent.Name), it's background (UserControl.Extender.Parent.BackColor), and so on.

TIP

Notice this important difference: UserControl.Extender.Name is the custom control's name (for example, Label3D1); UserControl.Extender.Parent.Name is the container's name (for example, Form1).

To experiment with a few of the Extender object's dimensions, insert the following lines in the FlxLabel UserControl's Click event, run the test Form, and click the FLEXLabel control.

Code 16.8: Accessing the Extender Object

```
Private Sub UserControl_Click()
Dim ExtProp As String
    ExtProp = "I'm a custom control. My name is " _
              & UserControl.Extender.Name
    ExtProp = ExtProp & "I'm located at (" & UserControl.Extender.Left _
              & ", " & UserControl.Extender.Left & ")"
    ExtProp = ExtProp & vbCrLf & " My dimensions are " _
              & UserControl.Extender.Width & " by " _
              & UserControl.Extender.Height
    ExtProp = ExtProp & vbCrLf & "I'm tagged as " _
              & UserControl.Extender.Tag
    ExtProp = ExtProp & vbCrLf & "I'm sited on a control named " _
              & UserControl.Extender.Parent.Name
    ExtProp = ExtProp & vbCrLf & "whose dimensions are " _
              & UserControl.Extender.Parent.Width _
              & " by " & UserControl.Extender.Parent.Height
    MsgBox ExtProp
    RaiseEvent Click
End Sub
```

You will see the message box shown in Figure 16.7 (you must assign a value to the Tag property, which is by default empty).

NOTE

This message is displayed from within the custom control's Click event, not from the test application. Only after the message is displayed does the test application receive the Click event. If you have programmed the control's Click event in the test application as discussed earlier in the chapter, you will see two message boxes. The first one is displayed from the UserControl's code; the second is displayed from the application's code.

FIGURE 16.7:
Use the Extender object to access certain properties of the custom control and its container.

The Default and Cancel Properties These two properties determine whether the custom control can be used as the Default (Extender.Default is True) or the Cancel (Extender.Cancel is True) control on the Form. These are two more properties you can't set from within your code. You can read them only while the control is used in the design of another application.

NOTE The DefaultCancel property of the UserControl object determines whether one of the Default and Cancel properties of the control can be set, and not whether the control will be the Default or Cancel control on the Form. If DefaultCancel is True, properties Default or Cancel will appear in the control's Properties window, and you can find out the values of these properties through the Extender object.

The Ambient Object

The Ambient object is similar to the Extender object, in that it provides information about the custom control's environment, and the two actually overlap in some ways. The Ambient object gives your control's code hints about the control's environment, such as the container's background color, its font, and so on. The single most important property of the Ambient object is UserMode, which indicates whether the control is operating in design or runtime mode.

As you know from working with regular controls, all VB controls operate in design and runtime modes. Because our control behaves identically in both modes, there's no need to distinguish between the two. In designing custom ActiveX controls, however, you frequently need to differentiate between the two modes from within your code and react to certain events differently, depending on whether the control is being used in design-time or runtime mode.

The UserMode Property This property is True when the control is operating in runtime mode and False when the control is operating in design mode. To see how this property works, let's display the text "Design Mode" while the control is in design mode. Open the Paint event's subroutine, and insert the following lines at the end of the Paint event's handler.

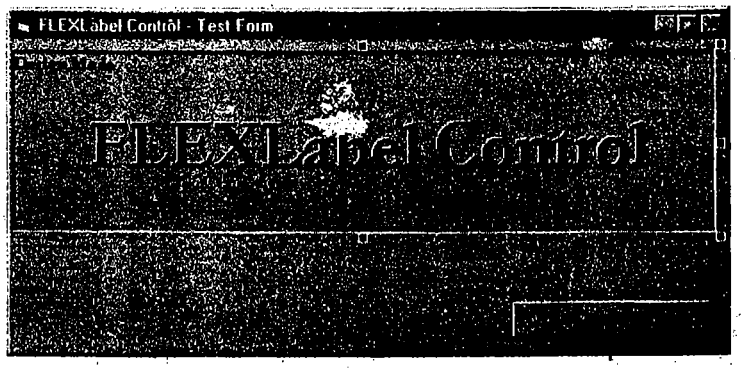
Code 16.9: Using the UserMode Property

```
Private Sub UserControl_Paint()
    DrawCaption
    OldFontSize = UserControl.Font.Size
    UserControl.Font.Size = 10
    If Not Ambient.UserMode Then
        UserControl.CurrentX = 0
        UserControl.CurrentY = 0
        UserControl.Print "Design Mode"
    End If
    UserControl.Font.Size = OldFontSize
End Sub
```

