PART IV

Templates and Custom Elements

CHAPTER 17

Control Templates

In the past, Windows developers were forced to choose between convenience and flexibility. For maximum convenience, they could use prebuilt controls. These controls worked well enough, but they offered limited customization and almost always had a fixed visual appearance. Occasionally, some controls provided a less-than-intuitive "owner drawing" mode that allowed developers to paint a portion of the control by responding to a callback. But the basic controls—buttons, text boxes, check boxes, list boxes, and so on—were completely locked down.

As a result, developers who wanted a bit more pizzazz were forced to build custom controls from scratch. This was a problem—not only was it slow and difficult to write the required drawing logic by hand, but custom control developers also needed to implement basic functionality from scratch (such as selection in a text box or key handling in a button). And even after the custom controls were perfected, inserting them into an existing application involved a fairly significant round of editing, which would usually necessitate changes in the code (and more rounds of testing). In short, custom controls were a necessary evil—they were the only way to get a modern, distinctive interface, but they were also a headache to integrate into an application and support.

WPF solves the control customization problem with styles (which you considered in Chapter 11) and templates (which you'll begin exploring in this chapter). These features work so well because of the dramatically different way that controls are implemented in WPF. In previous user interface technologies, such as Windows Forms, commonly used controls aren't actually implemented in .NET code. Instead, the Windows Forms control classes wrap core ingredients from the Win32 API, which are untouchable. But as you've already learned, in WPF every control is composed in pure .NET code, with no Win32 API glue in the background. As a result, it's possible for WPF to expose mechanisms (styles and templates) that allow you to reach into these elements and tweak them. In fact, *tweak* is the wrong word because, as you'll see in this chapter, WPF controls allow the most radical redesigns you can imagine.

Understanding Logical Trees and Visual Trees

Earlier in this book, you spent a great deal of time considering the content model of a window—in other words, how you can nest elements inside other elements to build a complete window.

For example, consider the extremely simple two-button window shown in Figure 17-1. To create this window, you nest a StackPanel control inside a Window. In the StackPanel, you place two Button controls, and inside of each you can add some content of your choice (in this case, two strings). Here's the markup:

```
<Window x:Class="SimpleWindow.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="SimpleWindow" Height="338" Width="356"
    >
    <StackPanel Margin="5">
      <Button Padding="5" Margin="5">First Button</Button>
      <Button Padding="5" Margin="5">Second Button</Button>
    </StackPanel>
</Window>
```

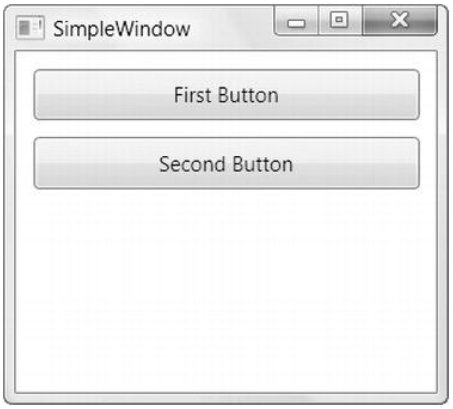

Figure 17-1. *A window with three elements*

The assortment of elements that you've added is called the *logical tree*, and it's shown in Figure 17-2. As a WPF programmer, you'll spend most of your time building the logical tree and then backing it up with event-handling code. In fact, all of the features you've considered so far (such as property value inheritance, event routing, and styling) work through the logical tree.

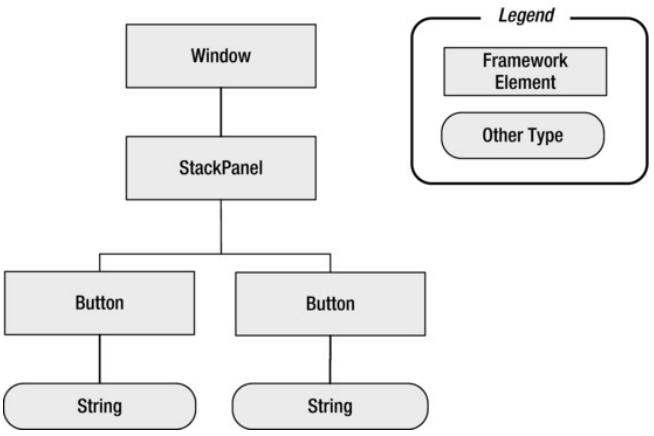

Figure 17-2. *The logical tree for SimpleWindow*

However, if you want to customize your elements, the logical tree isn't much help. Obviously, you could replace an entire element with another element (for example, you could substitute a custom FancyButton class in place of the current Button), but this requires more work, and it could disrupt your application's interface or its code. For that reason, WPF goes deeper with the visual tree.

A *visual tree* is an expanded version of the logical tree. It breaks elements down into smaller pieces. In other words, instead of seeing a carefully encapsulated black box such as the Button control, you see the visual components of that button—the border that gives buttons their signature shaded background (represented by the ButtonChrome class), the container inside (a ContentPresenter), and the block that holds the button text (represented by the familiar TextBlock). Figure 17-3 shows the visual tree for Figure 17-1.

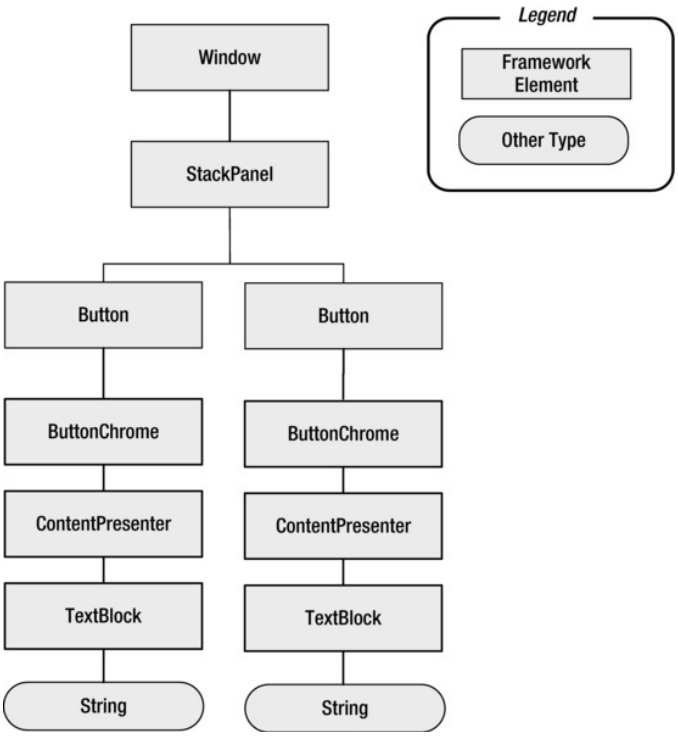

Figure 17-3. *The visual tree for SimpleWindow*

All of these details are themselves elements—in other words, every individual detail in a control such as Button is represented by a class that derives from FrameworkElement.

■ **Note** It's important to realize that there is more than one possible way to expand a logical tree into a visual tree. Details such as the styles you've used and the properties you've set can affect the way a visual tree is composed. For instance, in the previous example, the button holds text content, and as a result, it automatically creates a nested TextBlock element. But as you know, the Button control is a content control, so it can hold any other element you want to use, as long as you nest it inside the button.

So far, this doesn't seem that remarkable. You've just seen that all WPF elements can be decomposed into smaller parts. But what's the advantage for a WPF developer? The visual tree allows you to do two useful things:

- You can alter one of the elements in the visual tree by using styles. You can select the specific element you want to modify by using the Style.TargetType property. You can even use triggers to make changes automatically when control properties change. However, certain details are difficult or impossible to modify.
- You can create a new template for your control. In this case, your control template will be used to build the visual tree exactly the way you want it.

Interestingly enough, WPF provides two classes that let you browse through the logical and visual trees. These classes are System.Windows.LogicalTreeHelper and System.Windows.Media.VisualTreeHelper.

You've already seen the LogicalTreeHelper in Chapter 2, where it allowed you to hook up event handlers in a WPF application with a dynamically loaded XAML document. The LogicalTreeHelper provides the relatively sparse set of methods listed in Table 17-1. Although these methods are occasionally useful, in most cases you'll use the methods of a specific FrameworkElement instead.

Table 17-1. *LogicalTreeHelper Methods*

Name	Description
FindLogicalNode()	Finds a specific element by name, starting at the element you specify and searching down the logical tree.
BringIntoView()	Scrolls an element into view (if it's in a scrollable container and isn't currently visible). The FrameworkElement.BringIntoView() method performs the same trick.
GetParent()	Gets the parent element of a specific element.
GetChildren()	Gets the child element of a specific element. As you learned in Chapter 2, different elements support different content models. For example, panels support multiple children, while content controls support only a single child. However, the GetChildren() method abstracts away this difference and works with any type of element.

The VisualTreeHelper provides a few similar methods—GetChildrenCount(), GetChild(), and GetParent()—along with a small set of methods that are designed for performing lower-level drawing. (For example, you'll find methods for hit testing and bounds checking, which you considered in Chapter 14.)

The VisualTreeHelper also doubles as an interesting way to study the visual tree in your application. Using the GetChild() method, you can drill down through the visual tree of any window and display it for your consideration. This is a great learning tool, and it requires nothing more than a dash of recursive code.

Figure 17-4 shows one possible implementation. Here, a separate window displays an entire visual tree, starting at any supplied object. In this example, another window (named SimpleWindow) uses the VisualTreeDisplay window to show its visual tree.

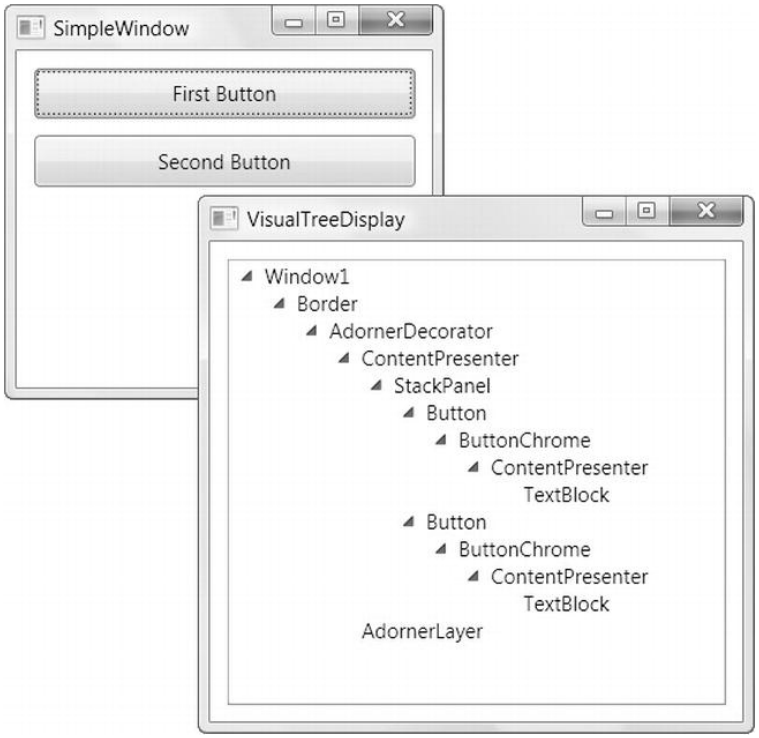

Figure 17-4. *Programmatically examining the visual tree*

Here, a window named Window1 contains a Border, which in turn holds an AdornerDecorator. (The AdornerDecorator class adds support for drawing content in the adorner layer, which is a special invisible region that overlays your element content. WPF uses the adorner layer to draw details such as focus cues and drag-and-drop indicators.) Inside the AdornerDecorator is a ContentPresenter, which hosts the content of the window. That content includes StackPanel with two Button controls, each of which comprises a ButtonChrome (which draws the standard visual appearance of the button) and a ContentPresenter (which holds the button content). Finally, inside the ContentPresenter of each button is a TextBlock that wraps the text you see in the window.

