18 Custom Windows Forms Controls

As if the controls in the Windows Forms package weren't enough, you can create your own controls. As a matter of fact, Visual Studio makes the creation of new controls a breeze. In this chapter, I'll show how you can create custom controls using one of the following approaches:

- Inheriting from an existing control
- Composing multiple controls
- Creating a control from scratch

You can deploy the control as a private assembly or install it in the GAC. The latter solution is preferable if the control must be used by many applications, but using private assemblies is OK in many cases. The control must not be installed in the GAC if it is being displayed from inside Internet Explorer. (See the section "Hosting Custom Controls in Internet Explorer" at the end of this chapter.)

Note To keep the code as concise as possible, all the code samples in this section assume that the following Imports statements are used at the file or project level:

```
Imports System.ComponentModel
Imports System.ComponentModel.Design
Imports System.Text.RegularExpressions
Imports System.Drawing.Drawing2D
Imports System.Threading
Imports System.Drawing.Design
Imports System.Windows.Forms.Design
```

Inheriting from an Existing Control

The easiest way to author a new Windows Forms control is by inheriting it from an existing control. This approach is the right one when you want to extend an existing control with new properties, methods, or events but without changing its appearance significantly. For example, you can create a ListBox that supports icons by using the owner-draw mode internally and exposing an Images collection to the outside. Here are other examples: an extended PictureBox control that supports methods for graphic effects and a TreeView-derived control that automatically displays the file and folder hierarchy for a given path.

In the example that follows, I use this technique to create an extended TextBox control named TextBoxEx, with several additional properties that perform advanced validation chores. I chose this example because it's relatively simple without being a toy control and because it's complete and useful enough to be used in a real application.

The TextBoxEx control has a property named IsRequired, which you set to True if the control must be filled before moving to another control, and the ValidateRegex property, which is a regular expression that the field's contents must match. There's also an ErrorMessage property and a Validate method, which returns True or False and can display a message box if the validation failed. The control will take advantage of the Validating event of the standard TextBox control to cancel the focus shift if the current value can't be validated.

Creating the Control Project

Create a new Windows Control Library project named CustomControlDemo. A project of this type is actually a Class Library project (that is, it generates a DLL) and contains a file named UserControl1.vb. Rename the file TextBoxEx.vb, then switch to its code portion, and replace the template created by Visual Studio with what follows:

```
Public Class TextBoxEx
    Inherits System.Windows.Forms.TextBox

End Class
```

This is just the skeleton of our new control, but it already has everything it needs to behave like a TextBox control. Because you want to expand on this control, let's begin by adding the new IsRequired property:

```
Public Class TextBoxEx
    Inherits System.Windows.Forms.TextBox

    Sub New()
        MyBase.New()
    End Sub

    ' The IsRequired property
    Dim m_IsRequired As Boolean

    Property IsRequired() As Boolean
        Get
            Return m_IsRequired
        End Get
        Set(ByVal Value As Boolean)
            m_IsRequired = Value
        End Set
    End Property
End Class
```

Note that you can't use a public field to expose a property in the Properties window, so you must use a Property procedure even if you don't plan to validate the value entered by the user, as in the preceding case.

The Sub New constructor isn't strictly required in this demo, but it doesn't hurt either. In a more complex custom control, you can use this event to initialize a property to a different value—for example, you might set the Text property to a null string so that the developer who uses this control doesn't have to do it manually.

The control doesn't do anything useful yet, but you can compile it by selecting Build on the Build menu. This action produces a DLL executable file. Take note of the path of this file because you'll need it very soon.

Creating the Client Application

On the File menu, point to Add Project and then click New Project to add a Windows Application project named CustomControlsTest to the same solution as the existing control. You can also create this test application using another instance of Visual Studio, but having both projects in the same solution has an important advantage, as I'll explain shortly.

Next make CustomControlsTest the start-up project so that it will start when you press F5. You can do this by right-clicking the project in the Solution Explorer window and clicking Set As Startup Project on the shortcut menu. Visual Studio confirms the new setting by displaying the new start-up project in boldface.

Right-click the Toolbox and click Add Tab on the shortcut menu to create a new tab named My Custom Controls, on which you'll place your custom controls. This step isn't required, but it helps you keep things well ordered.

Right-click the new Toolbox tab (which is empty), and click Add/Remove Items on the shortcut menu. Switch to the .NET Framework Components page in the Customize Toolbox dialog box that appears, click Browse, and select the CustomControlDemo.dll file that you compiled previously. The TextBoxEx control appears now in the Customize Toolbox dialog box, so you can ensure that its check box is selected and that it is ready to be added to the custom tab you've created in the Toolbox. (See Figure 18-1.)

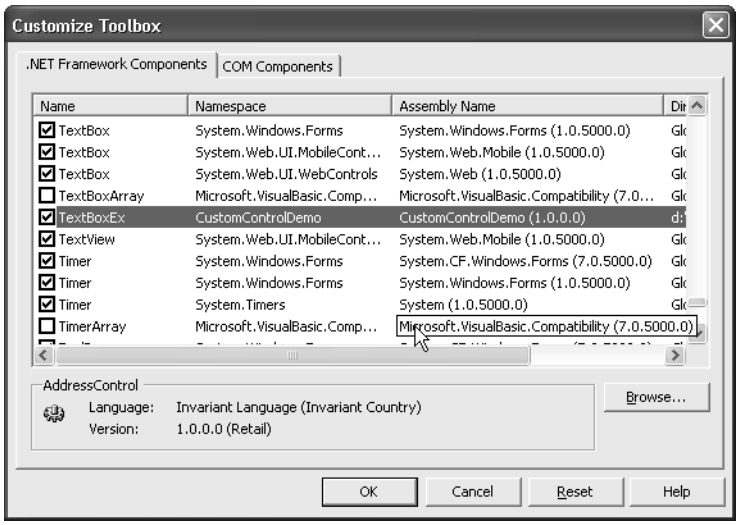

Figure 18-1 Adding a new control to the Toolbox

Drop an instance of the control on the form, and switch to the Properties window. You'll see all the properties of the TextBox control, which should be no surprise because the TextBoxEx control inherits them from its base class. Scroll the Properties window to ensure that the new IsRequired property is also there; the designer has detected that it's a Boolean property, so the designer knows that the property can be set to True or False.

Adding the Validation Logic

Now that you know your control is going to work correctly on a form's surface, you can go back to the TextBoxEx code module and add the remaining two properties. Notice that the code checks that the regular expression assigned to the ValidateRegex property is correct by attempting a search on a dummy string:

```
' The ErrorMessage property
Dim m_ErrorMessage As String

Property ErrorMessage() As String
    Get
        Return m_ErrorMessage
    End Get
    Set(ByVal Value As String)
        m_ErrorMessage = Value
    End Set
End Property

' The ValidateRegex property
Dim m_ValidateRegex As String
```

```
    Property ValidateRegex() As String
        Get
            Return m_ValidateRegex
        End Get
        Set(ByVal Value As String)
            ' Check that this is a valid regular expression.
            Try
                If Value <> "" Then
                    Dim dummy As Boolean = Regex.IsMatch("abcde", Value)
                End If
                ' If no error, value is OK.
                m_ValidateRegex = Value
            Catch ex As Exception
                MessageBox.Show(ex.Message, "Invalid Property", _
                    MessageBoxButtons.OK, MessageBoxIcon.Error)
            End Try
        End Set
    End Property
```

You now have all you need to implement the Validate method, which returns False if the current value doesn't pass the validation test. Notice that the code uses the MyBase.Text property to access the control's contents:

```
    Function Validate() As Boolean
        ' Assume control passed the validation.
        Validate = True
        ' Apply the IsRequired property.
        If Me.IsRequired And Me.Text = "" Then
            Validate = False
        End If
        ' Apply the ValidateRegex property if specified.
        If Validate = True And Me.ValidateRegex <> "" Then
            Validate = Regex.IsMatch(Me.Text, Me.ValidateRegex)
        End If
    End Function
```

The structure of the Validate method permits you to add other validation tests just before the End Function statement. Having a single place in which the validation occurs means that this is the only point to modify when you want to extend the control with new properties.

You still need to write the code that actually validates the current value when the user moves the input focus to a control that has CausesValidation set to True. The inner TextBox control fires a Validating event when this happens, so you can implement the validation by using the following naive approach:

```
Private Sub TextBoxEx_Validating(ByVal sender As Object, _
    ByVal e As CancelEventArgs) Handles MyBase.Validating
    ' If the validation fails, cancel the focus shift.
    If Me.Validate() = False Then e.Cancel = True
End Sub
```

This technique works in the sense that it does prevent the focus from leaving the TextBoxEx control. The problem, however, is that you can't prevent the Validating event from propagating to your client form. In other words, the developer who uses this control will see a Validating event even if the focus shift is going to be canceled anyway.

A far better technique consists of intercepting the validation action *before* the inner TextBox object fires the event. You can do this by overriding the OnValidating protected method in the base class. You can create the template for the overridden method by clicking Overrides in the Class Name combo box in the code editor and then clicking the method in question in the Method Name combo box. (See Figure 18-2.) The correct code for performing the internal validation follows; it correctly raises the Validating event only if the validation passed:

```
Protected Overrides Sub OnValidating(ByVal e As CancelEventArgs)
    If Me.Validate() Then
        ' If validation is OK, let the base class fire the Validating event.
        MyBase.OnValidating(e)
    Else
        ' Else, cancel the focus shift.
        e.Cancel = True
    End If
End Sub
```

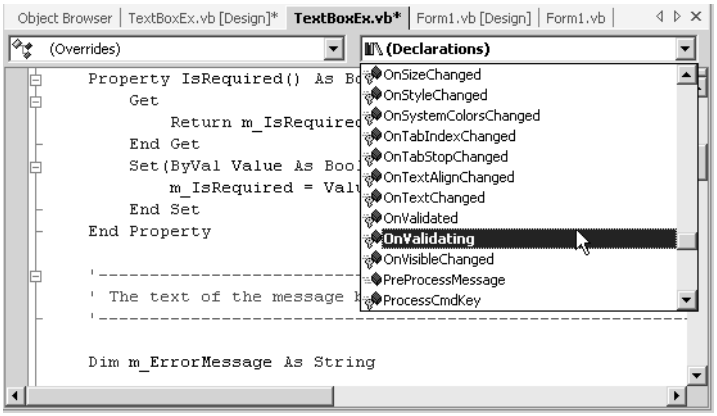

Figure 18-2 Using the code editor and the Class Name and Method Name combo boxes to create an overridden method

The focal point here is the call to MyBase.OnValidating, where the base class fires the Validating event and possibly performs additional actions.

Testing the Control

You can test your control in the client form at last. Create a form in the client application, and drop the following controls on it: a TextBoxEx control, a regular TextBox

control, and a Label control that you'll use for displaying messages. Next set the TextBoxEx properties as follows:

```
' A required field that can contain only digits
TextBoxEx1.IsRequired = True
TextBoxEx1.ValidateRegex = "\d+"
```

Run the program; you'll see that you can't move the focus from the TextBoxEx control to the TextBox control unless you enter one or more digits (or you set the TextBox control's CausesValidation property to False). You can also add a Validating event procedure that proves that the form receives this event only if the internal validation passed:

```
Private Sub TextBoxEx1_Validating(ByVal sender As Object, _
    ByVal e As CancelEventArgs) Handles TextBoxEx1.Validating
    MsgBox("Validating event in form")
End Sub
```

Design-Time Debugging

Whenever you set a control's property from inside the Properties window, Visual Studio .NET invokes the Property Set procedure for that property, so in general design-time property assignments work in much the same way as they work at run time. Sometimes, however, you might see slightly different behaviors at design time and run time and might like to run the control under a debugger to see what's going wrong. Alas, you can't debug a control when Visual Studio .NET is in design time because the debugger can't work on an instance of the DLL that isn't running yet. In this section, I'll show you a simple technique that lets you run the control under the debugger at run time *and* design time.

The trick is quite simple, once you know it. Create a new solution with the control library as its only project, open the Debugging page of the Project Properties dialog, set the Start Action to Start external program, and type the complete path to Visual Studio .NET's executable (C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE\devenv.exe for a standard installation). In the Command line arguments field, type the complete path to the demo project or solution that uses the custom control, enclosing it between double quotation marks if it contains spaces. (See Figure 18-3.) That's all you need!

Set all the breakpoints you want in the custom control's source code, and run the project. Visual Studio .NET will compile the control and run another instance of Visual Studio itself, which in turn will load the sample project that uses the custom control. You can now set properties in the Properties window and execution will stop at your breakpoints. (Of course, these breakpoints will be active also when the control is in runtime mode.)