These statements determine whether the control is being used in design or runtime mode. In design mode, the string "Design Mode" is displayed in its upper left corner, as shown in Figure 16.8. The statements that save and restore the font size are needed, because without them either the "Design Mode" string will be displayed in a large font (same as the caption), or the caption will be printed in the same (small) size as the string. The code sets the font size to a small value, displays the "Design Mode" string at the upper left corner of the control, and then restores the original font size.

FIGURE 16.8:

The "Design Mode" string is displayed in the control's upper left corner when it's open in design mode.



The ForeColor, BackColor Properties These two properties report the foreground and background colors of the container. Use them to initialize the equivalent properties of your custom control.

The Font Property This property reports the font used by the control's container. All controls inherit the font settings of the container on which they are

sited, the moment they are created. If the Form's font is changed after your custom control has been sited on it, the control's Font setting won't be affected.

The ShowGrabHandles Property This property lets you hide the grab handles that are used to resize the control at design time.

The ShowHatching Property If, under certain circumstances, you want to disable the control, set this property to True to draw a hatch pattern over the control, which will indicate that some other action must take place before the control can be manipulated.

The AmbientChanged Event Besides its properties, the Extender object provides the AmbientChanged() event, which informs your code about changes in the Ambient object. This event is recognized by the UserControl object and its declaration is:

```
Private Sub UserControl_AmbientChanged(PropertyName As String)
```

The AmbientChanged() event has a single argument, which is the name of the property that changed. Use this event to stay on top of changes in the control's container and to adjust the control's appearance or behavior accordingly.

To test this event, insert the following line in the AmbientChanged() event of a custom control (use the FLEXLabel control, if you haven't designed any other yet):

```
Private Sub UserControl_AmbientChanged(PropertyName As String)
    Debug.Print "The property " & PropertyName & " changed"
End Sub
```

Then switch to the test Form and change a few of the container's properties. Every time an Ambient property changes value, a message will be displayed in the Immediate window. Notice that only a few of the container's properties trigger the AmbientChanged() event. They are the ones that can affect the appearance of the control on the Form. If you change the Form's Name or Caption property, no AmbientChanged() event is raised in the UserControl object.

Designing Property Pages

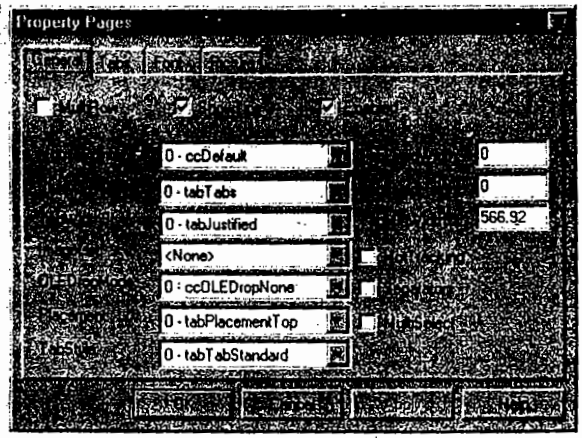
ActiveX controls can also have Property Pages. Property Pages are similar to the Properties window, in that they allow the developer (the programmer who is using your control in a VB project) to set the control's properties. Unlike the Properties window, however, the Property Pages offer a better and more flexible user interface, including instant visual feedback, for setting up a control.

Property Pages are basically design elements. Figure 16.9 shows the Property Pages for the TabStrip control. Through this interface, you can set up the TabStrip

FIGURE
The Property
TabStrip cor

control in ways that are simply impossible through the Properties window. The properties you can set through the Properties window apply to the entire control, and you can't set the titles and appearance of the individual tabs, their number, and so on. The Property Pages for this control contain several pages (General, Tabs, Font, and Picture) on which related properties are grouped.

FIGURE 16.9:
The Property Pages for the TabStrip control



The design of Property Pages is greatly simplified by (what else?) the Property Page Wizard. Let's add some Property Pages to the FLEXLabel control. The control has three custom properties—two enumerated properties and a string property—which we'll place on the same page. In addition, the control has several standard properties, which will be placed on different property pages. The Font property, for example, must appear on its own Property Page, which is similar to the Font common dialog box. Similarly, the Color property must appear on its own property page, which resembles the Color common dialog box.