■ **Note** In this example, the code builds a visual tree in another window. If you place the TreeView in the same window as the one you're examining, you'd inadvertently change the visual tree as you fill the TreeView with items.

Here's the complete code for the VisualTreeDisplay window:

```
public partial class VisualTreeDisplay : System.Windows.Window
{
    public VisualTreeDisplay()
    {
        InitializeComponent();
    }
```

```
    public void ShowVisualTree(DependencyObject element)
    {
        // Clear the tree.
        treeElements.Items.Clear();

        // Start processing elements, begin at the root.
        ProcessElement(element, null);
    }

    private void ProcessElement(DependencyObject element,
      TreeViewItem previousItem)
    {
        // Create a TreeViewItem for the current element.
        TreeViewItem item = new TreeViewItem();
        item.Header = element.GetType().Name;
        item.IsExpanded = true;

        // Check whether this item should be added to the root of the tree
        //(if it's the first item), or nested under another item.
        if (previousItem == null)
        {
            treeElements.Items.Add(item);
        }
        else
        {
            previousItem.Items.Add(item);
        }

        // Check whether this element contains other elements.
        for (int i = 0; i < VisualTreeHelper.GetChildrenCount(element); i++)
        {
            // Process each contained element recursively.
            ProcessElement(VisualTreeHelper.GetChild(element, i), item);
        }
    }
}
```

Once you've added this tree to a project, you can use this code from any other window to display its visual tree:

```
VisualTreeDisplay treeDisplay = new VisualTreeDisplay();
treeDisplay.ShowVisualTree(this);
treeDisplay.Show();
```

■ **Tip** You can delve into the visual tree of other applications by using the remarkable Snoop utility, which is available at `http://snoopwpf.codeplex.com`. Using Snoop, you can examine the visual tree of any currently running WPF application. You can also zoom in on any element, survey routed events as they're being executed, and explore and even modify element properties.

Understanding Templates

This look at the visual tree raises a few interesting questions. For example, how is a control translated from the logical tree into the expanded representation of the visual tree?

It turns out that every control has a built-in recipe that determines how it should be rendered (as a group of more fundamental elements). That recipe is called a *control template*, and it's defined by using a block of XAML markup.

■ **Note** Every WPF control is designed to be *lookless*, which means that its visuals (the "look") can be completely redefined. What doesn't change is the control's behavior, which is hardwired into the control class (although it can often be fine-tuned using various properties). When you choose to use a control such as the Button, you choose it because you want button-like behavior (in other words, an element that presents content can be clicked to trigger an action and can be used as the default or cancel button on a window). However, you're free to change the way a button looks and how it reacts when you mouse over it or press it, as well as any other aspect of its appearance and visual behavior.

Here's a simplified version of the template for the common Button class. It omits the XML namespace declarations, the attributes that set the properties of the nested elements, and the triggers that determine how the button behaves when it's disabled, focused, or clicked:

```
<ControlTemplate ... >
  <mwt:ButtonChrome Name="Chrome" ... >
    <ContentPresenter Content="{TemplateBinding ContentControl.Content}" ... />
  </mwt:ButtonChrome>
  <ControlTemplate.Triggers>
    ...
  </ControlTemplate.Triggers>
</ControlTemplate>
```

Although we haven't yet explored the ButtonChrome and ContentPresenter classes, you can easily recognize that the control template provides the expansion you saw in the visual tree. The ButtonChrome class defines the standard button visuals, while the ContentPresenter holds whatever content you've supplied. If you wanted to build a completely new button (as you'll see later in this chapter), you simply need to create a new control template. In place of ButtonChrome, you'd use something else—perhaps your own custom element or a shape-drawing element like the ones described in Chapter 12.

■ **Note** ButtonChrome derives from Decorator (much like the Border class). That means it's designed to add a graphical embellishment around another element—in this case, around the content of a button.

The triggers control how the button changes when it is focused, clicked, and disabled. There's actually nothing particularly interesting in these triggers. Rather than perform the heavy lifting themselves, the focus and click triggers simply modify a property of the ButtonChrome class that provides the visuals for the button:

```
<Trigger Property="UIElement.IsKeyboardFocused">
  <Setter Property="mwt:ButtonChrome.RenderDefaulted" TargetName="Chrome">
    <Setter.Value>
```

```
        <s:Boolean>True</s:Boolean>
      </Setter.Value>
    </Setter>
    <Trigger.Value>
      <s:Boolean>True</s:Boolean>
    </Trigger.Value>
  </Trigger>
  <Trigger Property="ToggleButton.IsChecked">
    <Setter Property="mwt:ButtonChrome.RenderPressed" TargetName="Chrome">
      <Setter.Value>
        <s:Boolean>True</s:Boolean>
      </Setter.Value>
    </Setter>
    <Trigger.Value>
      <s:Boolean>True</s:Boolean>
    </Trigger.Value>
  </Trigger>
```

The first trigger ensures that when the button receives focus, the RenderDefaulted property is set to true. The Second trigger ensures that when the button is clicked, the RenderPressed property is set to true. Either way, it's up to the ButtonChrome class to adjust itself accordingly. The graphical changes that take place are too complex to be represented by a few property setter statements.

Both of the Setter objects in this example use the TargetName property to act upon a specific piece of a control template. This technique is possible only when working with a control template. In other words, you can't write a style trigger that uses the TargetName property to access the ButtonChrome object, because the name *Chrome* isn't in scope in your style. This is just one of the ways that templates give you more power than styles alone.

Triggers don't always need to use the TargetName property. For example, the trigger for the IsEnabled property simply adjusts the foreground color of any text content in the button. This trigger does its work by setting the attached TextElement.Foreground property without the help of the ButtonChrome class:

```
<Trigger Property="UIElement.IsEnabled">
  <Setter Property="TextElement.Foreground">
    <Setter.Value>
      <SolidColorBrush>#FFADADAD</SolidColorBrush>
    </Setter.Value>
  </Setter>
  <Trigger.Value>
    <s:Boolean>False</s:Boolean>
  </Trigger.Value>
</Trigger>
```

You'll see the same division of responsibilities when you build your own control templates. If you're lucky enough to be able to do all your work directly with triggers, you may not need to create custom classes and add code. On the other hand, if you need to provide more-complex visual tailoring, you may need to derive a custom chrome class of your own. The ButtonChrome class itself provides no customization—it's dedicated to rendering the standard theme-specific appearance of a button.

■ **Note** All the XAML that you see in this section is extracted from the standard Button control template. A bit later, in the "Dissecting Controls" section, you'll learn how to view a control's default control template.

TYPES OF TEMPLATES

This chapter focuses on control templates, which allow you to define the elements that make up a control. However, there are actually three types of templates in the WPF world, all of which derive from the base FrameworkTemplate class. Along with control templates (represented by the ControlTemplate class), there are data templates (represented by DataTemplate and HierarchicalDataTemplate) and the more specialized panel template for an ItemsControl (ItemsPanelTemplate).

Data templates are used to extract data from an object and display it in a content control or in the individual items of a list control. Data templates are ridiculously useful in data-binding scenarios, and they're described in detail in Chapter 20. To a certain extent, data templates and control templates overlap. For example, both types of templates allow you to insert additional elements, apply formatting, and so on. However, data templates are used to add elements *inside* an existing control. The prebuilt aspects of that control aren't changed. On the other hand, control templates are a much more drastic approach that allows you to completely rewrite the content model of a control.

Finally, panel templates are used to control the layout of items in a list control (a control that derives from the ItemsControl class). For example, you can use them to create a list box that tiles its items from right to left and then down (rather than the standard top-to-bottom single-line display). Panel templates are described in Chapter 20.

You can certainly combine template types in the same control. For example, if you want to create a slick list control that is bound to a specific type of data, lays its items out in a nonstandard way, and replaces the stock border with something more exciting, you'll want to create your own data templates, panel template, and control template.

The Chrome Classes

The ButtonChrome class is defined in the Microsoft.Windows.Themes namespace, which holds a relatively small set of similar classes that render basic Windows details. Along with ButtonChrome, these classes include BulletChrome (for check boxes and radio buttons), ScrollChrome (for scrollbars), ListBoxChrome, and SystemDropShadowChrome. This is the lowest level of the public control API. At a slightly higher level, you'll find that the System.Windows.Controls.Primitives namespace includes a number of basic elements that you can use independently but are more commonly wrapped into more-useful controls. These include ScrollBar, ResizeGrip (for sizing a window), Thumb (the draggable button on a scrollbar), TickBar (the optional set of ticks on a slider), and so on. Essentially, System.Windows.Controls.Primitives provides bare-bones ingredients that can be used in a variety of controls and aren't very useful on their own, while Microsoft.Windows.Themes contains the down-and-dirty drawing logic for rendering these details.

There's one more difference. The types in System.Windows.Controls.Primitives are, like most WPF types, defined in the PresentationFramework.dll assembly. However, those in the Microsoft.Windows.Themes are defined separately in *three* assemblies: PresentationFramework.Aero.dll, PresentationFramework.Luna.dll, and PresentationFramework.Royale.dll. Each assembly includes its own version of the ButtonChrome class (and other chrome classes), with slightly different rendering logic. The one that WPF uses depends on your operating system and theme settings.

■ **Note** You'll learn more about the internal workings of a chrome class in Chapter 18, and you'll learn to build your own chrome class with custom rendering logic.

Although control templates often draw on the chrome classes, they don't always need to do so. For example, the ResizeGrip element (which is to create the grid of dots in the bottom-right corner of a resizable window) is simple enough that its template can use the drawing classes you learned about in Chapter 12 and Chapter 13, such as Path, DrawingBrush, and LinearGradientBrush. Here's the (somewhat convoluted) markup that it uses:

```
<ControlTemplate TargetType="{x:Type ResizeGrip}" ... >
  <Grid Background="{TemplateBinding Panel.Background}" SnapsToDevicePixels="True">
    <Path Margin="0,0,2,2" Data="M9,0L11,0 11,11 0,11 0,9 3,9 3,6 6,6 6,3 9,3z"
     HorizontalAlignment="Right" VerticalAlignment="Bottom">
      <Path.Fill>
        <DrawingBrush ViewboxUnits="Absolute" TileMode="Tile" Viewbox="0,0,3,3"
         Viewport="0,0,3,3" ViewportUnits="Absolute">
          <DrawingBrush.Drawing>
            <DrawingGroup>
              <DrawingGroup.Children>
                <GeometryDrawing Geometry="M0,0L2,0 2,2 0,2z">
                  <GeometryDrawing.Brush>
                    <LinearGradientBrush EndPoint="1,0.75" StartPoint="0,0.25">
                      <LinearGradientBrush.GradientStops>
                        <GradientStop Offset="0.3" Color="#FFFFFFFF" />
                        <GradientStop Offset="0.75" Color="#FFBBC5D7" />
                        <GradientStop Offset="1" Color="#FF6D83A9" />
                      </LinearGradientBrush.GradientStops>
                    </LinearGradientBrush>
                  </GeometryDrawing.Brush>
                </GeometryDrawing>
              </DrawingGroup.Children>
            </DrawingGroup>
          </DrawingBrush.Drawing>
        </DrawingBrush>
      </Path.Fill>
    </Path>
  </Grid>
</ControlTemplate>
```

■ **Note** It's common to see the SnapsToDevicePixels setting in a prebuilt control template (and it's useful in the one you create as well). As you learned in Chapter 12, SnapsToDevicePixels ensures that single-pixel lines aren't placed "between" pixels because of WPF's resolution independence, which creates a fuzzy 2-pixel line.