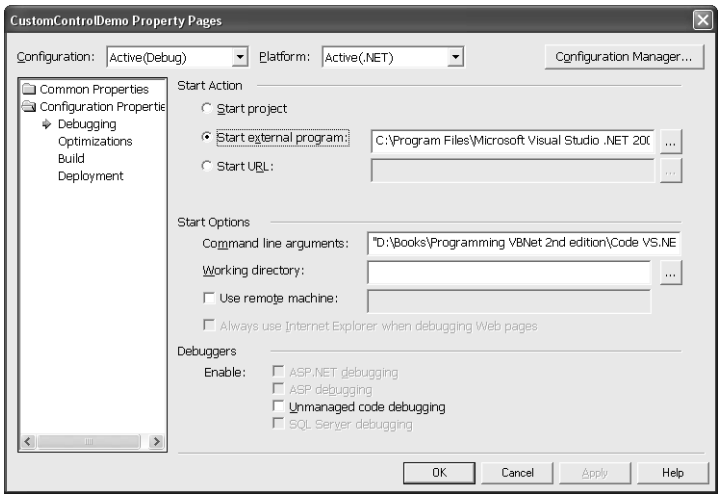

Figure 18-3 Setting project properties for testing a custom control at design time with a different instance of Visual Studio .NET

Improving the Custom Control

You can add many other properties to the TextBoxEx control to increase its usefulness and flexibility. Each property is an occasion to discuss additional advanced capabilities of custom control creation.

Working with Other Types

It is interesting to see how the Properties browser works with properties that aren't plain numbers or strings. For example, you can add a DisplayControl and an ErrorForeColor property. The former takes a reference to a control that will be used to display the error message, and the latter is the color used for the error message itself. Implementing these properties is straightforward:

```
Dim m_ErrorForeColor As Color = SystemColors.ControlText

Property ErrorForeColor() As Color
    Get
        Return m_ErrorForeColor
    End Get
    Set(ByVal Value As Color)
        m_ErrorForeColor = Value
    End Set
End Property

Dim m_DisplayControl As Control

Property DisplayControl() As Control
    Get
        Return m_DisplayControl
    End Get
```

```
    Set(ByVal Value As Control)
        m_DisplayControl = Value
    End Set
End Property
```

You reference these new properties at the end of the Validate method, after the validation code you wrote previously:

```
    ⋮
    ' If the validation failed but the client defined a display control
    ' and an error message, show the message in the control.
    If Not (DisplayControl Is Nothing) And Me.ErrorMessage <> "" Then
        If Validate() Then
            ' Delete any previous error message.
            DisplayControl.Text = ""
        Else
            ' Display error message, and enforce color.
            DisplayControl.Text = Me.ErrorMessage
            DisplayControl.ForeColor = m_ErrorForeColor
        End If
    End If
End Function
```

Rebuild the solution, select the TextBoxEx control, and switch to the Properties window, where you'll see a couple of interesting things. First, the ErrorForeColor property displays the same color palette that all other color properties expose. Second, the Properties window recognizes the nature of the new DisplayControl property and displays a drop-down list that lets you select one of the controls on the current form. In this particular example, you can set this property to the Label1 control so that all error messages appear there. Also, remember to store a suitable message in the ErrorMessage property. Of course, you can do these operations from code as well, which is necessary if the target control is on another form:

```
TextBoxEx1.ErrorMessage = "This field must contain a positive number"
TextBoxEx1.ErrorForeColor = Color.Red
TextBoxEx1.DisplayControl = Label1
```

You can see another smart behavior of the Properties window if you have an enumerated property. For example, you can create a ValueType property that states the type of variable to which the contents of the field will be assigned:

```
Enum ValidTypes
    AnyType = 0
    ByteType
    ShortType
    IntegerType
    LongType
    TypeSingleType
    DoubleType
    DecimalType
    DateTimeType
End Enum
```

```
Dim m_ValidType As ValidTypes = ValidTypes.AnyType

Property ValidType() As ValidTypes
    Get
        Return m_ValidType
    End Get
    Set(ByVal Value As ValidTypes)
        m_ValidType = Value
    End Set
End Property
```

You can browse the demo application to see how this feature is implemented, but for now just rebuild the solution and go to the Properties window again to check that the ValueType property corresponds to a combo box that lets the user select the valid type among those available.

Adding Attributes

You can further affect how your custom control uses and exposes the new properties by means of attributes. For example, all public properties are displayed in the Properties window by default. This isn't desirable for read-only properties (which are visible but unavailable by default) or for properties that should be assigned only at run time via code. You can control the visibility of elements in the Properties window by using the Browsable attribute. The default value for this attribute is True, so you must set it to False to hide the element. For example, let's say that you want to hide a read-only property that returns True if the control's current value isn't valid:

```
<Browsable(False)> _
ReadOnly Property IsInvalid() As Boolean
    Get
        Return Not Validate()
    End Get
End Property
```

The EditorBrowsable attribute is similar to Browsable, but it affects the visibility of a property, method, or event from inside the code editor, and in practice determines whether you see a given member in the little window that IntelliSense displays. It takes an EditorBrowsableState enumerated value, which can be Never, Always, or Advanced. The Advanced setting makes a member visible only if you clear the Hide advanced members option in the General page under the All languages folder (or the folder named after the language you're using), which in turn is in the Text Editor folder in the Options dialog box that you reach from the Tools menu:

```
<EditorBrowsable(EditorBrowsableState.Advanced)> _
Sub EnterExpertMode()
    ⋮
End Sub
```

Another frequently used attribute is Description, which defines the string displayed near the bottom edge of the Properties window:

```
<Description("The control that will display the error message")> _
Property DisplayControl() As Control
    ⋮
End Property
```

You put a property into a category in the Properties window by using the Category attribute. You can specify one of the existing categories—Layout, Behavior, Appearance—or define a new one. If a property doesn't belong to a specific category, it appears in the Misc category:

```
<Description("The control that will display the error message"), _
    Category("Validation")> _
Property DisplayControl() As Control
    ⋮
End Property
```

By default, properties aren't localizable and the form designer doesn't save their values in a separate source file when the user selects a language other than the default one. You can change this default behavior by using the Localizable attribute:

```
<Localizable(True) > _
Property HeaderCaption() As String
    ⋮
End Property
```

The DefaultProperty attribute tells the environment which property should be selected when the user creates a new instance of the control and then activates the Properties window. Similarly, the DefaultEvent attribute specifies the event handler that's automatically created when you double-click on a control in the designer. (For example, TextChanged is the default event for the TextBox control.) You apply these attributes to the class and pass them the name of the property or the event:

```
<DefaultProperty("IsRequired"), DefaultEvent("InvalidKey")> _
Public Class TextBoxEx
    Inherits System.Windows.Forms.TextBox

    Event InvalidKey(ByVal sender As Object, ByVal e As EventArgs)
    ⋮
End Class
```

The MergableProperty attribute tells whether a property is visible in the Properties window when multiple controls are selected. The default value of this attribute is True, so you must include it explicitly only if you don't want to allow the user to modify this property for all the selected controls. In practice, you use this attribute when a property can't have the same value for multiple controls on a form (as in the case of the TabIndex or Name property):

```
<MergableProperty(False)> _
Property ProcessOrder() As Integer
    ⋮
End Class
```

The RefreshProperties attribute is useful if a new value assigned to a property can affect other properties in the Properties window. The default behavior of the Properties window is that only the value of the property being edited is updated, but you can specify that all properties should be requeried and refreshed by using this attribute:

```
Dim m_MaxValue As Long = Long.MaxValue

<RefreshProperties(RefreshProperties.All)> _
Property MaxValue() As Long
    Get
        Return m_MaxValue
    End Get
    Set(ByVal Value As Long)
        m_MaxValue = Value
        ' This property can affect the Value property.
        If Value > m_MaxValue Then Value = m_MaxValue
    End Set
End Property
```

Working with Icons

Each of your carefully built custom controls should have a brand-new icon assigned to them, to replace the default icon that appears in Visual Studio .NET's control Toolbox. Control icons can be either stand-alone .bmp or .ico files, or bitmaps embedded in a DLL (typically that is the same DLL that contains the custom control, but it can be a different DLL as well). The lower left pixel in the bitmap determines the transparent color for the icon—for example, use a yellow color for this pixel to make all other yellow pixels in the icon transparent (so they will be shown as unavailable if your current color setting uses gray for the control Toolbox).

The procedure to embed the bitmap in the DLL that contains the control isn't exactly intuitive. First, use the Add New Item command from the Project menu to add a bitmap file to your project and name this bitmap after your control (TextBoxEx.bmp in this example). Next, ensure that the bitmap is 16-by-16 pixels by setting its Width and Height properties in the Properties window and draw the bitmap using your artistic abilities and the tools that Visual Studio .NET gives you. (See Figure 18-4.) Then go to the Solution Explorer, select the bitmap file, press F4 to display the Properties window for the file (as opposed to the Properties window for the bitmap that you used to set the image's size), and change the Build Action property to Embedded Resource. If you now recompile the CustomControlDemo.dll assembly, you'll see that the TextBoxEx control uses the new icon you defined. (You must use the Add/Remove Items command twice to see the new icon—once to remove the previous version and once to add the new one.)

Notice that you haven't used any attribute to assign the icon to the TextBoxEx because the association is implicitly made when you assign the bitmap the same name as the control it's related to. This technique works only if the custom control is in the project's default namespace.

You can make the association explicit by using a ToolboxBitmap attribute. For example, you need this attribute when the bitmap or icon file isn't embedded in the DLL:

```
<ToolboxBitmap("C:\CustomControlDemo\TextBoxEx.ico")> _
Public Class TextBoxEx
    ⋮
End Class
```

This attribute is also useful when the bitmap is embedded in a DLL other than the one that contains the custom control. In this case, you pass this attribute the System.Type object that corresponds to any class defined in the DLL containing the resource (Visual Studio .NET uses this Type object only to locate the assembly), and you can pass a second optional argument equal to the name of the bitmap (this is necessary if the bitmap isn't named after the custom control):

```
<ToolboxBitmap(GetType(TextBoxEx), "TextBoxIcon.bmp")> _
Public Class TextBoxEx
    ⋮
End Class
```

Another case for which you need the ToolboxBitmap attribute is when your control belongs to a nested namespace below the project's root namespace. For example, if the TextBoxEx control lives inside the CustomControlDemo.Controls namespace, you must rename the bitmap file as Controls.TextBoxIcon.bmp (even though you must continue to use TextBoxIcon.bmp in the attribute's constructor).

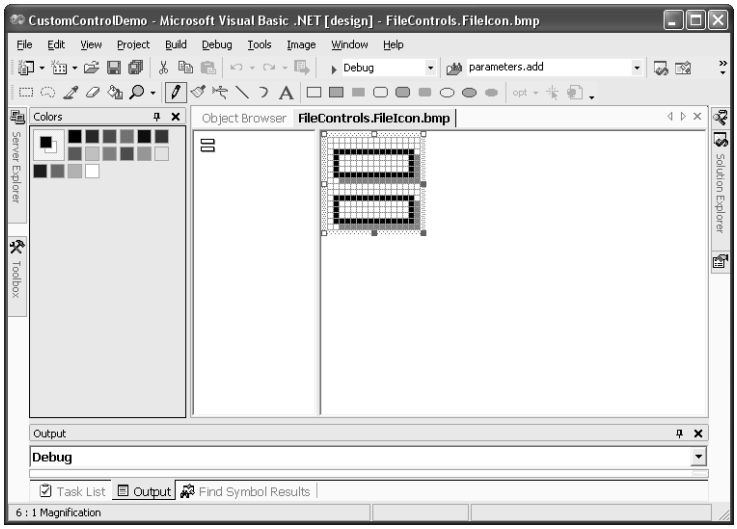

Figure 18-4 The Visual Studio .NET bitmap editor

Working with Default Values

You have surely noticed that the Properties window uses boldface to display values different from the property's default value, so you might have wondered where the default value is defined. As you might guess, you define the default value with yet another attribute, appropriately named DefaultValue. This example is taken from the demo application:

```
Dim m_BeepOnError As Boolean = True

<Description("If True, a beep will be emitted if validation fails"), _
    DefaultValue(True)> _
Property BeepOnError() As Boolean
    Get
        Return m_BeepOnError
    End Get
    Set(ByVal Value As Boolean)
        m_BeepOnError = Value
    End Set
End Property
```

Note that the DefaultValue attribute is just a directive to the Properties window: it doesn't actually initialize the property itself. For that, you must use an initializer or use the necessary code in the Sub New procedure.