Using the Property Page Wizard

To use the Property Page Wizard, follow these steps:

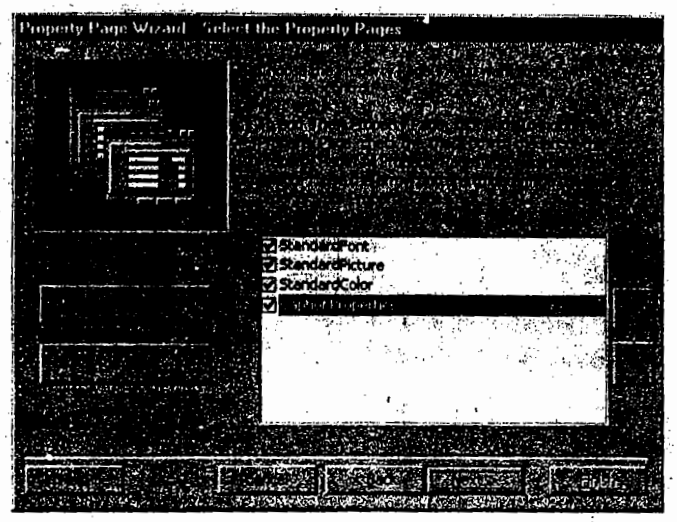
1. Open the FlxLabel project if it's not already the active project.
2. Choose Add-Ins ➢ Property Page Wizard to open the Add Property Page dialog box.
3. Select VB Property Page Wizard.

The Property Page Wizard will take you through the steps of setting up the control's property pages.

Introduction

This is an introductory window, which you can skip in the future by checking the Skip This Screen in the Future checkbox.

Selecting the Property Pages



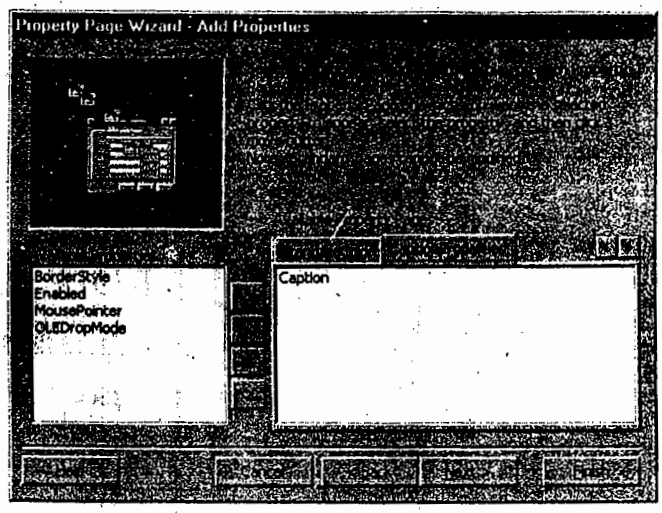
In this window, you select the Property Pages for your control. This window contains some (or all) of the standard Property Pages:

- **StandardFont**, which allows you to set the font
- **StandardPicture**, which allows you to set Picture properties
- **StandardColor**, which allows you to set color

For our control's Property Pages, we need the StandardColor page, plus a custom one, which we can add by clicking on the Add button. The Wizard prompts you to enter the name of the custom page. Enter **TextProperties**, and then click the Next button.

Can you see why the Wizard selected the StandardColor page for us? Our custom control provides the BackColor and ForeColor properties, which are set through the StandardColor property page. If you don't want a Property Page for the color-related properties, clear the checkbox that precedes the property name. But we do need the page for specifying color, so leave this page checked. You can also rename the pages with the Rename button.

Assigning Properties to the Property Pages



In this window, you specify the properties to be displayed in each Property Page. The Wizard has already assigned the color-related properties to the StandardColor page, the Font property to the StandardFont page, and the Picture property to the StandardPicture page. The custom properties, however, have not been assigned to any page, because the Wizard doesn't know where they belong. To add the Caption property to the Text Properties page, follow these steps:

1. Select the Text Properties tab.
2. Select the Caption property from the list on the left, and then click the single right-arrow button to add it to the Text Properties page.
3. Click Next to display the last page of the Property Page Wizard, and then click Finish.