Dissecting Controls

When you create a control template (as you'll see in the next section), your template replaces the existing template completely. This gives you a high level of flexibility, but it also makes life a little more complex. In most cases, you'll need to see the standard template that a control uses before you can create your own adapted version. In some cases, your control template might mirror the standard template with only a minor change.

The WPF documentation doesn't list the XAML for standard control templates. However, you can get the information you need programmatically. The basic idea is to grab a control's template from its Template property (which is defined as part of the Control class) and then serialize it to XAML by using the XamlWriter class. Figure 17-5 shows an example with a program that lists all the WPF controls and lets you view each one's control template.

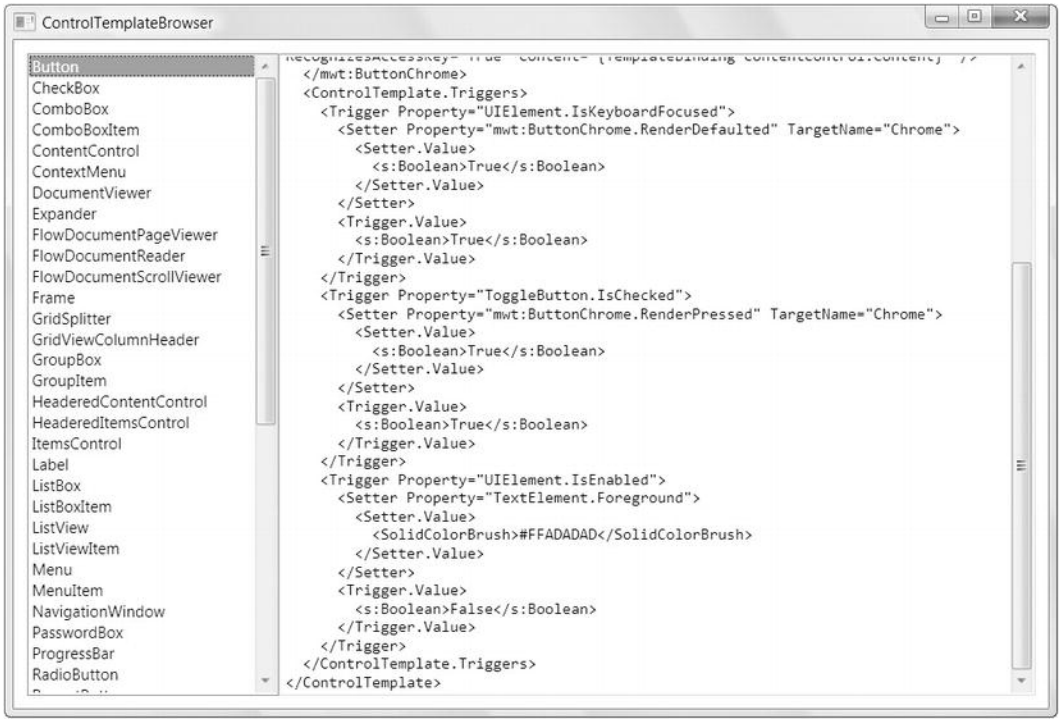

Figure 17-5. *Browsing WPF control templates*

The secret to building this application is a healthy dose of *reflection*, the .NET API for examining types. When the main window in this application is first loaded, it scans all the types in the core PresentationFramework.dll assembly (which is where the Control class is defined). It then adds these types to a collection, which it sorts by type name, and then binds that collection to a list.

```
private void Window_Loaded(object sender, EventArgs e)
{
    Type controlType = typeof(Control);
    List<Type> derivedTypes = new List<Type>();

    // Search all the types in the assembly where the Control class is defined.
    Assembly assembly = Assembly.GetAssembly(typeof(Control));
    foreach (Type type in assembly.GetTypes())
    {
        // Only add a type of the list if it's a Control, a concrete class,
        // and public.
        if (type.IsSubclassOf(controlType) && !type.IsAbstract && type.IsPublic)
```

```
            {
                derivedTypes.Add(type);
            }
        }

        // Sort the types. The custom TypeComparer class orders types
        // alphabetically by type name.
        derivedTypes.Sort(new TypeComparer());

        // Show the list of types.
        lstTypes.ItemsSource = derivedTypes;
    }
```

Whenever a control is selected from the list, the corresponding control template is shown in the text box on the right. This step takes a bit more work. The first challenge is that a control template is null until the control is displayed in a window. Using reflection, the code attempts to create an instance of the control and add it to the current window (albeit with a Visibility of Collapse so it can't be seen). The second challenge is that you need to convert the live ControlTemplate object to the familiar XAML markup. The static XamlWriter.Save() method takes care of this task, although the code uses the XmlWriter and XmlWriterSettings objects to make sure the XAML is indented so that it's easier to read. All of this code is wrapped in an exception-handling block, which catches the problems that result from controls that can't be created or can't be added to a Grid (such as another Window or a Page):

```
private void lstTypes_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    try
    {
        // Get the selected type.
        Type type = (Type)lstTypes.SelectedItem;

        // Instantiate the type.
        ConstructorInfo info = type.GetConstructor(System.Type.EmptyTypes);
        Control control = (Control)info.Invoke(null);

        // Add it to the grid (but keep it hidden).
        control.Visibility = Visibility.Collapsed;
        grid.Children.Add(control);

        // Get the template.
        ControlTemplate template = control.Template;

        // Get the XAML for the template.
        XmlWriterSettings settings = new XmlWriterSettings();
        settings.Indent = true;
        StringBuilder sb = new StringBuilder();
        XmlWriter writer = XmlWriter.Create(sb, settings);
        XamlWriter.Save(template, writer);

        // Display the template.
        txtTemplate.Text = sb.ToString();
```

```
        // Remove the control from the grid.
        grid.Children.Remove(control);
    }
    catch (Exception err)
    {
        txtTemplate.Text = "<< Error generating template: " + err.Message + ">>";
    }
}
```

It wouldn't be much more difficult to extend this application so you can edit the template in the text box, convert it back to a ControlTemplate object (using the XamlReader), and then assign that to a control to see its effect. However, you'll have an easier time testing and refining templates by putting them into action in a real window, as described in the next section.

■ **Tip** If you're using Expression Blend, you can also use a handy feature that lets you edit the template for any control that you're working with. (Technically, this step grabs the default template, creates a copy of it for your control, and then lets you edit the copy.) To try this, right-click a control on the design surface and choose Edit Control Parts (Template) ä Edit a Copy. Your control template copy will be stored as a resource (see Chapter 10), so you'll be prompted to choose a descriptive resource key, and you'll need to choose between storing your resource in the current window or in the global application resources so you can use your control template throughout your application.

Creating Control Templates

So far, you've learned a fair bit about the way templates work, but you haven't built a template of your own. In the following sections, you'll build a simple custom button and learn a few of the finer details about control templates in the process.

As you've already seen, the basic Button control uses the ButtonChrome class to draw its distinctive background and border. One of the reasons that the Button class uses ButtonChrome instead of the WPF drawing primitives is that a standard button's appearance depends on a few obvious characteristics (whether it's disabled, focused, or in the process of being clicked) and other subtler factors (such as the current Windows theme). Implementing this sort of logic with triggers alone would be awkward.

However, when you build your own custom controls, you're probably not as worried about standardization and theme integration. (In fact, WPF doesn't emphasize user interface standardization nearly as strongly as previous user interface technologies.) Instead, you're more concerned with creating attractive, distinctive controls that blend in with the rest of your user interface. For that reason, you might not need to create classes such as ButtonChrome. Instead, you can use the elements you already know (along with the drawing elements you learned about in Chapter 12 and Chapter 13, and the animation skills you picked in Chapter 15 and Chapter 16) to design a self-sufficient control template with no code.

■ **Note** For an alternate approach, check out Chapter 18, which explains how to build your own chrome with custom rendering logic and integrate it into a control template.

A Simple Button

To apply a custom control template, you simply set the Template property of your control. Although you can define an inline template (by nesting the control template tag inside the control tag), this approach rarely makes sense. That's because you'll almost always want to reuse your template to skin multiple instances of the same control. To accommodate this design, you need to define your control template as a resource and refer to it by using a StaticResource reference, as shown here:

```
<Button Margin="10" Padding="5" Template="{StaticResource ButtonTemplate}">
 A Simple Button with a Custom Template</Button>
```

Not only does this approach make it easier to create a whole host of customized buttons, it also gives you the flexibility to modify your control template later without disrupting the rest of your application's user interface.

In this particular example, the ButtonTemplate resource is placed in the Resources collection of the containing window. However, in a real application, you're much more likely to use application resources. The reasons (and a few design tips) are discussed a bit later in the "Organizing Template Resources" section.

Here's the basic outline for the control template:

```
<Window.Resources>
  <ControlTemplate x:Key="ButtonTemplate" TargetType="{x:Type Button}">
    ...
  </ControlTemplate>
</Window.Resources>
```

You'll notice that this control template sets the TargetType property to explicitly indicate it's designed for buttons. As a matter of style, this is always a good convention to follow. In content controls, such as the button, it's also a requirement, or the ContentPresenter won't work.

To create a template for a basic button, you need to draw your own border and background and then place the content inside the button. Two possible candidates for drawing the border are the Rectangle class and the Border class. The following example uses the Border class to combine a rounded orange outline with an eye-catching red background and white text:

```
<ControlTemplate x:Key="ButtonTemplate" TargetType="{x:Type Button}">
  <Border BorderBrush="Orange" BorderThickness="3" CornerRadius="2"
   Background="Red" TextBlock.Foreground="White">
    ...
  </Border>
</ControlTemplate>
```

This takes care of the background, but you still need a way to display the button content. You may remember from your earlier exploration that the Button class includes a ContentPresenter in its control template. The ContentPresenter is required for all content controls—it's the "insert content here" marker that tells WPF where to stuff the content:

```
<ControlTemplate x:Key="ButtonTemplate" TargetType="{x:Type Button}">
  <Border BorderBrush="Orange" BorderThickness="3" CornerRadius="2"
   Background="Red" TextBlock.Foreground="White">
    <ContentPresenter RecognizesAccessKey="True"></ContentPresenter>
  </Border>
</ControlTemplate>
```

This ContentPresenter sets the RecognizesAccessKey property to true. Although this isn't required, it ensures that the button supports *access keys*—underlined letters that you can use to quickly trigger the button. In this case, if your button has text such as *Click _Me*, the user can trigger the button by pressing Alt+M. (Under standard Windows settings, the underscore is hidden, and the access key—in this case, *M*—appears underlined as soon as you press the Alt key.) If you don't set RecognizesAccessKey to true, this detail will be ignored, and any underscores will be treated as ordinary underscores and displayed as part of the button content.