Unfortunately, the DefaultValue attribute has a problem: it can take only constant values, and in several cases the default value isn't a constant. For example, the value SystemColors.ControlText, which is the initial value for the ErrorForeColor property, isn't a constant, so you can't pass it to the DefaultValue attribute's constructor. To resolve this difficulty, the author of the custom control can create a special Reset*xxxx* procedure, where *xxxx* is the property name. Here is the code in the TextBoxEx class that the form designer implicitly calls to initialize the two color properties:

```
Sub ResetErrorForeColor()
    ErrorForeColor = SystemColors.ControlText
End Sub
```

If a property is associated with a Reset*xxxx* method, you can reset the property by clicking Reset on the context menu that you bring up by clicking on the property name in the Properties window.

The DefaultValue attribute has another, less obvious, use: if a property is set to its default value—as specified by this attribute—this property isn't persisted, and the designer generates no code for it. This behavior is highly desirable because it avoids the generation of a lot of useless code that would slow down the rendering of the parent form. Alas, you can't specify a nonconstant value, a color, a font, or another complex object in the DefaultValue attribute. In this case, you must implement a method named

ShouldSerialize*xxxxx* (where *xxxx* is the name of the property) that returns True if the property must be serialized and False if its current value is equal to its default value:

```
Function ShouldSerializeErrorForeColor() As Boolean
    ' We can't use the = operators on objects, so we use
    ' the Equals method.
    Return Not Me.ErrorForeColor.Equals(SystemColors.ControlText)
End Function
```

Creating Composed Multiple Controls

More complex custom controls require that you combine multiple controls. You can think of several custom controls of this type, such as a control that implements Windows Explorer–like functionality by grouping together a TreeView, a ListView, and a Splitter control; or a control made up of two ListBox controls that lets you move items from one ListBox to the other, using auxiliary buttons or drag-and-drop.

In this section, I build a composite custom control named FileTextBox, which lets the user enter a filename by typing its name in a field or by selecting it in an OpenFile common dialog box.

Creating the UserControl Component

Add a new UserControl module named FileTextBox to the CustomControlDemo project by selecting the project and then selecting Add UserControl on the Project menu. Ensure that you've selected the CustomControlDemo project before you perform this action—otherwise, the custom control will be added to the test application instead.

The UserControl class derives from the ContainerControl class, so it can work as a container for other controls. In this respect, the UserControl object behaves very much like the Form object, and in fact, the programming interface of these classes is very similar, with properties such as Font, ForeColor, AutoScroll, and so on. A few typical form properties are missing because they don't make sense in a control—for example, MainMenu and TopMost—but by and large, you code against a UserControl as if you were working with a regular form.

You can therefore drop on the UserControl's surface the three child controls you need for the FileTextBox control. These are a TextBox for the filename (named txtFilename, with a blank Text property); an OpenFileDialog control to display the dialog box (named OpenFileDialog1); and a Button control to let the user bring up the common dialog (named btnBrowse, with the Text property set to three dots). You can arrange these controls in the manner of Figure 18-5. Don't pay too much attention to their alignment, however, because you're going to move them on the UserControl's surface by means of code.

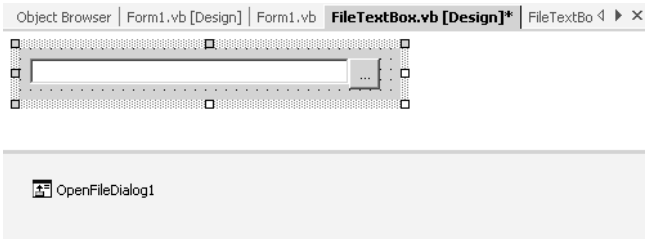

Figure 18-5 The FileTextBox custom control at design time

Before you start adding code, you should compile the CustomControlDemo project to rebuild the DLL and then switch to the client project to invoke the Customize ToolBox command. You'll see that the FileTextBox control doesn't appear yet in the list of available .NET controls in the Toolbox, so you have to click on the Browse button and select CustomControlDemo.dll once again.

If all worked well, the new FileTextBox is in the Toolbox and you can drop it on the test form. You can resize it as usual, but its constituent controls don't resize correctly because you haven't written any code that handles resizing.

Adding Properties, Methods, and Events

The FileTextBox control doesn't expose any useful properties yet, other than those provided by the UserControl class. The three child controls you placed on the UserControl's surface can't be reached at all because by default they have a Friend scope and can't be seen from code in the client project. Your next step is to provide programmatic access to the values in these controls. In most cases, all you need to do is create a property procedure that wraps directly around a child control property—for example, you can implement the Filename and Filter properties, as follows:

```
<Description("The filename as it appears in the Textbox control")> _
Property Filename() As String
    Get
        Return Me.txtFilename.Text
    End Get
    Set(ByVal Value As String)
        Me.txtFilename.Text = Value
    End Set
End Property

<Description("The list of file filters"), _
    DefaultValue("All files (*.*)|*.*")> _
Property Filter() As String
    Get
        Return OpenFileDialog1.Filter
    End Get
    Set(ByVal Value As String)
        OpenFileDialog1.Filter = Value
    End Set
End Property
```

If you wrap your property procedures around child control properties, you must be certain that you properly initialize the child controls so that the initial values of their properties match the values passed to the DefaultValue attribute. In this case, you must ensure that you set the OpenFileDialog1.Filter property correctly.

You don't create wrappers around child control properties only. For example, you can implement a ShowDialog method that wraps around the OpenFileDialog1 control method of the same name:

```
Function ShowDialog() As DialogResult
    ' Show the OpenFile dialog, and return the result.
    ShowDialog = OpenFileDialog1.ShowDialog
    ' If the result is OK, assign the filename to the TextBox.
    If ShowDialog = DialogResult.OK Then
        txtFilename.Text = OpenFileDialog1.FileName
    End If
End Function
```

You can also provide wrappers for events. For example, exposing the FileOk event gives the client code the ability to reject invalid filenames:

```
Event FileOk(ByVal sender As Object, ByVal e As CancelEventArgs)

Private Sub OpenFileDialog1_FileOk(ByVal sender As Object, _
    ByVal e As CancelEventArgs) Handles OpenFileDialog1.FileOk
    RaiseEvent FileOk(Me, e)
End Sub
```

Bear in mind that the client will receive the event from the FileTextBox control, not from the inner OpenFileDialog control, so you must pass Me as the first argument of the RaiseEvent method, as in the preceding code snippet. In some cases, you also need to change the name of the event to meet the client code's expectations—for example, the TextChanged event from the inner txtFilename should be exposed to the outside as FilenameChanged because Filename is the actual property that will appear as modified to clients. In other cases you might need to adjust the properties of the EventArgs-derived object—for example, to convert mouse coordinates so that they refer to the UserControl rather than the constituent control that fired the event.

Not all events should be exposed to the outside, however. The button's Click event, for example, is handled internally to automatically fill the txtFilename field when the user selects a file from the common dialog box:

```
Private Sub btnBrowse_Click(ByVal sender As Object, _
    ByVal e As EventArgs) Handles btnBrowse.Click
    ShowDialog()
End Sub
```

Shadowing and Overriding UserControl Properties

If you modify the Font property of the FileTextBox control, you'll notice that the new settings are immediately applied to the inner txtFilename control, so you don't have to manually implement the Font property. The custom control behaves this way because you never assign a specific value to txtFilename.Font, so it automatically inherits the parent UserControl's settings. This feature can save you a lot of code and time. The group of properties that you shouldn't implement explicitly includes Enabled and TabStop because all the constituent controls inherit these properties from their UserControl container.

You're not always that lucky. For example, TextBox controls don't automatically inherit the ForeColor of their container, and you have to implement this property manually. A minor annoyance is that the UserControl already exposes a property of this name, so you receive a compilation warning. You can get rid of this warning by using the Shadows keyword. Note that you also must use explicit shadowing with the Reset*xxxx* procedure:

```
Shadows Property ForeColor() As Color
    Get
        Return txtFilename.ForeColor
    End Get
    Set(ByVal Value As Color)
        txtFilename.ForeColor = Value
    End Set
End Property

Shadows Sub ResetForeColor()
    Me.ForeColor = SystemColors.ControlText
End Sub

Function ShouldSerializeForeColor() As Boolean
    Return Not Me.ForeColor.Equals(SystemColors.ControlText)
End Function
```

If your custom control exposes a property with the same name, return type, and default value as a property in the base UserControl, you can override it instead of shadowing it—for example, the FileTextBox control overrides the ContextMenu property so that the pop-up menu also appears when the end user right-clicks on constituent controls:

```
Overrides Property ContextMenu() As ContextMenu
    Get
        Return MyBase.ContextMenu
    End Get
    Set(ByVal Value As ContextMenu)
        MyBase.ContextMenu = Value
        ' Propagate the new value to constituent controls.
        ' (This generic code works with any UserControl.)
        For Each ctrl As Control In Me.Controls
            ctrl.ContextMenu = Me.ContextMenu
        Next
    End Set
End Property
```

In many cases, however, you really have to go as far as overriding a property in the base UserControl only if you need to cancel its default behavior. If you just want a notification that a property has changed, you can be satisfied by simply trapping the *xxxx*-Changed event—for example, the following code displays the btnBrowse button control with a flat appearance when the FileTextBox control is disabled:

```
Private Sub FileTextBox_EnabledChanged(ByVal sender As Object, _
    ByVal e As EventArgs) Handles MyBase.EnabledChanged
    If Me.Enabled Then
        btnBrowse.FlatStyle = FlatStyle.Standard
    Else
        btnBrowse.FlatStyle = FlatStyle.Flat
    End If
End Sub
```

Notice that *xxxx*Changed events don't fire at design time, so you can't use this method to react to setting changes in the Properties window.

You often need to shadow properties in the base class for the sole purpose of hiding them in the Properties window. For example, the demo FileTextBox control can't work as a scrollable container, so it shouldn't display the AutoScroll, AutoScrollMargins, AutoScrollPosition, and DockPadding items in the Properties window. You can achieve this by shadowing the property and adding a Browsable(False) attribute:

```
<Browsable(False)> _
Shadows Property AutoScroll() As Boolean
    Get
        ' Don't really need to delegate to inner UserControl.
    End Get
    Set(ByVal Value As Boolean)
        ' Don't really need to delegate to inner UserControl.
    End Set
End Property
```

Although you can easily hide a property in the Properties window, inheritance rules prevent you from completely wiping out a UserControl property from the custom control's programming interface. However, you can throw an exception when this property is accessed at run time programmatically so that the developer using this custom control learns the lesson more quickly. You can discern whether you're in design-time or run-time mode with the DesignMode property, which the UserControl inherits from the System.ComponentModel.Component object:

```
<Browsable(False)> _
Shadows Property AutoScroll() As Boolean
    Get
        If Not Me.DesignMode Then Throw New NotImplementedException()
    End Get
    Set(ByVal Value As Boolean)
        If Not Me.DesignMode Then Throw New NotImplementedException()
    End Set
End Property
```

You must check the DesignMode property before throwing the exception—otherwise, the control doesn't work correctly in design-time mode. You can use this property for many other purposes, such as displaying a slightly different user interface at design time or run time.

Adding Resize Logic

The code inside your UserControl determines how its constituent controls are arranged when the control is resized. In the simplest cases, you don't have to write code to achieve the desired effect because you can simply rely on the Anchor and Dock properties of constituent controls. However, this approach is rarely feasible with more complex custom controls. For example, the btnBrowse button in FileTextBox always should be square, and so its height and width depend on the txtFilename control's height, which in turn depends on the current font. Besides, the height of the FileTextBox control always should be equal to the height of its inner fields. All these constraints require that you write custom resize logic in a private RedrawControls procedure and call this procedure from the UserControl's Resize event:

```
Private Sub FileTextBox_Resize(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles MyBase.Resize
    RedrawControls()
End Sub

Private Sub RedrawControls()
    ' This is the inner width of the control.
    Dim width As Integer = Me.ClientRectangle.Width
    ' This is the (desired) height of the control.
    Dim btnSide As Integer = txtFilename.Height

    ' Adjust the height of the UserControl if necessary.
    If Me.ClientRectangle.Height <> btnSide Then
        ' Resize the UserControl.
        Me.SetClientSizeCore(Me.ClientRectangle.Width, btnSide)
        ' The above statement fires a nested Resize event, so exit right now.
        Exit Sub
    End If

    ' Resize the constituent controls.
    txtFilename.SetBounds(0, 0, width - btnSide, btnSide)
    btnBrowse.SetBounds(width - btnSide, 0, btnSide, btnSide)
End Sub
```

Don't forget that the custom control's height also should change when its Font property changes, so you must override the OnFontChanged method as well. (You can't simply trap the FontChanged event because it doesn't fire at design time.)

```
Protected Overrides Sub OnFontChanged(ByVal e As EventArgs)
    ' Let the base control update the TextBox control.
    MyBase.OnFontChanged(e)
    ' Now we can redraw controls if necessary.
    RedrawControls()
End Sub
```

Creating a Control from Scratch

The third technique for creating a custom control is building it from scratch by inheriting from the Control class and painting directly on its surface using graphic methods in the GDI+ package. In general, it's a more complex approach than the other two techniques shown so far.