If you examine this window of the Wizard carefully, you'll probably notice something strange. Not all of the custom properties appear in the Available Properties list. Instead, some other property names appear in the list. The Wizard can't handle properties of custom type. The TextAlignment and Effect properties are custom data types (the Align and Effects enumerated types we declared in the code), so they are omitted. If you want a Property Page on which the developer can specify the appearance of the control, you must add the TextAlignment and Effect properties to the Text Properties page. Unfortunately, you can't add these properties through the Wizard; you'll have to do it manually.

But first, let's see what the Wizard has done for us. Follow these steps:

1. Switch to the test Form, and right-click on the FLEXLabel control
2. From the shortcut menu, choose Properties to display the two Property Pages shown in Figures 16.10 and 16.11.

FIGURE 16.10:
The Color Property Page of the FLEXLabel control

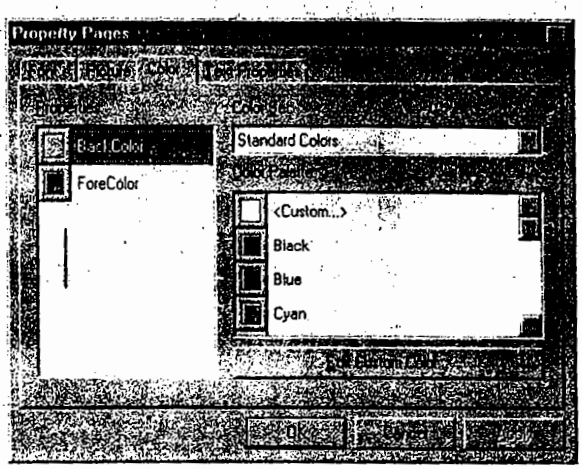
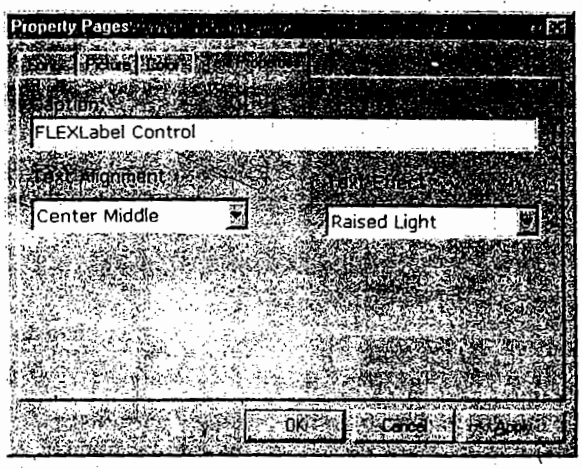


FIGURE 16.11:
The Text Properties Property Page of the FLEXLabel control



The Color page looks fine, but the Text Properties page needs drastic improvement. The Wizard just dumped a Label and a TextBox control on it, and you should not only fix their appearance, but provide the actual code as well.

The Color tab has a Properties list that contains the names of the color-related properties. Had you created more color properties for the custom control, they

within the Timer control's Timer event, you must update the display and test whether the alarm should go off.

Code 16.16: The Alarm Control's Timer Event

```
Private Sub Timer1_Timer()
Dim TimeDiff As Date
Dim StopNow As Boolean

    If Time - m_AlarmTime > 0 Then
        If NextDay = False Then
            StopNow = True
        Else
            TimeDiff = 24 - Time + m_AlarmTime
        End If
    Else
        If NextDay = True Then
            StopNow = True
        Else
            TimeDiff = m_AlarmTime - Time
        End If
    End If
    If m_CountDown Then
        Label1.Caption = Format$(Hour(TimeDiff) & ":" & _
            Minute(TimeDiff) & ":" & Second(TimeDiff), "hh:mm:ss")
    Else
        Label1.Caption = Format(Hour(Time - startTime) & ":" & _
            Minute(Time - startTime) & ":" & _
            Second(Time - startTime), "hh:mm:ss")
    End If

    If StopNow Then
        Timer1.Enabled = False
        RaiseEvent TimeOut
    End If
End Sub
```

The logic for stopping the timer and invoking the TimeOut event depends on whether the control is set to count down. When counting down, it displays the time remaining; when counting up, it displays the time elapsed since the timer started (the *startTime* variable set by the StartTimer method) and stops when the AlarmTime is reached. This condition is detected with the last If structure in the code. The variable *NextDay* is set in the StartTimer method. When it's time for the alarm to go off, the *NextDay* variable will change value, an event that signals that the alarm must go off.