■ **Note** If a control derives from ContentControl, its template will include a ContentPresenter that specifies where the content will be placed. If the control derives from ItemsControl, its template will include an ItemsPresenter that indicates where the panel that contains the list of items will be placed. In rare cases, the control may use a derived version of one of these classes—for example, the ScrollViewer's control template uses a ScrollContentPresenter, which derives from ContentPresenter.

Template Bindings

There's still one minor issue with this example. Right now the tag you've added for your button specifies a Margin value of 10 and a Padding of 5. The StackPanel pays attention to the Margin property of the button, but the Padding property is ignored, leaving the contents of your button scrunched up against the sides. The problem here is that the Padding property doesn't have any effect unless you specifically heed it in your template. In other words, it's up to your template to retrieve the padding value and use it to insert some extra space around your content.

Fortunately, WPF has a tool that's designed exactly for this purpose: *template bindings*. By using a template binding, your template can pull out a value from the control to which you're applying the template. In this example, you can use a template binding to retrieve the value of the Padding property and use it to create a margin around the ContentPresenter:

```
<ControlTemplate x:Key="ButtonTemplate" TargetType="{x:Type Button}">
  <Border BorderBrush="Orange" BorderThickness="3" CornerRadius="2"
   Background="Red" TextBlock.Foreground="White">
    <ContentPresenterRecognizesAccessKey="True"
     Margin="{TemplateBinding Padding}"></ContentPresenter>
  </Border>
</ControlTemplate>
```

This achieves the desired effect of adding some space between the border and the content. Figure 17-6 shows your modest new button.

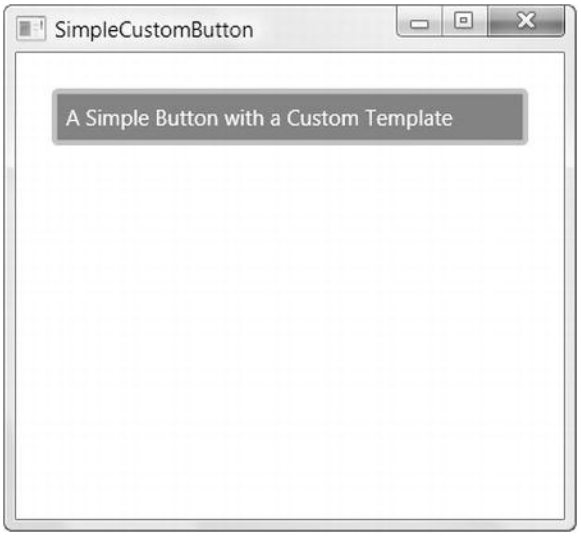

Figure 17-6. *A button with a customized control template*

Template bindings are similar to ordinary data bindings, but they're lighter weight because they're specifically designed for use in a control template. They support only one-way data binding (in other words, they can pass information from the control to the template but not the other way around), and they can't be used to draw information from a property of a class that derives from Freezable. If you run into a situation where template bindings won't work, you can use a full-fledged data binding instead. Chapter 18 includes a sample color picker that runs into this problem and uses a combination of template bindings and regular bindings.

■ **Note** Template bindings support the WPF change-monitoring infrastructure that's built into all dependency properties. That means that if you modify a property in a control, the template takes it into account automatically. This detail is particularly useful when you're using animations that change a property value repeatedly in a short space of time.

The only way you can anticipate what template bindings are needed is to check the default control template. If you look at the control template for the Button class, you'll find that it uses a template binding in exactly the same way as this custom template—it takes the padding specified on the button and converts it to a margin around the ContentPresenter. You'll also find that the standard button template includes a few more template bindings that aren't used in the simple customized template, such as HorizontalAlignment, VerticalAlignment, and Background. That means if you set these properties on the button, they'll have no effect on the simple custom template.

■ **Note** Technically, the ContentPresenter works because it has a template binding that sets the ContentPresenter.Content property to the Button.Content property. However, this binding is implicit, so you don't need to add it yourself.

In many cases, leaving out template bindings isn't a problem. In fact, you don't need to bind a property if you don't plan to use it or don't want it to change your template. For example, it makes sense that the current simple button sets the Foreground property for text to white and ignores any value you've set for the Background property because the foreground and background are intrinsic parts of this button's visual appearance.

There's another reason you might choose to avoid template bindings—your control may not be able to support them adequately. For example, if you've ever set the Background property of a button, you've probably noticed that this background isn't handled consistently when the button is pressed (in fact, it disappears at this point and is replaced with the default visual for pressed buttons). The custom template shown in this example is similar. Although it doesn't yet have any mouseover and mouse-pressed behavior, after you add these details you'll want to take complete control over the colors and how they change in different states.

Triggers That Change Properties

If you try the button that you created in the previous section, you'll find it's a major disappointment. Essentially, it's nothing more than a rounded red rectangle—as you move the mouse over it or click it, there's no visual feedback. The button simply lies there inert.

This problem is easily fixed by adding triggers to your control template. You first considered triggers with styles in Chapter 11. As you know, you can use triggers to change one or more properties when another property changes. The bare minimums that you'll want to respond to in your button are IsMouseOver and IsPressed. Here's a revised version of the control template that changes the colors when these properties change:

```
<ControlTemplate x:Key="ButtonTemplate" TargetType="{x:Type Button}">
  <Border Name="Border" BorderBrush="Orange" BorderThickness="3" CornerRadius="2"
   Background="Red" TextBlock.Foreground="White">
    <ContentPresenter RecognizesAccessKey="True"
     Margin="{TemplateBinding Padding}"></ContentPresenter>
  </Border>
  <ControlTemplate.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter TargetName="Border" Property="Background" Value="DarkRed" />
    </Trigger>
    <Trigger Property="IsPressed" Value="True">
      <Setter TargetName="Border" Property="Background" Value="IndianRed" />
      <Setter TargetName="Border" Property="BorderBrush" Value="DarkKhaki" />
    </Trigger>
  </ControlTemplate.Triggers>
</ControlTemplate>
```

There's one other change that makes this template work. The Border element has been given a name, and that name is used to set the TargetName property of each Setter. This way, the Setter can update the Background and BorderBrush properties of the Border that's specified in the template. Using names is the easiest way to make sure a single specific part of a template is updated. You could create an element-typed rule that affects all Border elements (because you know there is only a single border in the button template), but this approach is both clearer and more flexible if you change the template later.

There's one more required element in any button (and most other controls)—a focus indicator. There's no way to change the existing border to add a focus effect, but you can easily add another element that shows it and simply show or hide this element based on the Button.IsKeyboardFocused property by using a trigger. Although you could create a focus effect in many ways, the following example simply adds a

transparent Rectangle element with a dashed border. The Rectangle doesn't have the ability to hold child content, so you need to make sure the Rectangle overlaps the rest of the content. The easiest way to do this is to wrap the Rectangle and the ContentPresenter in a one-cell Grid, with both elements in the same cell.

Here's the revised template with focus support:

```
<ControlTemplate x:Key="ButtonTemplate" TargetType="{x:Type Button}">
  <Border Name="Border" BorderBrush="Orange" BorderThickness="3" CornerRadius="2"
   Background="Red" TextBlock.Foreground="White">
    <Grid>
      <Rectangle Name="FocusCue" Visibility="Hidden" Stroke="Black"
       StrokeThickness="1" StrokeDashArray="1 2"
       SnapsToDevicePixels="True"></Rectangle>
      <ContentPresenter RecognizesAccessKey="True"
       Margin="{TemplateBinding Padding}"></ContentPresenter>
    </Grid>
  </Border>
  <ControlTemplate.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter TargetName="Border" Property="Background" Value="DarkRed" />
    </Trigger>
    <Trigger Property="IsPressed" Value="True">
      <Setter TargetName="Border" Property="Background" Value="IndianRed" />
      <Setter TargetName="Border" Property="BorderBrush" Value="DarkKhaki" />
    </Trigger>
    <Trigger Property="IsKeyboardFocused" Value="True">
      <Setter TargetName="FocusCue" Property="Visibility" Value="Visible" />
    </Trigger>
  </ControlTemplate.Triggers>
</ControlTemplate>
```

Once again, the Setter finds the element it needs to change by using the TargetName property (which points to the FocusCue rectangle in this example).

■ **Note** This technique of hiding or showing elements in response to a trigger is a useful building block in many templates. You can use it to replace the visuals of a control with something completely different when its state changes. (For example, a clicked button could change from a rectangle to an ellipse by hiding the former and showing the latter.)

Figure 17-7 shows three buttons that use the revised template. The second button currently has focus (as represented by the dashed rectangle), while the mouse is hovering over the third button.

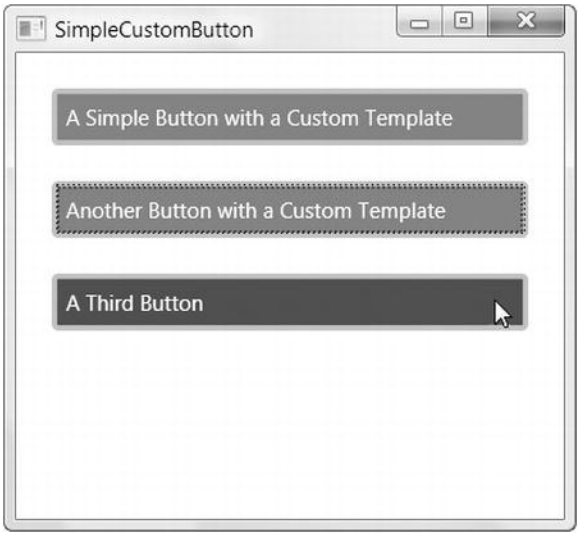

Figure 17-7. *Buttons with focus and mouseover support*

To really round out this button, you'll add an additional trigger that changes the button background (and possibly the text foreground) when the IsEnabled property of the button becomes false:

```
<Trigger Property="IsEnabled" Value="False">
  <Setter TargetName="Border" Property="TextBlock.Foreground" Value="Gray" />
  <Setter TargetName="Border" Property="Background" Value="MistyRose" />
</Trigger>
```

To make sure that this rule takes precedence over any conflicting trigger settings, you should define it at the end of the list of triggers. That way, it doesn't matter if the IsMouseOver property is also true; the IsEnabled property trigger takes precedence, and the button remains inactive.

TEMPLATES VS. STYLES

It might have occurred to you that there's a similarity between templates and styles. Both allow you to change the appearance of an element, usually throughout your application. However, styles are far more limited in scope. They're able to adjust properties of the control but not replace it with an entirely new visual tree that's made up of different elements.

Already, the simple button you've seen includes features that couldn't be duplicated with styles alone. Although you could use styles to set the background of a button, you'd have more trouble adjusting the background when the button was pressed because the built-in template for the button already includes a trigger for that purpose. You also wouldn't have an easy way to add the focus rectangle.