In this section, I'll illustrate a relatively simple example: a custom control named GradientControl that can be used to provide a gradient background for other controls. Figure 18-6 shows this control at design time, but most of the time you'll set its Docked property to Fill so that it spreads over the entire form. This control has only three properties: StartColor, EndColor, and GradientMode. This is the complete source code for this control:

```
Public Class GradientBackground
    Inherits System.Windows.Forms.Control

    ' The StartColor property
    Dim m_StartColor As Color = Color.Blue

    <Description("The start color for the gradient")> _
    Property StartColor() As Color
        Get
            Return m_StartColor
        End Get
        Set(ByVal Value As Color)
            m_StartColor = Value
            ' Redraw the control when this property changes.
            Me.Invalidate()
        End Set
    End Property

    Sub ResetStartColor()
        m_StartColor = Color.Blue
    End Sub

    Function ShouldSerializeStartColor() As Boolean
        Return Not m_StartColor.Equals(Color.Blue)
    End Function

    ' The EndColor property
    Dim m_EndColor As Color = Color.Black

    <Description("The end color for the gradient")> _
    Property EndColor() As Color
        Get
            Return m_EndColor
        End Get
        Set(ByVal Value As Color)
            m_EndColor = Value
            ' Redraw the control when this property changes.
            Me.Invalidate()
        End Set
    End Property
```

```
    Sub ResetEndColor()
        m_EndColor = Color.Black
    End Sub

    Function ShouldSerializeEndColor() As Boolean
        Return Not m_EndColor.Equals(Color.Black)
    End Function

    ' The GradientMode property
    Dim m_GradientMode As LinearGradientMode = _
        LinearGradientMode.ForwardDiagonal

    <Description("The gradient mode"), _
        DefaultValue(LinearGradientMode.ForwardDiagonal)> _
    Property GradientMode() As LinearGradientMode
        Get
            Return m_GradientMode
        End Get
        Set(ByVal Value As LinearGradientMode)
            m_GradientMode = Value
            ' Redraw the control when this property changes.
            Me.Invalidate()
        End Set
    End Property

    ' Render the control background.

    Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)
        ' Create a gradient brush as large as the client area, with specified
        ' start/end color and gradient mode.
        Dim br As New LinearGradientBrush(Me.ClientRectangle, _
            m_StartColor, m_EndColor, m_GradientMode)
        ' Paint the background and destroy the brush.
        e.Graphics.FillRectangle(br, Me.ClientRectangle)
        br.Dispose()
        ' Let the base control do its chores (e.g., raising the Paint event).
        MyBase.OnPaint(e)
    End Sub

    Private Sub GradientBackground_Resize(ByVal sender As Object, _
        ByVal e As EventArgs) Handles MyBase.Resize
        Me.Invalidate()
    End Sub
End Class
```

A custom control implemented by inheriting from the Control class must render itself in the overridden OnPaint method. In this particular case, redrawing the control is trivial because the System.Drawing.Drawing2d namespace exposes a LinearGradientBrush object that does all the work for you. In practice, for this particular control you only have to create a gradient brush as large as the control itself and then use this brush to paint the control's client rectangle. The last argument you pass to the brush's constructor is the gradient mode, an enumerated value that lets you create horizontal, vertical, forward diagonal (default), and backward diagonal gradients. The GradientMode prop-

erty is opportunely defined as type LinearGradientMode so that these four gradients appear in a drop-down list box in the Properties window.

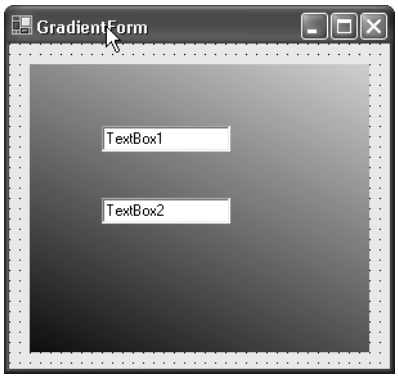

Figure 18-6 You can use the GradientControl to create eye-catching backgrounds by setting just three properties.

The only other detail to take care of is refreshing the control whenever a property changes. The best way to do so is by invalidating the control appearance with its Invalidate method so that the form engine can refresh the control at the first occurrence. This is considered a better practice than invoking the Refresh method directly because the form engine can delay all repaint operations until it's appropriate to perform them.

The ControlPaint Class

You might find the ControlPaint class especially useful when creating the user interface of your custom control. This class, in the System.Windows.Forms namespace, exposes shared methods for performing common graphic chores—for example, you can use the DrawFocusRectangle method for drawing the dotted rectangle around your control when it gets the focus:

```
Private Sub GradientBackground_GotFocus(ByVal sender As Object, _
   ByVal e As EventArgs) _
    Handles MyBase.GotFocus
    Dim gr As Graphics = Me.CreateGraphics
    ControlPaint.DrawFocusRectangle(gr, Me.Bounds)
    gr.Dispose()
End Sub
```

Other methods in the ControlPaint class can draw a border (DrawBorder), a three-dimensional border (DrawBorder3D), a button (DrawButton), a standard check box (DrawCheckBox), a three-state check box (DrawMixedCheckBox), a radio button (DrawRadioButton), a scroll bar button (DrawScrollButton), a combo box button (DrawComboButton), a disabled (grayed) string (DrawStringDisabled), and an image

in disabled state (DrawImageDisabled). All these borders and buttons can be rendered in normal, checked, flat, pushed, and inactive states. (See Figure 18-7.)

The DrawReversibleLine method draws a line in such a way that you can make the line disappear if you invoke the method again. The DrawReversibleFrame and FillReversibleRectangle methods do the same, but draw an empty and a filled rectangle, respectively. You can use these methods to implement "rubber band" techniques—for example, to let users draw lines and rectangles with the mouse. (Notice that you can't implement rubber banding with GDI+ methods because GDI+ doesn't support XOR drawing. For drawing any shape other than a line and a rectangle in rubber-banding mode, you must call the native Windows GDI functions though PInvoke.)

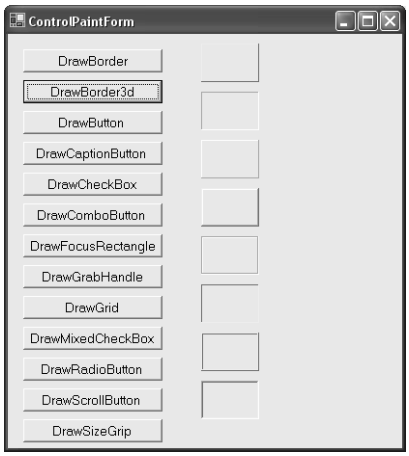

Figure 18-7 The demo application lets you preview the effect you can achieve by invoking some of the methods in the ControlPaint class.

Advanced Topics

Windows Forms control creation is a complex topic, and I don't have enough space to cover every detail. But what you learned in previous chapters and the techniques I'll cover in this section are more than sufficient to enable you to author useful and complex controls with relatively little effort.

The ISupportInitialize Interface

If you create a control or a component that is meant to interact with other controls on the form, you might have the following problem: the control hasn't been sited on the form's surface when its constructor method runs, so you can't reference the Container property from inside that method. And even if you could, your code couldn't see any controls that are added to the form after it.

You can easily solve these and other similar problems simply by having your control or component expose the System.ComponentModel.SupportInitialize interface. When Visual Studio .NET adds your control to the form, it will invoke the ISupportInitialize.BeginInit method before any control is added to the form, and the ISupportInitialize.EndInit method after all controls have been added to the form. Several built-in controls expose this interface, including the DataGrid, Timer, and NumericUpDown controls, and the DataSet, DataTable, and FileSystemWatcher components—for example, look at the code that Visual Studio .NET generates when you drop a DataGrid control on a form's surface:

```
<System.Diagnostics.DebuggerStepThrough()> Private Sub InitializeComponent()
    Me.TextBox1 = New System.Windows.Forms.TextBox
    Me.DataGrid1 = New System.Windows.Forms.DataGrid
    CType(Me.DataGrid1, ISupportInitialize).BeginInit()
    Me.SuspendLayout()
    ' ...(Assign properties of all the controls and the form itself)...
    ⋮
    ' ...(Add controls to the parent form)...
    Me.Controls.Add(Me.TextBox1)
    Me.Controls.Add(Me.DataGrid1)
    CType(Me.DataGrid1, ISupportInitialize).EndInit()
    Me.ResumeLayout(False)
End Sub
```

Here's how this interface looks when implemented in a component:

```
Public Sub BeginInit() Implements ISupportInitialize.BeginInit
    ' Code that runs before any other control is hosted on the form.
End Sub

Public Sub EndInit() Implements ISupportInitialize.EndInit
    ' Code that runs after any other control is hosted on the form.
End Sub
```

Multithreaded Controls

Creating a multithreaded control class isn't different from creating a new thread in a regular application, and you have several options: you can create a new Thread object, use a thread from the thread pool, or just use asynchronous method invocation. (Threading is covered in Chapter 12.) The only potential glitch you should watch for is that the control you create—whether it's inherited from a Control, a UserControl, or another control—must be accessed *exclusively* from the thread that created it. In fact, all the Windows Forms controls rely on the single-threaded apartment (STA) model because windows and controls are based on the Win32 message architecture, which is inherently apartment-threaded. This means that a control (or a form, for that matter) can be created on any thread, but all the methods of the control must be called from the thread that created the control. This constraint can create a serious problem because other .NET portions use the free-threading model, and carelessly mixing the two models isn't a wise idea.

The only methods that you can call on a control object from another thread are Invoke, BeginInvoke, and EndInvoke. You already know from Chapter 12 how to use the latter two methods for calling a method asynchronously, so in this section I'll focus on the Invoke method exclusively. This method takes a delegate pointing to a method (Sub or Function) and can take an Object array as a second argument if the method expects one or more arguments.

To illustrate how to use this method, I've prepared a CountdownLabel control, which continuously displays the number of seconds left until the countdown expires. (See Figure 18-8.) The code that updates the Label runs on another thread. You start the countdown by using the StartCountdown method, and you can stop it before the end by using the StopCountdown method.

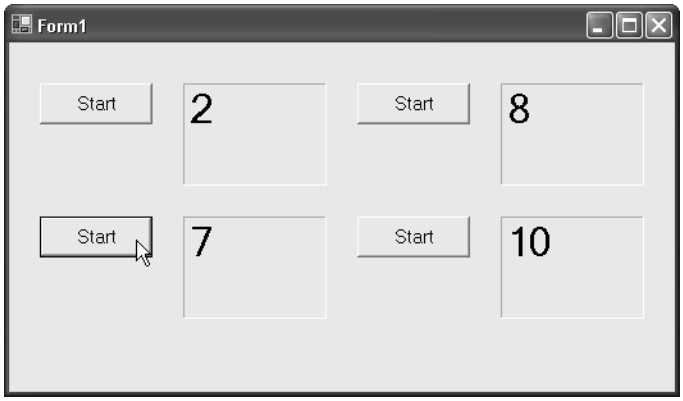

Figure 18-8 A form with four CountdownLabel instances

Here are the steps you must take to correctly implement multithreading in a control:

1. Define a private method that operates on the control or its properties. This method runs in the control's main thread and can therefore access all the control's members. In the sample control, this method is named SetText and takes the string to be displayed in the Label control.
2. Declare a delegate patterned after the method defined in step 1; in the sample control, this is called SetTextDelegate.
3. Declare a delegate variable and make it point to the method defined in step 1; this value is passed to the Invoke method. In the sample control, this variable is named SetTextMarshaler: the name comes from the fact that this delegate actually marshals data from the new thread to the control's main thread.
4. Create a method that spawns the new thread using one of the techniques described in Chapter 12. For simplicity's sake, the sample application uses the Thread object; in the sample control, this task is performed by the StartCountdown method.