You may think of detecting the TimeOut event by comparing the AlarmTime with the current time, with a statement such as the following:

```
If Time = AlarmTime Then
    Timer1.Enabled = False
    RaiseEvent TimeOut
End If
```

This code will not always work. If the computer is too busy when it's time for the alarm to go off (starting another application or scanning the hard disk, for example), the Timer event for the last second may not be triggered. If this event is skipped, you'll have to wait for another 24 hours before you get the next time out (and then you may not get it again!). The implementation I've chosen for the example will set off the alarm, even if this happens a second or two later.

See how simple it is to generate your own events? Simply call the RaiseEvent method from within your code, and Visual Basic sees that the event is reported to the host application. Any condition in your application can trigger an event at any time. In addition, you must insert the declaration of the event, along with the variable declarations, at the beginning of the code:

```
Event TimeOut()
```

Open the Alarm project and examine the code of the UserControl object, as well as the code of the test project. The code is straightforward, and that's why I've chosen to implement this control manually. It's usually easier to implement custom controls with the help of the ActiveX Control Interface Wizard, but it's important that you understand what goes on and what the Wizard does for you.

FIGUR

The CLDesi exhibits ru at design t

Enhancing Existing Controls

You can also develop custom ActiveX controls that enhance existing controls. There's not a single user who wouldn't like to add "new" features to existing controls. Many programmers add new features to standard ActiveX controls with the appropriate code from within their applications. A shortcoming of the ComboList control, for instance, is that any new entries added to its Edit box are not appended to the list of options. In other words, the ComboBox control lets you specify a new entry, but this entry isn't automatically added to the control's list of options. Many programmers capture the Enter keystroke in the ComboBox control and append new entries to the control's list manually.

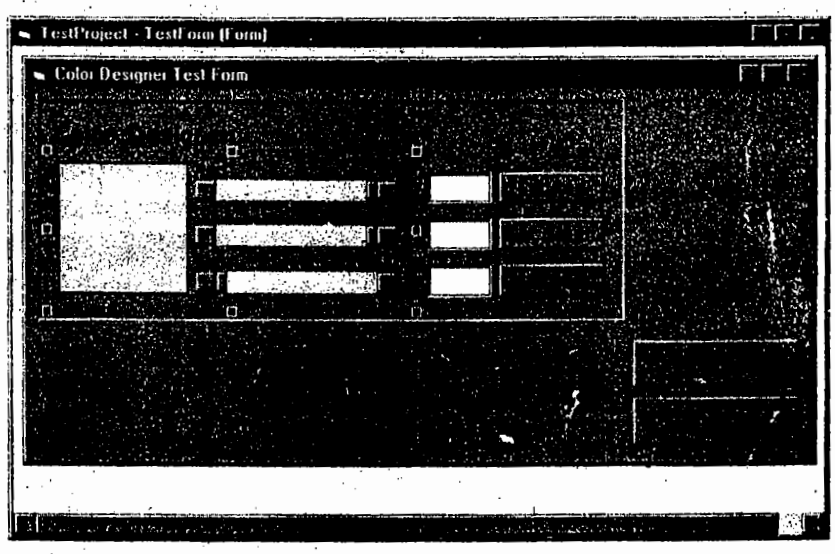
I'm sure each of you would like to add your own little feature to the standard VB controls. In this section, you'll learn how to enhance existing ActiveX controls

and package them as custom controls. Obviously, you can't change the basic functions of an existing control, as you don't have access to their code. But there's a lot you can do by adding some code and packaging the existing control along with your code as a new control.

In this section, I'm going to enhance the TextBox control. The CTextBox custom control is a regular TextBox control that changes its default behavior depending on whether it accepts an optional or a required field.

In this chapter's folder on the CD, you will also find the CLDesign project, which demonstrates how to add runtime characteristics to a control at design time. Did you ever try to set the value of a Scrollbar control at design time by scrolling its button? Most of us have tried this at one time or another, and the result was to move the entire control on the Form. The ScrollBar control's value can't be set visually at design time. An interesting enhancement to the ScrollBar control would be to provide a mechanism that would allow the developer to specify whether the control should be moved or operate as it would at runtime, even though it's in design mode. The CLDesign project is straightforward, and I will not discuss it in this chapter. All the information you need to make a control exhibit runtime behavior at design time is the *EditAtDesignTime* property of the UserControl object. If you set this property to True, the Edit command will be appended to its shortcut menu. Right-click the control on the test Form and select Edit; the control will behave as if it were in runtime mode. If it contains a Scrollbar control, for example, you can adjust the control's value with the mouse. The CLDesign control, shown on a Form in Figure 16.17, lets the developer select the initial color at design time by sliding the three scrollbars.