Control templates also open the door to many more exotic types of buttons that are unthinkable with styles. For example, rather than using a rectangular border, you can create a button that's shaped like an ellipse or uses a path to draw a more complex shape. All you need are the drawing classes from Chapter 12. The rest of your markup—even the triggers that switch the background from one state to another—require relatively few changes.

Triggers That Use Animation

As you learned in Chapter 11, triggers aren't limited to setting properties. You can also use event triggers to run animations when specific properties change.

At first glance, this may seem like a frill, but it's actually a key ingredient in all but the simplest WPF controls. For example, consider the button you've studied so far. Currently, it switches instantaneously from one color to another when the mouse moves over the top. However, a more modern button might use a very brief animation to *blend* from one color to the other, which creates a subtle but refined effect. Similarly, the button might use an animation to change the opacity of the focus cue rectangle, fading it quickly into view when the button gains focus rather than showing it in one step. In other words, event triggers allow controls to change from one state to another more gradually and more gracefully, which gives them that extra bit of polish.

Here's a revamped button template that uses triggers to make the button color pulse (shift continuously between red and blue) when the mouse is over it. When the mouse moves away, the button background returns to its normal color by using a separate 1-second animation:

```
<ControlTemplate x:Key="ButtonTemplate" TargetType="{x:Type Button}">
  <Border BorderBrush="Orange" BorderThickness="3" CornerRadius="2"
   Background="Red" TextBlock.Foreground="White" Name="Border">
    <Grid>
      <Rectangle Name="FocusCue" Visibility="Hidden" Stroke="Black"
       StrokeThickness="1" StrokeDashArray="1 2"
       SnapsToDevicePixels="True" ></Rectangle>
      <ContentPresenter RecognizesAccessKey="True"
         Margin="{TemplateBinding Padding}"></ContentPresenter>
    </Grid>
  </Border>

  <ControlTemplate.Triggers>
    <EventTrigger RoutedEvent="MouseEnter">
      <BeginStoryboard>
        <Storyboard>
          <ColorAnimation Storyboard.TargetName="Border"
           Storyboard.TargetProperty="Background.Color"
           To="Blue" Duration="0:0:1" AutoReverse="True"
           RepeatBehavior="Forever"></ColorAnimation>
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
    <EventTrigger RoutedEvent="MouseLeave">
      <BeginStoryboard>
        <Storyboard>
          <ColorAnimation Storyboard.TargetName="Border"
            Storyboard.TargetProperty="Background.Color"
            Duration="0:0:0.5"></ColorAnimation>
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </ControlTemplate.Triggers>
</ControlTemplate>
```

You can add the mouseover animation in two equivalent ways—by creating an event trigger that responds to the MouseEnter and MouseLeave events (as demonstrated here) or by creating a property trigger that adds enter and exit actions when the IsMouseOver property changes.

This example uses two ColorAnimation objects to change the button. Here are some other tasks you might want to perform with an EventTrigger-driven animation:

Show or hide an element: To do this, you need to change the Opacity property of an element in the control template.

Change the shape or position: You can use a TranslateTransform to tweak the positioning of an element (for example, offsetting it slightly to give the impression that the button has been pressed). You can use a ScaleTransform or a RotateTransform to twiddle the element's appearance slightly as the user moves the mouse over it.

Change the lighting or coloration: To do this, you need an animation that acts on the brush that you use to paint the background. You can use a ColorAnimation to change colors in a SolidBrush, but more-advanced effects are possible by animating more-complex brushes. For example, you can change one of the colors in a LinearGradientBrush (which is what the default button control template does), or you can shift the center point of a RadialGradientBrush.

■ **Tip** Some advanced lighting effects use multiple layers of transparent elements. In this case, your animation modifies the opacity of one layer to let other layers show through.

Organizing Template Resources

When using control templates, you need to decide how broadly you want to share your templates and whether you want to apply them automatically or explicitly.

The first question asks you to think about where you want to use your templates. For example, are they limited to a specific window? In most situations, control templates apply to multiple windows and possibly even the entire application. To avoid defining them more than once, you can define them in the Resources collection of the Application class.

However, this raises another consideration. Often control templates are shared between applications. It's quite possible that a single application might use templates that have been developed separately. However, an application can have only a single App.xaml file and a single Application.Resources collection. For that reason, it's a better idea to define your resources in separate resource dictionaries. That gives you the flexibility to bring them into action in specific windows or in the entire application. It also allows you to combine styles because any application can hold multiple resource dictionaries. To add a resource dictionary in Visual Studio, right-click your project in the Solution Explorer window, choose Add ä New Item, and then select Resource Dictionary (WPF).

You've already learned about resource dictionaries in Chapter 10. Using them is easy. You simply need to add a new XAML file to your application with content like this:

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" >
```

```
  <ControlTemplate x:Key="ButtonTemplate" TargetType="{x:Type Button}">
    ...
  </ControlTemplate>
</ResourceDictionary>
```

Although you could combine all your templates into a single resource dictionary file, experienced developers prefer to create a separate resource dictionary for each control template. That's because a control template can quickly become quite complex and can draw on a host of other related resources. Keeping these together in one place, but separate from other controls, is good organization.

To use your resource dictionary, you simply add it to the Resources collection of a specific window or, more commonly, your application. You do this by using the MergedDictionaries collection. For example, if your button template is in a file named Button.xaml in a project subfolder named Resources, you could use this markup in your App.xaml file:

```
<Application x:Class="SimpleApplication.App"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 StartupUri="Window1.xaml">
  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Resources\Button.xaml" />
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

Refactoring the Button Control Template

As you enhance and extend a control template, you may find that it wraps various details, including specialized shapes, geometries, and brushes. It's a good idea to pull these details out of your control template and define them as separate resources. One reason you'll take this step is to make it easier to reuse these brushes among a set of related controls. For example, you might decide that you want to create a customized Button, CheckBox, and RadioButton that use a similar set of colors. To make this easier, you could create a separate resource dictionary for your brushes (named Brushes.xaml) and merge that into the resource dictionary for each of your controls (such as Button.xaml, CheckBox.xaml, and RadioButton.xaml).

To see this technique in action, consider the following markup. It presents the complete resource dictionary for a button, including the resources that the control template uses, the control template, and the style rule that applies the control template to every button in the application. This is the order that you always need to follow because a resource needs to be defined before it can be used. (If you defined one of the brushes after the template, you'd receive an error because the template wouldn't be able to find the brush it requires.)

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

   <!-- Resources used by the template. -->
   <RadialGradientBrush RadiusX="1" RadiusY="5" GradientOrigin="0.5,0.3"
    x:Key="HighlightBackground">
```

```
  <GradientStop Color="White" Offset="0" />
  <GradientStop Color="Blue" Offset=".4" />
  </RadialGradientBrush>

<RadialGradientBrush RadiusX="1" RadiusY="5" GradientOrigin="0.5,0.3"
 x:Key="PressedBackground">
  <GradientStop Color="White" Offset="0" />
  <GradientStop Color="Blue" Offset="1" />
</RadialGradientBrush>

<SolidColorBrush Color="Blue" x:Key="DefaultBackground"></SolidColorBrush>
<SolidColorBrush Color="Gray" x:Key="DisabledBackground"></SolidColorBrush>

<RadialGradientBrush RadiusX="1" RadiusY="5" GradientOrigin="0.5,0.3"
 x:Key="Border">
  <GradientStop Color="White" Offset="0" />
  <GradientStop Color="Blue" Offset="1" />
</RadialGradientBrush>

<!-- The button control template. -->
<ControlTemplate x:Key="GradientButtonTemplate" TargetType="{x:Type Button}">
  <Border Name="Border" BorderBrush="{StaticResource Border}" BorderThickness="2"
   CornerRadius="2" Background="{StaticResource DefaultBackground}"
   TextBlock.Foreground="White">
    <Grid>
      <Rectangle Name="FocusCue" Visibility="Hidden" Stroke="Black"
       StrokeThickness="1" StrokeDashArray="1 2" SnapsToDevicePixels="True">
      </Rectangle>
      <ContentPresenter Margin="{TemplateBinding Padding}"
       RecognizesAccessKey="True"></ContentPresenter>
    </Grid>
  </Border>
  <ControlTemplate.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter TargetName="Border" Property="Background"
       Value="{StaticResource HighlightBackground}" />
    </Trigger>
    <Trigger Property="IsPressed" Value="True">
      <Setter TargetName="Border" Property="Background"
       Value="{StaticResource PressedBackground}" />
    </Trigger>
    <Trigger Property="IsKeyboardFocused" Value="True">
      <Setter TargetName="FocusCue" Property="Visibility"
        Value="Visible"></Setter>
    </Trigger>
    <Trigger Property="IsEnabled" Value="False">
      <Setter TargetName="Border" Property="Background"
       Value="{StaticResource DisabledBackground}"></Setter>
    </Trigger>
  </ControlTemplate.Triggers>
</ControlTemplate>
```

```
</ResourceDictionary>
```

Figure 17-8 shows the button that this template defines. In this example, a gradient fill is used when the user moves the mouse over the button. However, the gradient is always centered in the middle of the button. If you want to create a more exotic effect, such as a gradient that follows the position of the mouse, you'll need to use an animation or write code. Chapter 18 shows an example with a custom chrome class that implements this effect.

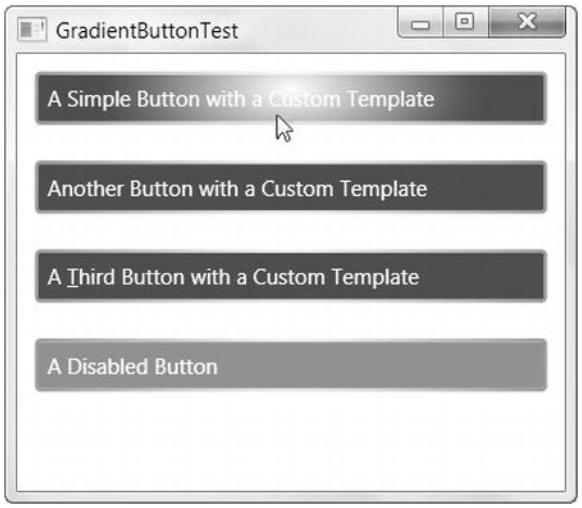

Figure 17-8. *A gradient button*

Applying Templates with Styles

There's one limitation in this design. The control template essentially hard-codes quite a few details, such as the color scheme. That means that if you want to use the same combination of elements in your button (Border, Grid, Rectangle, and ContentPresenter) and arrange them in the same way but you want to supply a different color scheme, you'll be forced to create a new copy of the template that references different brush resources.

This isn't necessarily a problem (after all, the layout and formatting details may be so closely related that you don't want to separate them anyway). However, it does limit your ability to reuse your control template. If your template uses a complex arrangement of elements that you know you'll want to reuse with a variety of formatting details (usually colors and fonts), you can pull these details out of your template and put them into a style.

To make this work, you'll need to rework your template. Instead of using hard-coded colors, you need to pull the information out of control properties by using template bindings. The following example defines a streamlined template for the fancy button you saw earlier. The control template treats some details as fundamental, unchanging ingredients—namely, the focus box and the rounded 2-unit-thick border. The background and border brushes are configurable. The only trigger that remains is the one that shows the focus box:

```
<ControlTemplate x:Key="CustomButtonTemplate" TargetType="{x:Type Button}">
  <Border Name="Border" BorderThickness="2" CornerRadius="2"
   Background="{TemplateBinding Background}"
   BorderBrush="{TemplateBinding BorderBrush}">
```