Here's the complete listing of the CountdownLabel control:

```
Public Class CountdownLabel
    Inherits System.Windows.Forms.Label

    ' A delegate that points to the SetText procedure
    Delegate Sub SetTextDelegate(ByVal Text As String)
    ' An instance of this delegate that points to the SetText procedure
    Dim SetTextMarshaler As SetTextDelegate = AddressOf Me.SetText
    ' The internal counter for number of seconds left
    Dim secondsLeft As Integer
    ' The end time for countdown
    Dim endTime As Date
    ' The thread object: if Nothing, no other thread is running.
    Dim thr As Thread

    Sub StartCountdown(ByVal seconds As Integer)
        ' Wait until all variables can be accessed safely.
        SyncLock Me
            ' Save values where the other thread can access them.
            secondsLeft = seconds
            endTime = Now.AddSeconds(seconds)

            ' Create a new thread, and run the procedure on that thread
            ' only if the thread isn't running already.
            If (thr Is Nothing) Then
                thr = New Thread(AddressOf CountProc)
                thr.Start()
            End If
        End SyncLock

        ' Display the initial value in the label.
        SetText(CStr(seconds))
    End Sub

    Sub StopCountdown()
        SyncLock Me
            ' This statement implicitly causes CountProc to exit.
            endTime = Now
        End SyncLock
    End Sub

    ' This procedure is just a wrapper for a simple property set and runs
    ' on the control's creation thread. The other thread(s) must call it
    ' through the control's Invoke method.
    Private Sub SetText(ByVal Text As String)
        Me.Text = Text
    End Sub

    ' This procedure runs on another thread.
    Private Sub CountProc()
        Do
            ' Ensure that this is the only thread that is accessing variables.
            SyncLock Me
```

```
                ' Calculate the number of seconds left.
                Dim secs As Integer = CInt(endTime.Subtract(Now).TotalSeconds)
                ' If different from current value, update the Text property.
                If secs <> secondsLeft Then
                    ' Never display negative numbers.
                    secondsLeft = Math.Max(secs, 0)
                    ' Arguments must be passed in an Object array.
                    Dim args() As Object = {CStr(secondsLeft)}
                    ' Update the Text property with current number of seconds.
                    MyBase.Invoke(SetTextMarshaler, args)

                    ' Terminate the thread if countdown is over.
                    If secondsLeft <= 0 Then
                        ' Signal that no thread is running, and exit.
                        thr = Nothing
                        Exit Do
                    End If
                End If
            End SyncLock
            ' Wait for 100 milliseconds.
            Thread.Sleep(100)
        Loop
    End Sub
End Class
```

As usual in multithreaded applications, you must pay a lot of attention to how you access variables shared among threads to prevent your control from randomly crashing after hours of testing. Other problems can arise if the user closes the form while the other thread is running because this extra thread attempts to access a control that no longer exists and would prevent the application from shutting down correctly. You can work around this obstacle by killing the other thread when the control is being destroyed, which you do by overriding the OnHandleDestroyed method:

```
' Kill the other thread if the control is being destroyed.
Protected Overrides Sub OnHandleDestroyed(ByVal e As EventArgs)
    SyncLock Me
        If Not (thr Is Nothing) AndAlso thr.IsAlive Then
            thr.Abort()
            thr.Join()
        End If
    End SyncLock
    MyBase.OnHandleDestroyed(e)
End Sub
```

Remember that you can't use the RaiseEvent statement from another thread. To have the CountdownLabel control fire an event when the countdown is complete, you must adopt the same technique described previously. You must create a method that calls RaiseEvent and runs on the main thread, define a delegate that points to it, and use Invoke from the other thread when you want to raise the event.

The techniques described in this section should be used whenever you access a Windows Forms control from another thread and not just when you're creating a custom control. If you are unsure whether you must use the Invoke method on a control (in

other words, you don't know whether your code is running in the same thread as the control), just check the InvokeRequired read-only property. If it returns True, you must use the technique described in this section.

Extender Provider Controls

You can create property extender controls similar to those provided with the Windows Forms package, such as ToolTip and HelpProvider. In this section, I'll guide you through the creation of a component named UserPropExtender, which adds a property named UserRole to all the visible controls on the form. The developer can assign a user role to this list, or even a semicolon-separated list of roles, such as Manager;Accountants. At run time, the UserPropExtender control makes invisible those controls that are associated with a user role different from the current role (information that you assign to the UserPropExtender control's CurrentUserRole property). Thanks to the UserPropExtender control, you can provide different control layouts for different user roles without writing any code. Here are the steps you must follow when creating an extender provider control:

1. You define a class for your extender provider, making it inherit from System.Windows.Forms.Control (if the new control has a user interface) or from System.ComponentModel.Component (if the new control isn't visible at run time and should be displayed in the component tray of the form designer).
2. Associate a ProvideProperty attribute with the class you created in the preceding step. The constructor for this attribute takes the name of the property that's added to all the controls on the form and a second System.Type argument that defines which type of objects can be extended by this extender provider. In this example, the new property is named UserList and can be applied to Control objects.
3. All extender providers must implement the IExtenderProvider interface, so you must add a suitable Implements statement. This interface has only one method, CanExtend, which receives an Object and is expected to return True if that object can be extended with the new property. The code in the sample control returns True for all control classes except the Form class.
4. Declare and initialize a class-level Hashtable object that stores the value of the UserName property for all the controls on the form, where the control itself is used as a key in the Hashtable. In the sample project, this collection is named userListValues.
5. Define two methods, Get*xxxx* and Set*xxxx*, where *xxxx* is the name of the property that the extender provider adds to all other controls. (These methods are named GetUserName and SetUserName in the sample control.) These methods can read and write values in the Hashtable defined in the preceding step and modify the control's user interface as necessary. (The Set*xxxx* method is invoked from the code automatically generated by the form designer.)

Armed with this knowledge, you should be able to decode the complete listing for the UserPropExtender component quite easily:

```
' Let the form know that this control will add the UserRole property.
<ProvideProperty("UserRole", GetType(Control))> _
Public Class UserPropExtender
    Inherits Component
    Implements IExtenderProvider

    ' Return True for all controls that can be extended with
    ' the UserRole property.
    Public Function CanExtend(ByVal extendee As Object) As Boolean _
        Implements IExtenderProvider.CanExtend
        ' Extend all controls but not forms.
        If Not (TypeOf extendee Is Form) Then
            Return True
        End If
    End Function

    ' The Hashtable object that associates controls with their UserRole
    Dim userRoleValues As New Hashtable()

    ' These are the Get/Set methods related to the property being added.

    Function GetUserRole(ByVal ctrl As Control) As String
        ' Check whether a property is associated with this control.
        Dim value As Object = userRoleValues(ctrl)
        ' Return the value found or an empty string.
        If value Is Nothing Then value = ""
        Return value.ToString
    End Function

    Sub SetUserRole(ByVal ctrl As Control, ByVal value As String)
        ' In case Nothing is passed
        If Value Is Nothing Then Value = ""
        If Value.Length = 0 And userRoleValues.Contains(ctrl) Then
            ' Remove the control from the hash table.
            userRoleValues.Remove(ctrl)
            ' Remove event handlers, if any (none in this example).
            ⋮
        ElseIf Value.Length > 0 Then
            If Not userRoleValues.Contains(ctrl) Then
                ' Add event handlers here (none in this example).
                ⋮
            End If
            ' Assign the new value, and refresh the control.
            userRoleValues.Item(ctrl) = Value
            SetControlVisibility(ctrl)
        End If
    End Sub

    ' This property is assigned the name of the current user.
    Dim m_CurrentUserRole As String
```

```
    Property CurrentUserRole() As String
        Get
            Return m_CurrentUserRole
        End Get
        Set(ByVal Value As String)
            m_CurrentUserRole = Value
            RefreshAllControls()      ' Redraw all controls.
        End Set
    End Property

    ' Hide/show all controls based on their UserRole property.
    Sub RefreshAllControls()
        For Each ctrl As Control In userRoleValues.Keys
            SetControlVisibility(ctrl)
        Next
    End Sub

    ' Hide/show a single control based on its UserRole property.
    Private Sub SetControlVisibility(ByVal ctrl As Control)
        ' Do nothing if no current role or control isn't in the hash table.
        If CurrentUserRole = "" Then Exit Sub
        If Not userRoleValues.Contains(ctrl) Then Exit Sub

        ' Get the value in the hash table.
        Dim value As String = userRoleValues(ctrl).ToString
        ' Check whether current role is among the role(s) defined
        ' for this control.
        If InStr(";" & value & ";", ";" & CurrentUserRole & ";", _
            CompareMethod.Text) > 0 Then
            ctrl.Visible = True
        Else
            ctrl.Visible = False
        End If
    End Sub
End Class
```

The UserPropExtender control is fully functional and uses many of the techniques you should know about when writing extender providers. But because of its simplicity, it doesn't need to trap events coming from other controls on the form, which is often a requirement for extender providers. For example, the ToolTip control intercepts mouse events for all controls that have a nonempty ToolTip property, and the HelpProvider control intercepts the HelpRequested event to display the associated help page or string.

Intercepting events from controls isn't difficult; however, when the control is added to the Hashtable (typically in the Set*xxxx* method), you use the AddHandler command to have one of its events trapped by a local procedure, and you use RemoveHandler to remove the event added dynamically when the control is removed from the Hashtable. Remarks in the preceding listing clearly show where these statements should be inserted.

Custom Property Editors

If you're familiar with custom control authoring under Visual Basic 6, you might have noticed that I haven't mentioned property pages, either when describing built-in controls or in this section devoted to custom control creation. Property pages aren't supported in the .NET architecture and have been replaced by custom property editors.

The most common form of property editor displays a Windows Forms control in a drop-down area inside the Properties window. You can display any control in the drop-down area, such as a TrackBar, a ListBox, or even a complex control such as a TreeView or a DataGrid, but the limitation is that you can display only *one* control. If you want to display more controls, you'll have to create a custom control with multiple child controls for the sole purpose of using it in the drop-down area—for example, the editors used by Anchor and Dock properties work in this way.

The Windows Form designer also supports property editors that display modal forms. Because you're in charge of drawing the appearance of such modal forms, you can display tab pages and multiple controls, and even modify more than one property. For example, you can add OK, Cancel, and Apply buttons and have a Visual Basic 6–like property page. Because these forms are modal, the Property window should be updated only when the user closes them, but you can offer a preview of what the custom control being edited will look like when the property or properties are assigned. As a matter of fact, it's perfectly legal to use your custom control inside the property editor you create for one of its properties, regardless of whether these editors use the drop-down area or a modal form.

Implementing a custom property editor isn't simple. Worse, the MSDN documentation is less than perfect—to use a euphemism—so I had to dig deep in the samples provided with the .NET Framework and use some imagination. The companion source code with this book includes a GradientBackgroundEx control that extends GradientBackground with a new RotateAngle property, which permits you to rotate the gradient brush. (I have purposely chosen to extend an existing control so that I don't need to lead you through all the steps necessary to create a brand-new custom control.) The new property is associated with a custom property editor that uses a TrackBar control in a drop-down area of the Properties window to let the user select the angle with the mouse. Thanks to inheritance, the code for the GradientBackgroundEx control is concise and includes only the new Property procedure and a redefined OnPaint procedure, which differs from the original OnPaint procedure by one statement only, here shown in boldface:

```
Public Class GradientBackgroundEx
  Inherits GradientBackground

  Dim m_RotateAngle As Single
```

```
    <Description("The rotation angle for the brush"), DefaultValue(0)> _
    Property RotateAngle() As Single
        Get
            Return m_RotateAngle
        End Get
        Set(ByVal Value As Single)
            m_RotateAngle = Value
            Me.Invalidate()
        End Set
    End Property

    ' Redefine the OnPaint event to account for the new property.
    Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)
        ' Create a gradient brush as large as the client area, with specified
        ' start/end color and gradient mode.
        Dim br As New LinearGradientBrush(Me.ClientRectangle, _
            Me.StartColor, Me.EndColor, Me.GradientMode)
        ' Apply the rotation angle.
        br.RotateTransform(Me.RotateAngle)
        ' Paint the background and destroy the brush.
        e.Graphics.FillRectangle(br, Me.ClientRectangle)
        br.Dispose()
        ' Let the base control do its chores (e.g., raising the Paint event).
        MyBase.OnPaint(e)
    End Sub
End Class
```

The first step in defining a custom editor for a given property is to specify the editor itself in the Editor attribute associated with the property procedure itself. The editor we're going to create is named RotateAngleEditor, so the new version of the RotateAngle Property procedure becomes:

```
    <Description("The rotation angle for the brush"), DefaultValue(0), _
        Editor(GetType(RotateAngleEditor), GetType(UITypeEditor))> _
    Property RotateAngle() As Single
        ⋮
    End Property
```

Note that the attribute's constructor takes two System.Type arguments, so you must use the GetType function. The second argument is always GetType(UITypeEditor).

The property editor is a class that you define. If you don't plan to reuse this property editor for other custom controls, you can avoid namespace pollution by making the editor a nested class of the custom control class.

The property editor class inherits from System.Drawing.Design.UITypeEditor and must override two methods in its base class, GetEditStyle and EditValue. The form designer calls the GetEditStyle method when it's filling the Properties window with the values of all the properties of the control currently selected. This method must return an enumerated value that tells the designer whether your property editor is going to display a single control in a drop-down area or a modal form. In the former case, a down-arrow button is displayed near the property name; in the latter case, a button with an ellipsis is used instead.