FIGURE 16.17:
The CLDesign project exhibits runtime behavior at design time.



An Enhanced TextBox Control

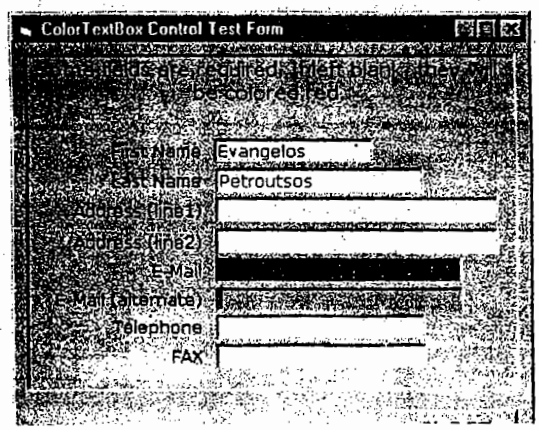
The CTextBox control is a custom ActiveX control that enhances the operation of the standard TextBox control. It's a simple control (it inherits all of the TextBox control's functionality), but I found it very useful in designing data-entry applications. As you will see, the enhancements are quite simple and really trivial to implement.

Most of you have developed data-entry screens in which some fields are required, and you may have designed Forms with TextBoxes colored differently, depending on whether the corresponding field is required. Or you may have used the required field's LostFocus event to keep the focus in the required field until the user enters a value.

How about an enhanced TextBox control that changes color after a value is entered? Figure 16.18 shows a data-entry Form using the enhanced TextBox control. The fields First Name, Last Name, and E-Mail are required, and the corresponding TextBox controls are initially colored red. If the user moves the focus without entering a value in these fields, they remain red. If a value is entered, their color changes to white. Another feature of the enhanced TextBox control is that it changes color when it receives the focus so that the user can quickly spot the active control on the Form. As you can probably guess, I got the idea from data-entry Forms on Web pages, which use an asterisk to indicate the required fields.

FIGURE 16.18:

The CTextBox control enhances the standard TextBox control by adding a few custom properties.



VB6 at Work: The CTextBox Project

Open the CTextBox project on the CD and run it. When you open the project for the first time, you'll see an error message, indicating that the CTextBox control can't be loaded. Continue loading the project, and then open the project's test

Form. All instances of the CTextBox control are replaced by PictureBoxes. Delete the PictureBox controls on the Form, and create an array of seven CTextBox controls. Place them on the Form as shown in Figure 16.18, aligning them with their corresponding captions and with one another. Then run the project by pressing F5, and check out the functionality of the new control.

The CTextBox Control's Specifications

The design of the CTextBox control is fairly simple. It's identical to the standard TextBox control, and it provides a few additional properties.

EnterFocusColor When the control receives the focus, its background color is set to this value. If you don't want the currently active control to change color, set its EnterFocusColor to white.

LeaveFocusColor When the control loses the focus, its background color is set to this value (this property is usually white for optional fields and has the same value as the MandatoryColor for required fields).

Mandatory This property indicates whether the control corresponds to a required field, if Mandatory = 1 (Required), or an optional field, if Mandatory = 0 (Optional).

MandatoryColor This is the background color of the control if its Mandatory property is 1 (Required). The MandatoryColor overwrites the LeaveFocusColor setting. In other words, if the user skips a mandatory field, the corresponding control is painted with the MandatoryColor and not with the LeaveFocusColor. Notice that required fields (those with Mandatory=1) behave like optional fields after they have been assigned a value.

To understand how these properties are used in the design of a data-entry Form, open the CTextBox project and experiment with the settings of its custom properties to see how they affect its operation. Because the CTextBox control is not a standard element of the Windows interface, your users may not understand what the changing colors mean, but it won't take long for anyone to get the hang of it and use this feature efficiently.