```
    <Grid>
      <Rectangle Name="FocusCue" Visibility="Hidden" Stroke="Black"
       StrokeThickness="1" StrokeDashArray="1 2" SnapsToDevicePixels="True">
      </Rectangle>
      <ContentPresenter Margin="{TemplateBinding Padding}"
       RecognizesAccessKey="True"></ContentPresenter>
    </Grid>
  </Border>
  <ControlTemplate.Triggers>
    <Trigger Property="IsKeyboardFocused" Value="True">
      <Setter TargetName="FocusCue" Property="Visibility"
        Value="Visible"></Setter>
    </Trigger>
  </ControlTemplate.Triggers>
</ControlTemplate>
```

The associated style applies this control template, sets the border and background colors, and adds triggers that change the background depending on the state of the button:

```
<Style x:Key="CustomButtonStyle" TargetType="{x:Type Button}">
  <Setter Property="Control.Template"
   Value="{StaticResource CustomButtonTemplate}"></Setter>
  <Setter Property="BorderBrush"
   Value="{StaticResource Border}"></Setter>
  <Setter Property="Background"
     Value="{StaticResource DefaultBackground}"></Setter>
      <Setter Property="TextBlock.Foreground"
       Value="White"></Setter>
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="Background"
       Value="{StaticResource HighlightBackground}" />
    </Trigger>
    <Trigger Property="IsPressed" Value="True">
      <Setter Property="Background"
       Value="{StaticResource PressedBackground}" />
    </Trigger>
    <Trigger Property="IsEnabled" Value="False">
      <Setter Property="Background"
       Value="{StaticResource DisabledBackground}"></Setter>
    </Trigger>
  </Style.Triggers>
</Style>
```

Ideally, you'd be able to keep all the triggers in the control template because they represent control behavior and use the style simply to set basic properties. Unfortunately, that's not possible here if you want to give the style the ability to set the color scheme.

■ **Note** If you set triggers in both the control template and style, the style triggers win out.

To use this new template, you need to set the Style property of a button rather than the Template property:

```
<Button Margin="10" Padding="5" Style="{StaticResource CustomButtonStyle}">
 A Simple Button with a Custom Template</Button>
```

You can now create new styles that use the same template but bind to different brushes to apply a new color scheme.

There's one significant limitation in this approach. You can't use the Setter.TargetName property in this style because the style doesn't contain the control template (it simply references it). As a result, your style and its triggers are somewhat limited. They can't reach deep into the visual tree to change the aspect of a nested element. Instead, your style needs to set a property of the control, and the element in the control needs to bind the property by using a template binding.

CONTROL TEMPLATES VS. CUSTOM CONTROLS

You can get around both of the problems discussed here—being forced to define control behavior in the style with triggers and not being able to target specific elements—by creating a custom control. For example, you could build a class that derives from Button and adds properties such as HighlightBackground, DisabledBackground, and PressedBackground. You could then bind to these properties in the control template and simply set them in the style with no triggers required. However, this approach has its own drawback. It forces you to use a different control in your user interface (such as CustomButton instead of just Button). This is more trouble when designing the application.

Usually, you'll switch from custom control templates to custom controls in one of two situations:

If you decide to create a custom control, Chapter 18 has all the information you need.

Applying Templates Automatically

In the current example, each button is responsible for hooking itself up to the appropriate template by using the Template or Style property. This makes sense if you're using your control template to create a specific effect in a specific place in your application. It's less convenient if you want to re-skin every button in your entire application with a custom look. In this situation, it's more likely that you want all the buttons in your application to acquire your new template automatically. To make this a reality, you need to apply your control template with a style.

The trick is to use a typed style that affects the appropriate element type automatically and sets the Template property. Here's an example of the style you'd place in the resources collection of your resource dictionary to give your buttons a new look:

```
<Style TargetType="{x:Type Button}">
  <Setter Property="Control.Template" Value="{StaticResource ButtonTemplate}"
</Style>
```

This works because the style doesn't specify a key name, which means the element type (Button) is used instead.

Remember, you can still opt out of this style by creating a button that explicitly sets its Style to a null value:

```
<Button Style="{x:Null}" ... ></Button>
```

■ **Tip** This technique works even better if you've followed good design practices and defined your button in a separate resource dictionary. In this situation, the style doesn't sprint into action until you add a ResourceDictionary tag that imports your resources into the entire application or a specific window, as described earlier.

A resource dictionary that contains a combination of type-based styles is often called (informally) a *theme*. The possibilities of themes are remarkable. They allow you to take an existing WPF application and completely re-skin all its controls without changing the user interface markup at all. All you need to do is add the resource dictionaries to your project and merge them into the Application.Resources collection in the App.xaml file.

If you hunt around the Web, you'll find more than a few themes that you can use to revamp a WPF application. For example, you can download several sample themes as part of the WPF Futures release at `http://tinyurl.com/ylojdry`.

To use a theme, add the .xaml file that contains the resource dictionary to your project. For example, WPF Futures includes a theme file named ExpressionDark.xaml. Then you need to make the styles active in your application. You could do this on a window-by-window basis, but it's quicker to import them at the application level by adding markup like this:

```
<Application ... >
  <Application.Resources>
    <ResourceDictionary Source="ExpressionDark.xaml"/>
  </Application.Resources>
</Application>
```

Now the type-based styles in the resource dictionary will be in full force and will automatically change the appearance of every common control in every window of your application. If you're an application developer in search of a hot new user interface but you don't have the design skills to build it yourself, this trick makes it easy to plug in third-party pizzazz with almost no effort.

Working with User-Selected Skins

In some applications, you might want to alter templates dynamically, usually in response to user preferences. This is easy enough to accomplish, but it's not well-documented. The basic technique is to load a new resource dictionary at runtime and use it to replace the current resource dictionary. (It's not necessary to replace all your resources, just those that are used for your skin.)

The trick is retrieving the ResourceDictionary object, which is compiled and embedded as a resource in your application. The easiest approach is to use the ResourceManager class described in Chapter 10 to load up the resources you want.

For example, imagine you've created two resources that define alternate versions of the same button control template. One is stored in a file named GradientButton.xaml, while the other is in a file named GradientButtonVariant.xaml. Both files are placed in the Resources subfolder in the current project for better organization.

Now you can create a simple window that uses one of these resources, using a Resources collection like this:

```
<Window.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary
       Source="Resources/GradientButton.xaml"></ResourceDictionary>
```

```
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Window.Resources>
```

Now you can swap in a different resource dictionary by using code like this:

```
ResourceDictionary newDictionary = new ResourceDictionary();
newDictionary.Source = new Uri(
  "Resources/GradientButtonVariant.xaml", UriKind.Relative);
this.Resources.MergedDictionaries[0] = newDictionary;
```

This code loads the resource dictionary named GradientButtonVariant and places it into the first slot in the MergedDictionaries collection. It doesn't clear the MergedDictionaries collection (or any other window resources) because it's possible that you might be linking to other resource dictionaries that you want to continue using. It doesn't add a new entry to the MergedDictionaries collection because there could then be conflict between resources with the same name but in different collections.

If you were changing the skin for an entire application, you'd use the same approach, but you'd use the resource dictionary of the application. You could update this resource dictionary by using code like this:

```
Application.Current.Resources.MergedDictionaries[0] = newDictionary;
```

You can also load a resource dictionary that's defined in another assembly by using the pack URI syntax described in Chapter 7:

```
ResourceDictionary newDictionary = new ResourceDictionary();
newDictionary.Source = new Uri(
  "ControlTemplateLibrary;component/GradientButtonVariant.xaml",
  UriKind.Relative);
this.Resources.MergedDictionaries[0] = newDictionary;
```

When you load a new resource dictionary, all the buttons are automatically updated to use the new template. You can also include basic styles as part of your skin if you don't need to be quite as ambitious when modifying a control.

This example assumes that the GradientButton.xaml and GradientButtonVariant.xaml resources use an element-typed style to change your buttons automatically. As you know, there's another approach—you can opt in to a new template by manually setting the Template or Style property of your Button objects. If you take this approach, make sure you use a DynamicResource reference instead of a StaticResource. If you use a StaticResource, the button template won't be updated when you switch skins.

■ **Note** When using a DynamicResource reference, you're making an assumption that the resource you need will appear somewhere in the resource hierarchy. If it doesn't, the resource is simply ignored, and the buttons revert to their standard appearance without generating an error.

There's another way to load resource dictionaries programmatically. You can create a code-behind class for your resource dictionary in much the same way you create code-behind classes for windows. You can then instantiate that class directly rather than using the ResourceDictionary.Source property. This approach has the benefit of being strongly typed (there's no chance of entering an invalid URI for the Source property), and it allows you to add properties, methods, and other functionality to your resource class. For example, you'll use this ability to create a resource that has event-handling code for a custom window template in Chapter 23.

Although it's easy enough to create a code-behind class for your resource dictionary, Visual Studio doesn't do it automatically. Instead, you need to add a code file with a partial class that derives from ResourceDictionary and calls InitializeComponent in the constructor:

```
public partial class GradientButtonVariant : ResourceDictionary
{
    public GradientButtonVariant()
    {
        InitializeComponent();
    }
}
```

Here, the class name GradientButtonVariant is used, and the class is stored in a file named GradientButtonVariant.xaml.cs. The XAML file holding the resource is named GradientButtonVariant.xaml. It's not necessary to make these names consistent, but it's a good idea, and it's in keeping with the convention Visual Studio uses when you create windows and pages.

The next step is to link your class to the resource dictionary. You do that by adding the Class attribute to the root element of your resource dictionary, just as you do with a window and just as you can do with any XAML class. You then supply the fully qualified class name. In this example, the project is named ControlTemplates, which is the reason for the default namespace, so the finished tag looks like this:

```
<ResourceDictionary x:Class="ControlTemplates.GradientButtonVariant" ... >
```

You can now use this code to create your resource dictionary and apply it to a window:

```
GradientButtonVariant newDictionary = new GradientButtonVariant();
this.Resources.MergedDictionaries[0] = newDictionary;
```

If you want your GradientButtonVariant.xaml.cs file to appear nested under the GradientButtonVariant.xaml file in the Solution Explorer, you need to modify the .csproj project file in a text editor. Find the code-behind file in the <ItemGroup> section and change this:

```
<Compile Include="Resources\GradientButtonVariant.xaml.cs" />
```

to this:

```
<Compile Include="Resources\GradientButtonVariant.xaml.cs">
  <DependentUpon> Resources\GradientButtonVariant.xaml</DependentUpon>
</Compile>
```

Building More Complex Templates

There is an implicit contract between a control's template and the code that underpins it. If you're replacing a control's standard template with one of your own, you need to make sure your new template meets all the requirements of the control's implementation code.

In simple controls, this process is easy, because there are few (if any) real requirements on the template. In a complex control, the issue is subtler, because it's impossible for the visuals and the implementation to be completely separated. In this situation, the control needs to make some assumptions about its visual display, no matter how well it has been designed.

You've already seen two examples of the requirements a control can place on its control template, with placeholder elements (such as ContentPresenter and ItemsPresenter) and template bindings. In the following sections, you'll see two more: elements with specific names (starting with *PART_*) and elements that are specially designed for use in a particular control's template (such as Track in the ScrollBar control). To create a successful control template, you need to look carefully at the standard template for the control

in question, make note of how these four techniques are used, and then duplicate them in your own templates.

■ **Note** There's another way to get comfortable with the interaction between controls and control templates. You can create your own custom control. In this case, you'll have the reverse challenge—you'll need to create code that uses a template in a standardized way and that can work equally well with templates supplied by other developers. You'll tackle this challenge in Chapter 18 (which makes a great complement to the perspective you'll get in this chapter).

Nested Templates

The template for the button control can be decomposed into a few relatively simple pieces. However, many templates aren't so simple. In some cases, a control template will contain a number of elements that every custom template will require as well. And in some cases, changing the appearance of a control involves creating more than one template.

For example, imagine you're planning to revamp the familiar ListBox control. The first step to create this example is to design a template for the ListBox and (optionally) add a style that applies the template automatically. Here are both ingredients rolled into one:

```
<Style TargetType="{x:Type ListBox}">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type ListBox}">
        <Border
          Name="Border"
          Background="{StaticResource ListBoxBackgroundBrush}"
          BorderBrush="{StaticResource StandardBorderBrush}"
          BorderThickness="1" CornerRadius="3">
          <ScrollViewer Focusable="False">
            <ItemsPresenter Margin="2"></ItemsPresenter>
          </ScrollViewer>
        </Border>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

This style draws on two brushes for painting the border and the background. The actual template is a simplified version of the standard ListBox template, but it avoids the ListBoxChrome class in favor of a simpler Border. Inside the Border is the ScrollViewer that provides the list scrolling, and an ItemsPresenter that holds all the items of the list.