The form designer calls the EditValue method when the user clicks the button beside the property name. This method is overloaded, but we don't have to override all the overloaded versions of the method. In the most general overloaded version—the only one I override in the demo program—this method receives three arguments:

- The first argument is an ITypeDescriptorContext type that can provide additional information about the context in which the editing action is being performed—for example, Context.Instance returns a reference to the control whose property is being edited, and Context.Container returns a reference to the control's container.
- The second argument is an IServiceProvider type. You can query the GetService method of this object to get the editor service object that represents the properties editor; this is an IWindowsFormEditorService object that exposes the three methods that let you open the drop-down area (DropDownControl), close it (CloseDropDown), or display a modal form (ShowDialog).
- The third argument is the current value of the property being edited. You should use this value to correctly initialize the control about to appear in the drop-down area (or the controls on the modal form). The EditValue method is expected to return the new value of the property being edited.

Here's the complete listing of the RotateAngleEditor class. Its many remarks and the details already given should suffice for you to understand how it works:

```
Class RotateAngleEditor
    Inherits UITypeEditor

    ' Override the GetEditStyle method to tell that this editor supports
    ' the DropDown style.
    Overloads Overrides Function GetEditStyle( _
        ByVal context As ITypeDescriptorContext) As UITypeEditorEditStyle
        If Not (context Is Nothing) AndAlso _
            Not (context.Instance Is Nothing) Then
            ' Return DropDown if you have a context and a control instance.
            Return UITypeEditorEditStyle.DropDown
        Else
            ' Otherwise, return the default behavior, whatever it is.
            Return MyBase.GetEditStyle(context)
        End If
    End Function

    ' This is the TrackBar control that is displayed in the editor.
    Dim WithEvents tb As TrackBar
    ' This the editor service that creates the drop-down area
    ' or shows a dialog.
    Dim wfes As IWindowsFormsEditorService
```

```
    ' Override the EditValue function,
    ' and return the new value of the property.
    Overloads Overrides Function EditValue( _
        ByVal context As ITypeDescriptorContext, _
        ByVal provider As IServiceProvider, _
        ByVal value As Object) As Object

        ' Exit if no context, instance, or provider is provided.
        If (context Is Nothing) OrElse (context.Instance Is Nothing) _
            OrElse (provider Is Nothing) Then
            Return value
        End If
        ' Get the Editor Service object; exit if not there.
        wfes = CType(provider.GetService( _
            GetType(IWindowsFormsEditorService)), IWindowsFormsEditorService)
        If (wfes Is Nothing) Then Return value

        ' Create the TrackBar control, and set its properties.
        tb = New TrackBar()
        ' Always set Orientation before Size property.
        tb.Orientation = Orientation.Vertical
        tb.Size = New Size(50, 150)
        tb.TickStyle = TickStyle.TopLeft
        tb.TickFrequency = 45
        tb.SetRange(0, 360)
        ' Initalize its Value property.
        tb.Value = CInt(value)

        ' Show the control. (It returns when the drop-down area is closed.)
        wfes.DropDownControl(tb)
        ' The return value must be of the correct type.
        EditValue = CSng(tb.Value)
        ' Destroy the TrackBar control.
        tb.Dispose()
        tb = Nothing
    End Function

    ' Close the drop-down area when the mouse button is released.
    Private Sub TB_MouseUp(ByVal sender As Object, _
        ByVal e As MouseEventArgs) Handles tb.MouseUp
        If Not (wfes Is Nothing) Then
            wfes.CloseDropDown()
        End If
    End Sub
End Class
```

The RotateAngleEditor class automatically closes the drop-down area when the user releases the mouse button, but this isn't strictly necessary because the Properties window closes the drop-down area when the user clicks somewhere else. I implemented this detail only to show you how you can react to user selections in the drop-down area. Figure 18-9 shows the new property editor in action.

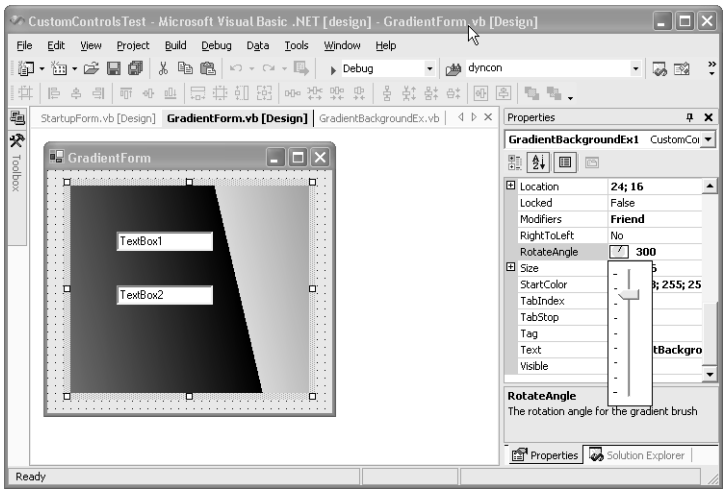

Figure 18-9 The RotateAngle property of a GradientBackgroundEx control is being edited with a custom property editor.

The steps you must take to display a modal form instead of the drop-down area are the same as those you've seen so far, with only two differences:

- The GetEditStyle method must return the UITypeEditorEditStyle.Modal value to let the Property window know that an ellipsis button must be displayed beside the property name.
- Your class library project must contain a form class in which you drop the controls that make up the editor interface. In the EditValue method, you create an instance of this form and pass it to the ShowModal method of the IWindowsFormsEditorService object (instead of the DropDownControl method, as in the preceding code example).

Custom property editors provide support for one more feature: the ability to offer the visual representation of the current value in the Properties window in a small rectangle to the left of the actual numeric or string value. (You can see how the form designer uses such rectangles for the ForeColor, BackColor, and BackgroundImage properties.) In this case, you must override two more methods of the base UITypeEditor class: GetPaintValueSupported (which should return True if you want to implement this feature) and PaintValue (where you place the code that actually draws inside the small rectangle). The latter method receives a PaintValueEventArgs object, whose properties give you access to the Graphics object on which you can draw, the bounding rectangle, and the value to be printed. The following code extends the RotateAngleEditor class with the ability to display a small yellow circle, plus a black line that shows the current value of the RotateAngle property in a visual manner:

```
' Let the property editor know that we want to paint the value.
Overloads Overrides Function GetPaintValueSupported( _
    ByVal context As ITypeDescriptorContext) As Boolean
```

```
    ' In this demo, we return True regardless of the actual editor.
    Return True
End Function

' Display a yellow circle to the left of the value in the Properties window
' with a line forming the same angle as the value of the RotateAngle property.
Overloads Overrides Sub PaintValue(ByVal e As PaintValueEventArgs)
    ' Get the angle in radians.
    Dim a As Single = CSng(e.Value) * CSng(Math.PI) / 180!
    ' Get the rectangle in which we can draw.
    Dim rect As Rectangle = e.Bounds
    ' Evaluate the radius of the circle.
    Dim r As Single = Math.Min(rect.Width, rect.Height) / 2!
    ' Get the center point.
    Dim p1 As New PointF(rect.Width / 2!, rect.Height / 2!)
    ' Calculate where the line should end.
    Dim p2 As New PointF(CSng(p1.X + Math.Cos(a) * r), _
        CSng(p1.Y + Math.Sin(a) * r))
    ' Draw the yellow-filled circle.
    e.Graphics.FillEllipse(Brushes.Yellow, rect.Width / 2! - r, _
        rect.Height / 2! - r, r * 2, r * 2)
    ' Draw the line.
    e.Graphics.DrawLine(Pens.Black, p1, p2)
End Sub
```

You can see the effect in Figure 18-9.

Object Properties

A few common properties, such as Font and Location, return objects instead of scalar values. These properties are displayed in the Properties window with a plus sign (+) to the left of each of their names so that you can expand those items to edit the individual properties of the object. If you implement properties that return objects defined in the .NET Framework (such as Font, Point, and Size objects), your control automatically inherits this behavior. However, when your control exposes an object defined in your application, you must implement a custom TypeConverter class to enable this feature.

The companion source code includes an AddressControl custom control, which lets the user enter information such as street, city, postal code, state, and country. (See the form on the left in Figure 18-10.) Instead of exposing this data as five independent properties, this control exposes them as a single Address object, which is defined in the same application:

```
Public Class Address
    ' This event fires when a property is changed.
    Event PropertyChanged(ByVal propertyName As String)

    ' Private members
    Dim m_Street As String
    Dim m_City As String
    Dim m_Zip As String
    Dim m_State As String
    Dim m_Country As String
```

```
    Property Street() As String
        Get
            Return m_Street
        End Get
        Set(ByVal Value As String)
            If m_Street <> Value Then
                m_Street = Value
                RaiseEvent PropertyChanged("Street")
            End If
        End Set
    End Property

    ' ...(Property procedures for City, Zip, State, and Country
    '     omitted because substantially identical to Street property)...
    ⋮

    Overrides Function ToString() As String
        Return "(Address)"
    End Function
End Class
```

Note that property procedures in the Address class are just wrappers for the five private variables and that they raise a PropertyChanged event when any property changes. The ToString method is overridden to provide the text that will appear as a dummy value for the Address property in the Properties window. The AddressControl control has an Address property that returns an Address object:

```
Public Class AddressControl
    Inherits System.Windows.Forms.UserControl

#Region " Windows Form Designer generated code "
    ⋮
#End Region

    Dim WithEvents m_Address As New Address()

    <TypeConverter(GetType(AddressTypeConverter)), _
        DesignerSerializationVisibility(DesignerSerializationVisibility.Content)> _
    Property Address() As Address
        Get
            Return m_Address
        End Get
        Set(ByVal Value As Address)
            m_Address = Value
            RefreshControls()
        End Set
    End Property

    ' Refresh controls when any property changes.
    Private Sub Address_PropertyChanged(ByVal propertyName As String) _
        Handles m_Address.PropertyChanged
        RefreshControls()
    End Sub
```

```
    ' Display Address properties in the control's fields.
    Private Sub RefreshControls()
        txtStreet.Text = m_Address.Street
        txtCity.Text = m_Address.City
        txtZip.Text = m_Address.Zip
        txtState.Text = m_Address.State
        txtCountry.Text = m_Address.Country
    End Sub

    ' Update a member property when user updates a field.
    Private Sub Controls_TextChanged(ByVal sender As Object, _
     ByVal e As EventArgs) Handles txtStreet.TextChanged, txtCity.TextChanged, _
        txtZip.TextChanged, txtState.TextChanged, txtCountry.TextChanged
        Dim Text As String = DirectCast(sender, Control).Text
        If sender Is txtStreet Then
            m_Address.Street = Text
        ElseIf sender Is txtCity Then
            m_Address.City = Text
        ElseIf sender Is txtZip Then
            m_Address.Zip = Text
        ElseIf sender Is txtState Then
            m_Address.State = Text
        ElseIf sender Is txtCountry Then
            m_Address.Country = Text
        End If
    End Sub
End Class
```

The key statement in the preceding code is the TypeConverter attribute, which tells the form designer that a custom TypeConverter object is associated with the Address property. Or you can associate this attribute with the Address class itself, in which case all the controls that expose a property of type Address will automatically use the AddressTypeConverter class. (The .NET Framework uses this approach for classes such as Point and Font.) The DesignerSerializationVisibility attribute, which is also assigned to the Address property, tells Visual Studio .NET that it must serialize each and every property of the Address object when it generates the source code for initializing this object.

The AddressTypeConverter class derives from TypeConverter and overrides two methods. The GetPropertiesSupported method must return True to let the editor know that a plus symbol must be displayed to the left of the Address property in the Properties window; the GetProperties method must return a PropertyDescriptorCollection object, which describes the items that will appear when the plus symbol is clicked:

```
' The TypeConverter class for the Address property
' (It can be a nested class of AddressControl.)

Public Class AddressTypeConverter
    Inherits TypeConverter

    Overloads Overrides Function GetPropertiesSupported( _
        ByVal context As ITypeDescriptorContext) As Boolean
        ' Tell the editor to display a + symbol near the property name.
```

```
        Return True
    End Function

    Overloads Overrides Function GetProperties( _
        ByVal context As ITypeDescriptorContext, ByVal value As Object, _
        ByVal attributes() As Attribute) As PropertyDescriptorCollection

        ' Use the GetProperties shared method to return a collection of
        ' PropertyDescriptor objects, one for each property of Address.
        Return TypeDescriptor.GetProperties(GetType(Address))
    End Function
End Class
```

You see the effect of this custom TypeConverter class in the right portion of Figure 18-10.