Designing the CTextBox Control

The design of the CTextBox control is straightforward. We'll use the ActiveX Control Interface Wizard to design a custom control that has all the members of the standard TextBox control (except for the properties that relate to data binding). The Wizard will create the source code for us, and we'll add a few statements that change the control's background color according to the settings of its properties and its contents.

Start a new ActiveX control project and add a test project as usual. Then name the project's components as follows:

1. Select the project and change its name to ColorTextBox.
2. Select the UserControl object and change its name to CTextBox.
3. Select the test project and change its name to TestProject.
4. Select the test Form and change its name to TestForm.

Since the custom control is nothing less than a TextBox control, place an instance of the TextBox control on it. The TextBox control must cover the entire UserControl object, so you must enter the following code in the UserControl object's Resize event handler:

```
Private Sub UserControl_Resize()
    Text1.Move 0, 0, UserControl.Width, UserControl.Height
End Sub
```

The remaining custom code must use the Mandatory property, so we can't add it now. At this point, you can start the ActiveX Control Interface Wizard to generate most of the code. Our goal is to incorporate all the functionality of the TextBox control into our custom control.

In the Select Interface Members window, move the following members from the Available Names list to the Selected Names list:

Appearance	KeyDown	MouseUp	OLEGiveFeedback
BackColor	KeyPress	MultiLine	OLESetData
Click	KeyUp	OLECompleteDrag	OLEStartDrag
Change	MaxLength	OLEDrag	PasswordChar
DblClick	MouseDown	OLEDragDrop	Refresh
Enabled	MouseIcon	OLEDragMode	Text
Font	MouseMove	OLEDragOver	ToolTip
ForeColor	MousePointer	OLEDropMode	

These are the basic members of the TextBox control, with the exception of the data-binding properties (DataSource, DataMember, and so on). When you duplicate the functionality of an existing control with your custom control, you must make sure that all the members a developer expects to find in the custom control are there. I have skipped the data-binding properties because I don't plan to use this control with databases. (The topic of creating data-bound controls is fairly advanced and is not covered in this book.)

Click the Next button to display the next window, Create Custom Interface Members. Here you must add the following custom members:

EnterFocusColor The control's background color when it receives the focus.

LeaveFocusColor The control's background color when it loses the focus.

Mandatory If this property is True, the control's background is set to the MandatoryColor property's value, to indicate that the control is used for a required field.

MandatoryColor The control's background color when its Mandatory property is True.

In the next window of the Wizard, map all members except the custom member to the corresponding properties of the TextBox1 control. No members of the CTextBox control are mapped to the UserControl object, simply because the TextBox takes over the entire UserControl object.

In the next window, Set Attributes, you define the properties of the custom properties. Enter the attributes shown in Table 16.4 in the appropriate fields in the Set Attributes window of the Wizard.

TABLE 16.4: Properties of the Custom Properties

Property Name	Data Type	Default Value	Runtime	Design Time
EnterFocusColor	OLE_COLOR	&H00FFFF	Read/Write	Read/Write
LeaveFocusColor	OLE_COLOR	&HFFFFFF	Read/Write	Read/Write
Mandatory	Boolean	False	Read/Write	Read/Write
MandatoryColor	OLE_COLOR	&HFF0000	Read/Write	Read/Write

In the same window, you also enter a description for each property, which will appear in the Properties window when the corresponding property is selected. Click the Next button, and then click Finish to generate the control's code.

You can now open the UserControl object's Code window and see the code generated by the Wizard. The following variables and initial values appear at the top of the Code window:

```
'Default Property Values:
Const m_def_Mandatory = False
Const m_def_EnterFocusColor = &HFF00FF
Const m_def_MandatoryColor = &HFF
Const m_def_LeaveFocusColor = &HFFFFFF
```

```
'Property Variables:
Dim m_Mandatory As Boolean
Dim m_EnterFocusColor As Variant
Dim m_MandatoryColor As OLE_COLOR
Dim m_LeaveFocusColor As OLE_COLOR
```

These variable and constant definitions correspond to the properties we specified in the windows of the Wizard. All standard members have already been implemented for you, and you need not change them, except for the Property procedures of the Appearance property. The Wizard will implement this property as an Integer, but an Enumerated type works better for properties with a limited number of settings. So, insert the following Type declaration:

Code

```
Enum Flat3D
    Flat
    [3D]
End Enum
```

and then change the Property procedures as follows.