This template is most notable for what it doesn't let you do—namely, configure the appearance of individual items in the list. Without this ability, the selected item is always highlighted with the familiar blue background. To change this behavior, you need to add a control template for the ListBoxItem, which is a content control that wraps the content of each individual item in the list.

As with the ListBox template, you can apply the ListBoxItem template by using an element-typed style. The following basic template wraps each item in an invisible border. Because the ListBoxItem is a content

control, you use the ContentPresenter to place the item content inside. Along with these basics are triggers that react when an item is moused over or clicked:

```
<Style TargetType="{x:Type ListBoxItem}">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type ListBoxItem}">
        <Border ... >
          <ContentPresenter />
        </Border>
        <ControlTemplate.Triggers>
          <EventTrigger RoutedEvent="ListBoxItem.MouseEnter">
            <EventTrigger.Actions>
              <BeginStoryboard>
                <Storyboard>
                  <DoubleAnimation Storyboard.TargetProperty="FontSize"
                   To="20" Duration="0:0:1"></DoubleAnimation>
                </Storyboard>
              </BeginStoryboard>
            </EventTrigger.Actions>
          </EventTrigger>
          <EventTrigger RoutedEvent="ListBoxItem.MouseLeave">
            <EventTrigger.Actions>
              <BeginStoryboard>
                <Storyboard>
                  <DoubleAnimation Storyboard.TargetProperty="FontSize"
                   BeginTime="0:0:0.5" Duration="0:0:0.2"></DoubleAnimation>
                </Storyboard>
              </BeginStoryboard>
            </EventTrigger.Actions>
          </EventTrigger>

          <Trigger Property="IsMouseOver" Value="True">
            <Setter TargetName="Border" Property="BorderBrush" ... />
          </Trigger>
          <Trigger Property="IsSelected" Value="True">
            <Setter TargetName="Border" Property="Background" ... />
            <Setter TargetName="Border" Property="TextBlock.Foreground" ... />
          </Trigger>
        </ControlTemplate.Triggers>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

Together, these two templates allow you to create a list box that uses animation to enlarge the item over which the mouse is currently positioned. Because each ListBoxItem can have its own animation, when you run your mouse up and down the list, you'll see several items start to grow and then shrink back again, creating an intriguing "fish-eye" effect. (A more extravagant fish-eye effect would enlarge and warp the item over which you're hovering, using animated transforms.)

Although it's not possible to capture this effect in a single image, Figure 17-9 shows a snapshot of this list after the mouse has moved rapidly over several items.

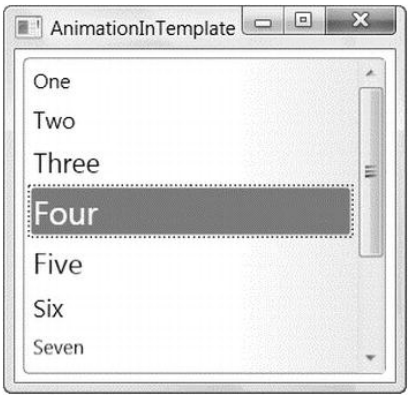

Figure 17-9. *Using a custom template for the ListBoxItem*

You won't reconsider the entire template ListBoxItem example here, because it's built from many pieces that style the ListBox, the ListBoxItem, and the various constituents of the ListBox (such as the scrollbar). The important piece is the style that changes the ListBoxItem template.

In this example, the ListBoxItem enlarges relatively slowly (over 1 second) and then decreases much more quickly (in 0.2 seconds). However, there is a 0.5-second delay before the shrinking animation begins.

Note that the shrinking animation leaves out the From and To properties. That way, it always shrinks the text from its current size to its original size. If you move the mouse on and off a ListBoxItem, you'll get the result you expect—it appears as though the item simply continues expanding while the mouse is on top and continues shrinking when the mouse is moved away.

■ **Tip** As always, the best way to get used to these conventions is to play with the template browser shown earlier to look at the control templates for basic controls. You can then copy and edit the template to use it as a basis for your custom work.

Modifying the Scrollbar

There's one aspect of the list box that's remained out of touch: the scrollbar on the right. It's part of the ScrollViewer, which is part of the ListBox template. Even though this example redefines the ListBox template, it doesn't alter the ScrollViewer of the ScrollBar.

To customize this detail, you could create a new ScrollViewer template for use with the ListBox. You could then point the ScrollViewer template to your custom ScrollBar template. However, there's an easier option. You can create an element-typed style that changes the template of all the ScrollBar controls it comes across. This avoids the extra work of creating the ScrollViewer template.

Of course, you also need to think about how this design affects the rest of your application. If you create an element-typed style ScrollBar and add it to the Resources collection of a window, all the controls in that window will have the newly styled scrollbars whenever they use the ScrollBar control, which may

be exactly what you want. On the other hand, if you want to change only the scrollbar in the ListBox, you must add the element-typed ScrollBar style to the resources collection of the ListBox itself. And finally, it you want to change the way scrollbars look in your entire application, you can add it to the resources collection in the App.xaml file.

The ScrollBar control is surprisingly sophisticated. It's actually built out of a collection of smaller pieces, as shown in Figure 17-10.

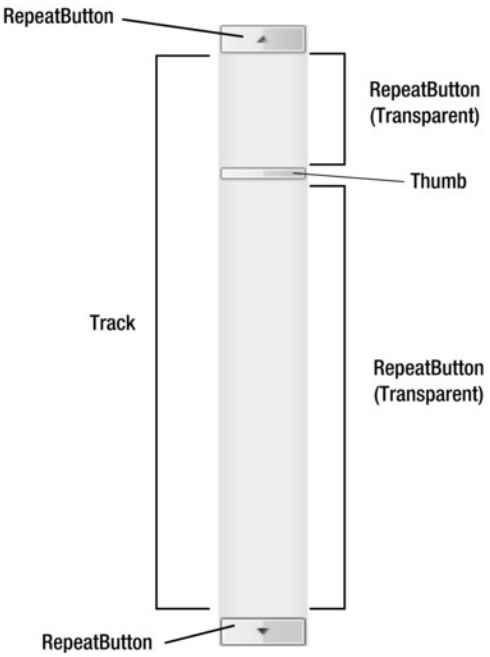

Figure 17-10. *Dissecting the scrollbar*

The background of the scrollbar is represented by the Track class—it's usually a shaded rectangle that's stretched out over the length of the scrollbar. At the far ends of the scrollbar are buttons that allow you to move one increment up or down (or to the left or right). These are instances of the RepeatButton class, which derives from ButtonBase. The key difference between a RepeatButton and the ordinary Button class is that if you hold the mouse down on a RepeatButton, the Click event fires over and over again (which is handy for scrolling).

In the middle of the scrollbar is a Thumb that represents the current position in the scrollable content. And, most interesting of all, the blank space on either side of the thumb is made up of two more RepeatButton objects, which are transparent. When you click either one of these, the scrollbar scrolls an entire *page* (a page is defined as the amount that fits in the visible window of the scrollable content). This gives you the familiar ability to jump quickly through scrollable content by clicking the bar on either side of the thumb.

Here's the template for a vertical scrollbar:

```
<ControlTemplate x:Key="VerticalScrollBar" TargetType="{x:Type ScrollBar}">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition MaxHeight="18"/>
```

```
        <RowDefinition Height="*"/>
        <RowDefinition MaxHeight="18"/>
      </Grid.RowDefinitions>

      <RepeatButton Grid.Row="0" Height="18"
        Style="{StaticResource ScrollBarLineButtonStyle}"
        Command="ScrollBar.LineUpCommand" >
        <Path Fill="{StaticResource GlyphBrush}"
         Data="M 0 4 L 8 4 L 4 0 Z"></Path>
      </RepeatButton>

      <Track Name="PART_Track" Grid.Row="1"
        IsDirectionReversed="True" ViewportSize="0">
        <Track.DecreaseRepeatButton>
          <RepeatButton Command="ScrollBar.PageUpCommand"
           Style="{StaticResource ScrollBarPageButtonStyle}">
          </RepeatButton>
        </Track.DecreaseRepeatButton>
        <Track.Thumb>
          <Thumb Style="{StaticResource ScrollBarThumbStyle}">
          </Thumb>
        </Track.Thumb>
        <Track.IncreaseRepeatButton>
          <RepeatButton Command="ScrollBar.PageDownCommand"
           Style="{StaticResource ScrollBarPageButtonStyle}">
          </RepeatButton>
        </Track.IncreaseRepeatButton>
      </Track>

      <RepeatButton
        Grid.Row="3" Height="18"
        Style="{StaticResource ScrollBarLineButtonStyle}"
        Command="ScrollBar.LineDownCommand"
        Content="M 0 0 L 4 4 L 8 0 Z">
      </RepeatButton>