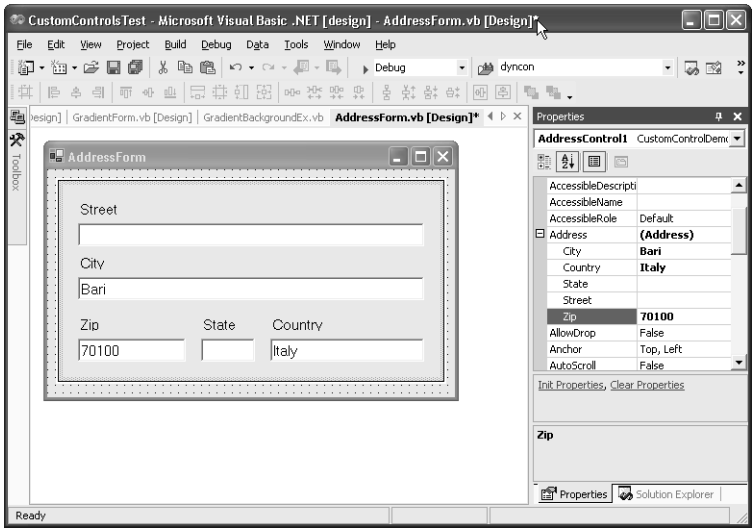

Figure 18-10 The AddressControl control includes a TypeConverter for its custom property, Address.

To keep code as concise as possible, the AddressTypeConverter class uses the TypeDescriptor.GetProperties shared method to create a PropertyDescriptorCollection object that describes all the properties of the Address class. In some cases, you might need to create this collection manually. You must take this step, for example, when not all the properties should be made available in the Properties window, or when you need to define a custom editor for one of them.

A custom TypeConverter object can do more than add support for object properties. For example, you can use a custom TypeConverter class to validate the string entered in the Properties window or to convert this string to a value type other than a standard numeric and date .NET type. For more information about custom TypeConverter classes, see the MSDN documentation.

Custom Control Designers

You surely noticed that a few Windows Forms controls display one or more hyperlinks near the bottom edge of the Properties window, or additional commands on the context menu that appears when you right-click on them. Implementing this and a few other features is relatively simple, and requires that you create a custom control designer and associate it with your custom control.

A control designer is a class that inherits from System.Windows.Forms.Design.ControlDesigner class, which is defined in the System.Design.dll assembly. (You must add a reference to this assembly because Visual Basic .NET projects don't reference it by default.) Providing one or more commands in the Property window or in the context menu requires that you override the Verbs read-only property. In the demo application, I created a control designer for the AddressControl; my custom designer adds two verbs to the control, InitProperties and ClearProperties:

```
Class AddressControlDesigner
    Inherits System.Windows.Forms.Design.ControlDesigner

    ' Return a collection of verbs for this control
    Public Overrides ReadOnly Property Verbs() As DesignerVerbCollection
        Get
            Dim col As New DesignerVerbCollection
            col.Add(New DesignerVerb("Init Properties", AddressOf InitProperties))
            col.Add(New DesignerVerb("Clear Properties", AddressOf ClearProperties))
            Return col
        End Get
    End Property
    ⋮
End Class
```

When the user clicks on the link in the Properties window or selects the verb from the context menu, Visual Studio .NET invokes the handler pointed to by the corresponding delegate. The ClearProperties routine simply resets the control's Address property, but you must take additional steps to let Visual Studio .NET know that you assigned a new value to a property. I encapsulated these steps in the SetAddressProperty private procedure, so that you can easily adapt my code to other procedures:

```
    Sub ClearProperties (ByVal sender As Object, ByVal e As EventArgs)
        SetAddressProperty(New Address)
    End Sub

    ' Assign a new Address property, raise all expected events.
    Private Sub SetAddressProperty(ByVal newValue As Address)
        Dim thisCtrl As AddressControl = CType(Me.Control, AddressControl)
        ' Let Visual Studio know we're about to change the Address property.
        Dim pdCol As PropertyDescriptorCollection = _
          TypeDescriptor.GetProperties(thisCtrl)
        Dim pd As PropertyDescriptor = pdCol.Find("Address", True)
        RaiseComponentChanging(pd)
        ' Assign the value, but remember old value.
```

```
        Dim oldValue As Address = thisCtrl.Address
        thisCtrl.Address = newValue
        ' Let Visual Studio know we've done the assignment.
        RaiseComponentChanged(pd, oldValue, thisCtrl.Address)
    End Sub
```

The InitProperties procedure does something more interesting: it creates a form that displays another instance of the AddressControl class so that the programmer can change the Street, City, Zip, State, and Country properties simply by typing in the control:

```
    Sub InitPropertiesHandler(ByVal sender As Object, ByVal e As EventArgs)
        ' Create a new control that points to the same Address object.
        Dim thisCtrl As AddressControl = CType(Me.Control, AddressControl)
        Dim newCtrl As New AddressControl
        newCtrl.Address = thisCtrl.Address
        ' Display a form that displays the new control.
        Dim frm As New Form
        frm.Text = "Set AddressControl's Address members"
        frm.ClientSize = newCtrl.Size
        frm.Controls.Add(newCtrl)
        frm.ShowDialog()
        ' Raise all required events.
        SetAddressProperty(thisCtrl.Address)
    End Sub
```

You complete your job by flagging the AddressControl class with an appropriate Designer attribute:

```
<Designer(GetType(AddressControlDesigner))> _
Public Class AddressControl
⋮
End Class
```

Figure 18-11 shows what happens when the programmer clicks on the Init Properties command. Interestingly, the AddressControl reacts to the PropertyChanged events of the Address class; therefore, the underlying control is immediately updated as you type in the foreground form.

A control designer can do more than provide custom commands. For example, you can react to the mouse hovering over the control at design time by overriding the OnMouseEnter and OnMouseLeave protected methods of the ControlDesigner class, or you can limit the way a control can be resized by overriding the SelectionRules property—for example, the AddressControl doesn't know how to resize its constituent controls, so you might make it visible and moveable, but not resizable:

```
    Public Overrides ReadOnly Property SelectionRules() As SelectionRules
        Get
            Return SelectionRules.Moveable Or SelectionRules.Visible
        End Get
    End Property
```

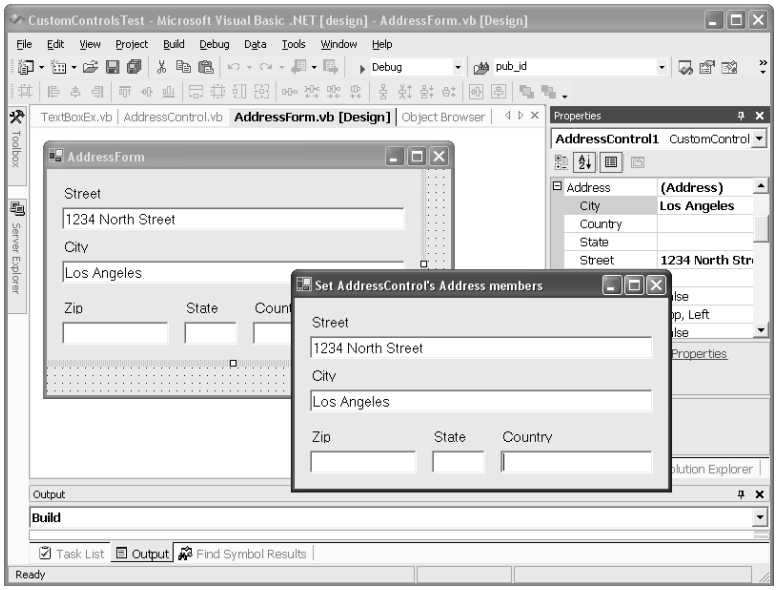

Figure 18-11 You can add verbs to the Properties window and the context menu that appears when you right-click a custom control.

> **Note** When you create a component rather than a control, you can associate a custom designer to it by creating a class that inherits from ComponentDesigner instead of ControlDesigner. For an example of a component that uses a custom designer, see the MessageBoxDisplayer component introduced in Chapter 13.

Data-Binding Support

Chances are that you want to add data-binding support to your custom controls, and the good news is that in most cases you don't need to do anything to support it. In fact, because data binding is offered by the Windows Forms infrastructure, users of your control can bind any public property simply by clicking on the (Advanced) button under (DataBindings) in the Property window, and select which property is bound to which field in any data source defined on the form. (Of course, you have to do the binding via code if you wish to bind to a component that doesn't appear in the parent form's tray area.)

In practice, however, you can expect that your users will want to bind only one or two properties in the control. This is the case of the Text property for TextBox-like controls, or the Checked property for CheckBox-like controls. You can make this task much simpler if you mark this property (or properties) with the Bindable attribute:

```
<Bindable(True)> _
Public Property Caption() As String
    ⋮
End Property
```

When you do this, the property appears under (DataBindings) and users don't have to search for it in the list of all bindable properties.

Things become more interesting (and complicated) if your control must support complex data binding, the type of binding that controls such as ListBox and DataGrid support. To let users bind your control to a DataSet or a DataTable, as they do with built-in controls, you should expose a DataSource property of the IListSource type:

```
Private m_DataSource As IListSource

<Category("Data")> _
Public Property DataSource() As IListSource
    Get
        Return m_DataSource
    End Get
    Set(ByVal Value As IListSource)
        SetDataObject(Value, DataMember)
        m_DataSource = Value
    End Set
End Property
```

You should also expose a DataMember property to fully support the DataSet as a potential data source. DataMember is a string property, but built-in controls let users select it among all the members exposed by the object assigned to the DataSource property. It took some investigation with ILDASM for me to discover what attribute gives the desired effect:

```
Private m_DataMember As String

<Category("Data"), DefaultValue(""), _
Editor("System.Windows.Forms.Design.DataMemberListEditor, System.Design,
    Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a", _
    "System.Drawing.Design.UITypeEditor,System.Drawing,
    Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a")> _
    Public Property DataMember() As String
        Get
            Return m_DataMember
        End Get
        Set(ByVal Value As String)
            SetDataObject(DataSource, Value)
            m_DataMember = Value
        End Set
    End Property
```

It's essential that the two long boldface strings be entered on the same logical line, though they are split here for typographical reasons. Also, keep in mind that these strings are valid only for version 1.1 of the .NET Framework.

The code for both the DataSource and DataMember properties invoke a common SetDataObject private procedure, which is where the control can get a reference to the actual CurrencyManager associated with the data source:

```
Dim WithEvents CurrencyManager As CurrencyManager

Private Sub SetDataObject(ByVal dataSource As IListSource, ByVal dataMember As String)
    ' Do nothing at design-time.
    If Me.DesignMode Then Exit Sub

    Dim cm As CurrencyManager
    ' Find a reference to the CurrencyManager (Throws if invalid).
    If dataSource Is Nothing Or Me.BindingContext Is Nothing Then
        cm = Nothing
    ElseIf Not dataSource.ContainsListCollection Then
        ' Ignore DataMember if data source doesn't contain a collection.
        cm = DirectCast(Me.BindingContext(dataSource), CurrencyManager)
    Else
        ' Nothing to do if DataMember hasn't been set.
        If dataMember = "" Then Exit Sub
        cm = DirectCast(Me.BindingContext(dataSource, dataMember), CurrencyManager)
    End If

    If Not (cm Is Me.CurrencyManager) Then
        ' Only if the CurrencyManager has actually changed.
        Me.CurrencyManager = cm
        ReadDataSchema()
        DisplayData()
    End If
End Sub
```

The control reads all data from the data source and displays them in the ReadDataSchema and DisplayData procedures, respectively. The example included on the companion CD is a data-bound ListView control that displays all the columns in a data source. (See Figure 18-12.) Here's how you can read the schema of the underlying data source and fill the control with data:

```
Sub ReadDataSchema()
    If Me.CurrencyManager Is Nothing Then Exit Sub
    Me.Columns.Clear()
    ' Add one ListView column for each property in data source.
    For Each pd As PropertyDescriptor In Me.CurrencyManager.GetItemProperties
        Me.Columns.Add(pd.Name, 100, HorizontalAlignment.Left)
    Next
End Sub

Private Sub DisplayData()
    If Me.CurrencyManager Is Nothing Then Exit Sub
    ' Get the list of values managed by the CurrencyManager.
    Dim innerList As IList = Me.CurrencyManager.List
    If innerList Is Nothing Then Exit Sub

    ' Iterate over all the rows in the data source.
    For index As Integer = 0 To innerList.Count - 1
        ' Iterate over all columns in the data source.
        Dim currItem As ListViewItem = Nothing
        For Each pd As PropertyDescriptor In Me.CurrencyManager.GetItemProperties
            ' Get the value of this property for Nth row.
```

```
            Dim value As Object = pd.GetValue(innerList(index))
            ' Add as a ListView item or subitem.
            If currItem Is Nothing Then
                currItem = Me.Items.Add(value.ToString)
            Else
                currItem.SubItems.Add(value.ToString)
            End If
        Next
    Next
End Sub
```

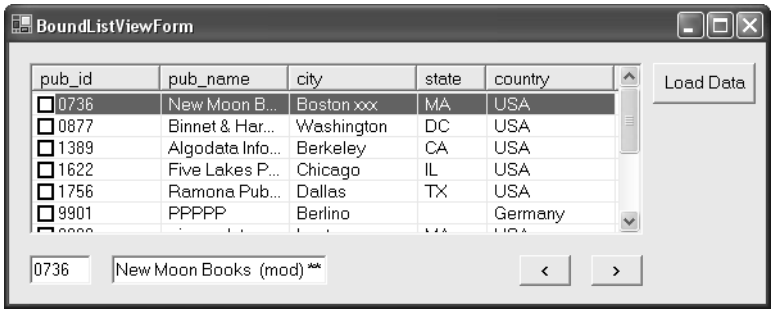

Figure 18-12 The DataBoundListView control is an example of how you can implement complex data binding.