Code 16.17: The Revised Appearance Property Procedures

```
Public Property Get Appearance() As Flat3D
    Appearance = Text1.Appearance
End Property

Public Property Let Appearance(ByVal New_Appearance As Flat3D)
    Text1.Appearance() = New_Appearance
    PropertyChanged "Appearance"
End Property
```

You must also modify the code of the Property Let procedure for the MandatoryColor property. This property can be set only if the control's Mandatory property is True. If it's False, the user must first change it and then set the MandatoryColor property.

Code 16.18: The Revised MandatoryColor Property Procedures

Code

```
Public Property Let MandatoryColor(ByVal New_MandatoryColor As
OLE_COLOR)
    m_MandatoryColor = New_MandatoryColor
    If m_Mandatory Then Text1.BackColor = New_MandatoryColor
    PropertyChanged "MandatoryColor"
End Property
```

The Mandatory property is an integer, and you can enter any integer values in its field, in the Properties window. Define the following Enumerated type:

```
Enum ReqOpt
    [Optional]
    Required
End Enum
```

Notice that the Optional value must be enclosed in square brackets, because it's a Visual Basic keyword. Modify the definitions of the Property procedures of the Mandatory property as follows.

Code 16.19: The Modified Mandatory Property Procedures

```
Public Property Get Mandatory() As ReqOpt
    Mandatory = m_Mandatory
End Property

Public Property Let Mandatory(ByVal New_Mandatory As ReqOpt)
    m_Mandatory = New_Mandatory
    If m_Mandatory = Required Then
        Text1.BackColor = m_MandatoryColor
    Else
        Text1.BackColor = m_def_LeaveFocusColor
    End If
    PropertyChanged "Mandatory"
End Property
```

When the Mandatory property is set to True, the control automatically sets its background color property to the color value of the MandatoryValue property.

This takes care of the trivial code necessary for the control's proper operation. You're now ready to add the custom code to enhance the operation of the TextBox control. The code that enhances the operation of the TextBox control is located in the UserControl object's LostFocus event. When the user moves the focus away from a CTextBox control, the code examines the control's contents and its Mandatory property. If the control is empty and its Mandatory property is True, the TextBox control's background is set to the value of the MandatoryColor property.

Code 16.20: The UserControl's LostFocus Event Handler

```
Private Sub Text1_LostFocus()
    If Len(Trim(Text1.Text)) = 0 And m_Mandatory = Required Then
        Text1.BackColor = m_MandatoryColor
    Else
```

```
        Text1.BackColor = LeaveFocusColor
    End If
End Sub
```

NOTE Notice that the code isn't raising the LostFocus event. Although the control's behavior when it loses the focus is determined by its code, you may still want to be able to program the LostFocus event. You *can* use the LostFocus event, because this event can't be triggered by the control itself. It's triggered by the control's container (the Form), and you can't raise it from within the control's code. So, even though the statement RaiseEvent LostFocus() doesn't appear anywhere in the control's code, the Lostfocus event can still be programmed. The same is true for the GotFocus event, which is also raised by the container, not by the control.

The CTextBox control has another feature: when it's active, it changes its background color to the value of the EnterFocusColor property:

```
Private Sub Text1_GotFocus()
    Text1.BackColor = EnterFocusColor
End Sub
```

If you don't like this behavior, simply set the EnterFocusColor to the same value as the control's Background color.

To summarize, ActiveX controls combine design elements from both standard VB applications and ActiveX component design. Their properties, methods, and events are handled just as their counterparts in ActiveX components:

- Properties are private variables, which can be read or set through Property procedures.
- Methods are public subroutines.
- Events can be raised from anywhere in an ActiveX control with the RaiseEvent method.

The control's visible interface is drawn on a UserControl object, which is quite similar to a Form. It supports nearly all of a Form's properties and methods, including the Drawing methods. There are no means for loading and unloading UserControls as there are for Forms, but you can make a control visible or invisible at runtime from within your code.

The code of the control resides in key events of the control, such as the Paint and Resize events, and is no different from the code you use to develop a standalone application with similar functionality.

The integration of an ActiveX control in the development environment is the responsibility of Visual Basic. The properties you attach to the control are automatically displayed in the Properties window, and the syntax of its methods is displayed as you type code (they are incorporated into the AutoList Members feature of the Visual Basic IDE). In short, developing an ActiveX control is strikingly similar to developing a standard VB application. The result is a new animal that can live in various environments, including Web pages, as you'll see in the last part of this book.