For a fully functional data-bound control you must ensure that the control is always in sync with the CurrencyManager. In other words, a new row in the data source becomes current when the user selects a different row in your control, and a new row in your control becomes selected when the user moves to another row by some other means (for example, via navigational buttons). The DataBoundListView control achieves this synchronization by overriding the OnSelectedIndexChanged protected method and by trapping the CurrencyManager's PositionChanged event. Please see the sample code for more details on how you can complete these tasks and how you can write edited values back in the data source.

Design-Time and Run-Time Licensing

If you plan to sell your custom controls and components, you probably want to implement some type of licensing for them to enforce restrictions on their use. The good news is that the .NET Framework comes with a built-in licensing scheme, but you can override it to create your own licensing method.

To explain how licensing works, I'll show how to create a license provider class named LicWindowsFileLicenseProvided, which checks the design-time license of a control and refuses to load the control in the Visual Studio .NET environment after a given expiration date. This expiration date must be stored in a text file named *controlname*.lic stored in c:\Windows\System32 directory (more in general, the path of your Windows System32 directory). The first line of this file must be in this format:

```
controlname license expires on expirationdate
```

where *controlname* is the full name of the control and *expirationdate* is the date of the end of the license agreement. For example, according to this license schema, a license file for the AddressControl in the CustomControlDemo namespace must be named CustomControlDemo.AddressControl.lic. If the license for this control expires on January 1, 2006, the first line of this file must be

```
CustomControlDemo.AddressControl license expires on 01/01/2006
```

(Character case is significant.) You must do two things to apply a license provider to the class that must be licensed. First, you decorate the class definition with a LicenseProvider attribute. Second, you query the .NET licensing infrastructure from inside the control's constructor method by using the LicenseManager.Validate shared method, which throws an exception if the .lic file can't be found or its contents aren't correct. If successful, the method returns a License object, which you later dispose of when the object is destroyed. Here's a revised version of the AddressControl class that employs the licensing mechanism based on the LicWindowsFileLicenseProvider custom license provider. (Added lines are in boldface.)

```
<LicenseProvider(GetType(LicWindowsFileLicenseProvider))> _
Public Class AddressControl
    Inherits System.Windows.Forms.UserControl

    ' The License object
    Private lic As License

    Public Sub New()
        MyBase.New()
        ' Validate the License.
        lic = LicenseManager.Validate(GetType(AddressControl), Me)
        ' This call is required by the Windows Form Designer.
        InitializeComponent()
    End Sub

    ' Destroy the License object without waiting for the garbage collection.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            ' Destroy the license.
            If Not (lic Is Nothing) Then
                lic.Dispose()
                lic = Nothing
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub
    ⋮
End Class
```

Alternatively, you can use the LicenseManager.IsValid shared method to check a license object without throwing an exception if the license isn't valid.

Obviously, this licensing mechanism is easy to violate, in that a user might change the expiration date in the .lic file to postpone the license expiration, but you can use this example to create a more robust schema. For example, you might encrypt the license text and store it in a nonobvious location (including the registry).

A custom license provider is a class that derives from the abstract LicenseProvider class and overrides the virtual GetLicense method. This method is indirectly called by the LicenseManager.Validate method (in the custom control's constructor method) and receives several arguments that let you determine the name of the control or component in question, whether you're at design time or run time, and other pieces of information. The Windows Forms infrastructure expects the GetLicense method to return a license object if it's OK to use the custom control or component; otherwise, it returns Nothing or throws a LicenseException object. (You decide whether the method should return Nothing or throw the exception by inspecting the allowException argument passed to the GetLicense method.) Here's the implementation of the LicWindowsFileLicenseProvided class:

```
Class LicWindowsFileLicenseProvider
    Inherits LicenseProvider

    Public Overrides Function GetLicense(ByVal context As LicenseContext, _
        ByVal typ As System.Type, ByVal instance As Object, _
        ByVal allowExceptions As Boolean) As License

        ' This is the name of the control.
        Dim ctrlName As String = typ.FullName

        If context.UsageMode = LicenseUsageMode.Designtime Then
            ' We are in design mode.
            ' Check that there is a .lic file in Windows system directory.
            ' Build the full path of the .lic file.
            Dim filename As String = Environment.SystemDirectory() _
                & "\" & ctrlName & ".lic"
            ' This is the text that we expect at the beginning of file.
            Dim licenseText As String = ctrlName & " license expires on "
            Dim fs As System.IO.StreamReader
            Try
                ' Open and read the license file (throws exception if not found).
                fs = New System.IO.StreamReader(filename)
                ' Read its first line.
                Dim text As String = fs.ReadLine
                ' Throw if it doesn't match the expected text.
                If Not text.StartsWith(licenseText) Then Throw New Exception
                ' Parse the expiration date (which follows the expected text).
                Dim expireDate As Date = _
                  Date.Parse(text.Substring(licenseText.Length))
                ' Throw if license has expired.
                If Now > expireDate Then Throw New Exception
            Catch ex As Exception
```

```
                ' Throws a LicenseException or just returns Nothing.
                If allowExceptions Then
                    Throw New LicenseException(typ, instance, _
                        "Can't find design-time license for " & ctrlName)
                Else
                    Return Nothing
                End If
            Finally
                ' In all cases, close the StreamReader.
                If Not (fs Is Nothing) Then fs.Close()
            End Try

            ' If we get here, we can return a RuntimeLicense object.
            Return New DesignTimeLicense(Me, typ)
        Else
            ' We enforce no licensing at run time,
            ' so we always return a RunTimeLicense.
            Return New RuntimeLicense(Me, typ)
        End If
    End Function
End Class
```

In general, you should return two different license objects, depending on whether you're at design time or run time. (Otherwise, a malicious and clever user might use the run-time license at design time.) A license object is an instance of a class that derives from System.ComponentModel.License and overrides its LicenseKey and Dispose virtual methods. Here's a simple implementation of the DesignTimeLicense and RuntimeLicense classes referenced by the preceding code snippet. (These classes can be nested inside the LicWindowsFileLicenseProvider class.)

```
Class LicWindowsFileLicenseProvider
    ⋮
    ' Nested class for design-time license

    Public Class DesignTimeLicense
        Inherits License

        Private owner As LicWindowsFileLicenseProvider
        Private typ As Type

        Sub New(ByVal owner As LicWindowsFileLicenseProvider, ByVal typ As Type)
            Me.owner = owner
            Me.typ = typ
        End Sub

        Overrides ReadOnly Property LicenseKey() As String
            Get
                ' Just return the type name in this demo.
                Return typ.FullName
            End Get
        End Property
```

```
        Overrides Sub Dispose()
            ' There is nothing to do here.
        End Sub
    End Class

    ' Nested class for run-time license

    Public Class RuntimeLicense
        Inherits License

        Private owner As LicWindowsFileLicenseProvider
        Private typ As Type

        Sub New(ByVal owner As LicWindowsFileLicenseProvider, ByVal typ As Type)
            Me.owner = owner
            Me.typ = typ
        End Sub

        Overrides ReadOnly Property LicenseKey() As String
            Get
                ' Just return the type name in this demo.
                Return typ.FullName
            End Get
        End Property

        Overrides Sub Dispose()
            ' There is nothing to do here.
        End Sub
    End Class
End Class
```

Read the remarks in the demo project to see how to test the LicWindowsFileLicenseProvider class with the AddressControl custom control. Figure 18-13 shows the kind of error message you see in Visual Studio if you attempt to load a form that contains a control for which you don't have a design-time license.

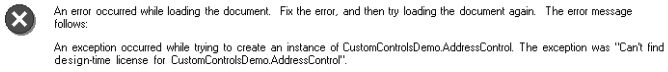

Figure 18-13 The error message that Visual Studio displays when you load a form containing a control for which you don't have a design-time license

Hosting Custom Controls in Internet Explorer

Windows Forms controls have one more intriguing feature that I have yet to discuss: you can host a Windows Forms control in Internet Explorer in much the same way that you can use an ActiveX control, with an important improvement over ActiveX controls—Windows Forms controls don't require any registration on the client machine. However, the .NET Framework must be installed on the client, so in practice you can adopt this technique only for intranet installations.

You specify a Windows Forms control in an HTML page using a special <OBJECT> tag that contains a CLASSID parameter pointing to the DLL containing the control. For example, the following HTML page hosts an instance of the TextBoxEx control. You can see the effect in Figure 18-14.

```
<HTML><BODY>
<H2>Loading the TextBoxEx custom control in IE</H2>

Enter a 5-digit value:<BR>
<OBJECT ID="TextBoxEx1"
    CLASSID="/CustomControlDemo.dll#CustomControlDemo.TextBoxEx"
    HEIGHT=80 WIDTH="300" VIEWASTEXT>
        <PARAM NAME="Text" VALUE="12345">
</OBJECT>
<P><INPUT type="button" value="Clear Text" onClick="ClearText()">

<SCRIPT>
function ClearText() {
        TextBoxEx1.Text = "";
}
</SCRIPT>
</BODY></HTML>
```

The boldface portions show how you insert the control in the page, initialize its properties using the <PARAM> tag, and access its properties programmatically from a JScript function. The value of the CLASSID attribute is the path of the DLL hosting the control, followed by a pound sign (#) and the complete name of the control in question. In the preceding example, the DLL is in the root virtual directory, but it can be located anywhere in the virtual directory tree of Internet Information Services (IIS). When testing the preceding code, remember that you must deploy the HTML file in an IIS directory and access it from Internet Explorer using HTTP. Nothing happens if you display it by double-clicking the file from inside Windows Explorer.

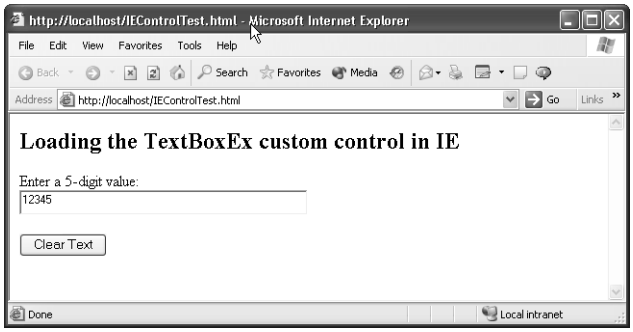

Figure 18-14 Hosting the TextBoxEx control in Internet Explorer 6

Using a control hosted in Internet Explorer has other limitations. For example, you can't use a control stored in the local machine's GAC and you can't use a CAB file to

host multiple assemblies because Internet Explorer 6 is able to deal only with CAB files containing only one assembly. There are limitations also when accessing the control programmatically. For one, you can't access object properties (such as the Address property of the AddressControl sample described in this chapter) because the runtime creates a COM Callable Wrapper (CCW) to make the control visible to the script, but it doesn't create a CCW for dependent properties. (Read Chapter 30 for more details about the CCW.)

Finally, the IIS virtual directory containing the control must have its security set to Scripts only. Any other value, including Scripts and Executables, will prevent the control from loading. The code running in the control appears to the runtime as having been downloaded from the Internet; therefore, many operations won't be allowed unless you grant this assembly broader privileges. (You can learn more about these security-related limitations in Chapter 33.)

Note An ASP.NET page can check whether the client browser supports the .NET runtime by querying the HTTPRequest.Browser.ClrVersion property. A well-behaved ASP.NET application might use ActiveX controls, DHTML, or plain HTML if the browser isn't able to host Windows Forms controls.

At the end of our exploration of built-in controls, GDI+, and custom Windows Forms controls, you should be able to create truly functional and eye-catching Win32 applications. In the next chapter, you'll see how you can create even more powerful Win32 programs by coding against a few .NET components that have no user interface.