      <RepeatButton
        Grid.Row="3" Height="18"
        Style="{StaticResource ScrollBarLineButtonStyle}"
        Command="ScrollBar.LineDownCommand">
        <Path Fill="{StaticResource GlyphBrush}"
          Data="M 0 0 L 4 4 L 8 0 Z"></Path>
      </RepeatButton>
    </Grid>
  </ControlTemplate>
```

This template is fairly straightforward, once you understand the multipart structure of the scrollbar (as shown in Figure 17-10). There are a few key points to note:

- The vertical scrollbar consists of a three-row grid. The top and bottom rows hold the buttons at either end (and appear as arrows). They're fixed at 18 units. The middle section, which holds the track, takes the rest of the space.
- The RepeatButton elements at both ends use the same style. The only difference is the Content property that contains a Path that draws the arrow— because the top button has an up arrow, while the bottom button has a down arrow. For conciseness, these arrows are represented by using the path mini-language described in Chapter 13. The other details, such as the background fill and the circle that appears around the arrow, are defined in the control template, which is set out in the ScrollButtonLineStyle.
- Both buttons are linked to a command in the ScrollBar class (LineUpCommand and LineDownCommand). This is how they do their work. As long as you provide a button that's linked to this command, it doesn't matter what its name is, what it looks like, or what specific class it uses. (Commands are covered in detail in Chapter 9.)
- The Track has the name PART_Track. You must use this name in order for the ScrollBar class to hook up its code successfully. If you look at the default template for the ScrollBar class (which is similar, but lengthier), you'll see it appears there as well.

■ **Note** If you're examining a control with reflection (or using a tool such as Reflector), you can look for the TemplatePart attributes attached to the class declaration. There should be one TemplatePart attribute for each named part. The TemplatePart attribute indicates the name of the expected element (through the Name property) and its class (through the Type property). In Chapter 18, you'll see how to apply the TemplatePart attribute to your own custom control classes.

- The Track.ViewportSize property is set to 0. This is a specific implementation detail in this template. It ensures that the Thumb always has the same size. (Ordinarily, the thumb is sized proportionately based on the content so that if you're scrolling through content that mostly fits in the window, the thumb becomes much larger.)
- The Track wraps two RepeatButton objects (whose style is defined separately) and the Thumb. Once again, these buttons are wired up to the appropriate functionality by using commands.

You'll also notice that the template uses a key name that specifically identifies it as a vertical scrollbar. As you learned in Chapter 11, when you set a key name on a style, you ensure that it isn't applied automatically, even if you've also set the TargetType property. The reason this example uses this approach is that the template is suitable only for scrollbars in the vertical orientation. Another, element-typed style uses a trigger to automatically apply the control template if the ScrollBar.Orientation property is set to Vertical:

```
<Style TargetType="{x:Type ScrollBar}">
  <Setter Property="SnapsToDevicePixels" Value="True"/>
  <Setter Property="OverridesDefaultStyle" Value="true"/>
  <Style.Triggers>
    <Trigger Property="Orientation" Value="Vertical">
```

```
      <Setter Property="Width" Value="18"/>
      <Setter Property="Height" Value="Auto" />
      <Setter Property="Template" Value="{StaticResource VerticalScrollBar}" />
    </Trigger>
  </Style.Triggers>
</Style>
```

Although you could easily build a horizontal scrollbar out of the same basic pieces, this example doesn't take that step (and so retains the normally styled horizontal scrollbar).

The final task is to fill in the styles that format the various RepeatButton objects and the Thumb. These styles are relatively modest, but they do change the standard look of the scrollbar. First, the Thumb is shaped like an ellipse:

```
<Style x:Key="ScrollBarThumbStyle" TargetType="{x:Type Thumb}">
  <Setter Property="IsTabStop" Value="False"/>
  <Setter Property="Focusable" Value="False"/>
  <Setter Property="Margin" Value="1,0,1,0" />
  <Setter Property="Background" Value="{StaticResource StandardBrush}" />
  <Setter Property="BorderBrush" Value="{StaticResource StandardBorderBrush}" />
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type Thumb}">
        <Ellipse Stroke="{StaticResource StandardBorderBrush}"
         Fill="{StaticResource StandardBrush}"></Ellipse>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

Next, the arrows at either end are drawn inside nicely rounded circles. The circles are defined in the control template, while the arrows are provided from the content of the RepeatButton and inserted into the control template by using the ContentPresenter:

```
<Style x:Key="ScrollBarLineButtonStyle" TargetType="{x:Type RepeatButton}">
  <Setter Property="Focusable" Value="False"/>
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type RepeatButton}">
        <Grid Margin="1">
          <Ellipse Name="Border" StrokeThickness="1"
           Stroke="{StaticResource StandardBorderBrush}"
           Fill="{StaticResource StandardBrush}"></Ellipse>
          <ContentPresenter HorizontalAlignment="Center"
           VerticalAlignment="Center"></ContentPresenter>
        </Grid>
        <ControlTemplate.Triggers>
          <Trigger Property="IsPressed" Value="true">
            <Setter TargetName="Border" Property="Fill"
             Value="{StaticResource PressedBrush}" />
          </Trigger>
        </ControlTemplate.Triggers>
      </ControlTemplate>
```

```
    </Setter.Value>
  </Setter>
</Style>
```

The RepeatButton objects that are displayed over the track aren't changed. They simply use a transparent background so the track shows through:

```
<Style x:Key="ScrollBarPageButtonStyle" TargetType="{x:Type RepeatButton}">
  <Setter Property="IsTabStop" Value="False"/>
  <Setter Property="Focusable" Value="False"/>
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type RepeatButton}">
        <Border Background="Transparent" />
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

Unlike the normal scrollbar, in this template no background is assigned to the Track, which leaves it transparent. That way, the gently shaded gradient of the list box shows through. Figure 17-11 shows the final list box.

Figure 17-11. *A list box with a customized scrollbar*

Exploring the Control Template Examples

As you've seen, giving a new template to a common control can be a detailed task. That's because all the requirements of a control template aren't always obvious. For example, a typical ScrollBar requires a combination of two RepeatButton objects and a Track. Other control templates need elements with specific PART_ names. In the case of a custom window, you need to make sure the adorner layer is defined because some controls will require it.

Although you can discover these details by exploring the default template for a control, these default templates are often complicated and include details that aren't important and bindings that you probably

won't support anyway. Fortunately, there's a better place to get started: the ControlTemplateExamples sample project (formerly known as *SimpleStyles*).

The control template examples provide a collection of simple, streamlined templates for all WPF's standard controls, which makes them a useful jumping-off point for any custom control designer. Unlike the default control templates, these use standard colors, perform all their work declaratively (with no chrome classes), and leave out optional parts such as template bindings for less commonly used properties. The goal of control template examples is to give developers a practical starting point that they can use to design their own graphically enhanced control templates. Figure 17-12 shows about half of the control template examples.

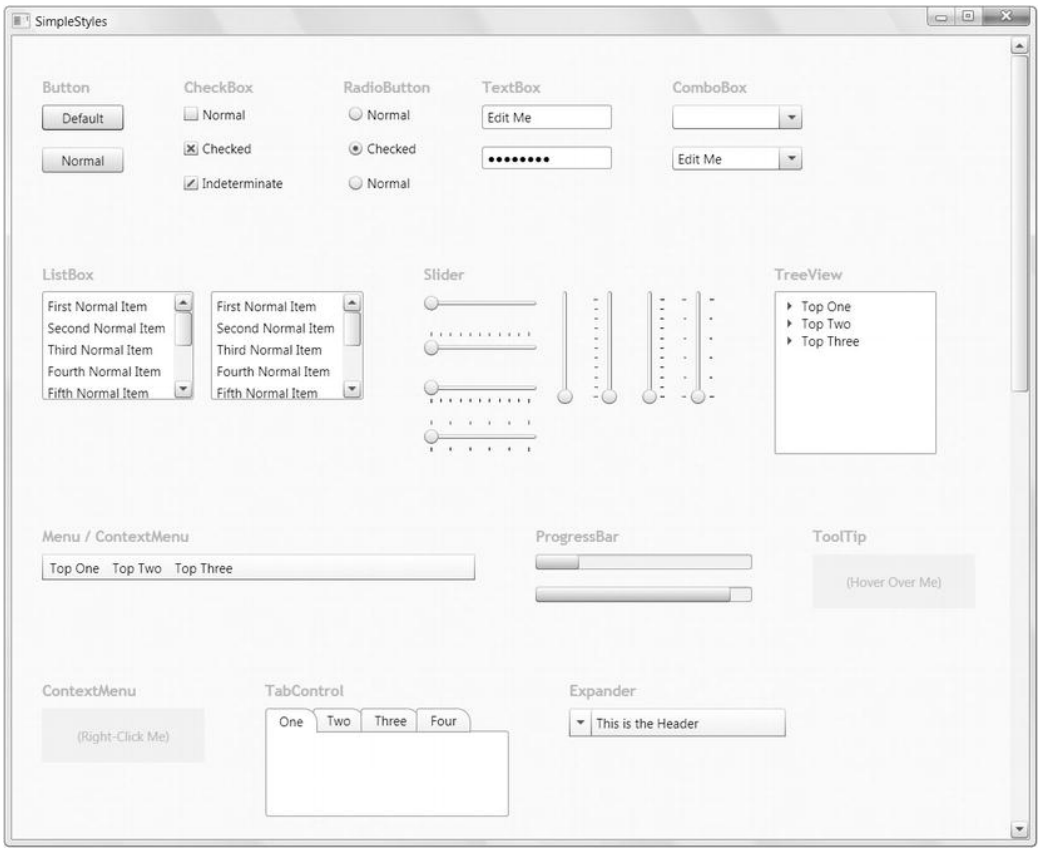

Figure 17-12. *WPF controls with bare-bones styles*

The SimpleStyles examples are included with the .NET Framework SDK. The easiest way to get them is to download them directly from `http://tinyurl.com/9jtk93x`.

■ **Tip** The SimpleStyles are one of the hidden gems of WPF. They provide templates that are easier to understand and enhance than the default control templates. If you need to enhance a common control with a custom look, this project should be your first stop.

Visual States

So far, you've learned the most direct (and most popular) way to craft a control template: using a mix of elements, binding expressions, and triggers. The elements create the overall visual structure of the control. The bindings pull information from the properties of the control class and apply it to the elements inside. And the triggers create the interactivity, allowing the control to change its appearance when its state changes.

The advantage of this model is that it's extremely powerful and extremely flexible. You can do just about anything you want. This isn't immediately apparent in the button example, because the control template relies on built-in properties such as IsMouseOver and IsPressed. But even if these properties weren't available, you could still craft a control template that changes itself in response to mouse movements and button clicks. The trick would be to use event triggers that apply animations. For example, you could add an event trigger that reacts to the Border.MouseOver by starting an animation that changes the border background color. This animation doesn't even need to look like an animation—if you give it a duration of 0 seconds, it will apply itself immediately, just like the property triggers you're using now. In fact, this exact technique is used in a number of professional template examples.

Despite their capabilities, trigger-based templates have a downside: they require that the template designer has a detailed understanding of the way the control works. In the button example, the template designer needs to know about the existence of the IsMouseOver and IsPressed properties, for example, and how to use them. And these aren't the only details—for example, most controls need to react visually to mouse movements, being disabled, getting focus, and many other state changes. When these states are applied in combination, it can be difficult to determine exactly how the control should look. The trigger-based model is also notoriously awkward with *transitions*. For example, imagine you want to create a button that pulses while the mouse is over it. To get a professional result, you may need two animations—one that changes the state of the button from normal to mouseover and one that applies the continuous pulsing effect immediately after that. Managing all these details with a trigger-based template can be a challenge.

In WPF 4, Microsoft added a feature called *visual states*, which addresses this challenge. Using named parts (which you've already seen) and visual states, a control can provide a standardized visual contract. Rather than understanding the entire control, a template designer simply needs to understand the rules of the visual contract. As a result, it's much easier to design a simple control template—especially when it's for a control you've never worked with before.

Much as controls can use the TemplatePart attribute to indicate specific named elements (or parts) that the control template should include, they can use the TemplateVisualState attribute to indicate the visual states they support. For example, an ordinary button would provide a set of visual states like this:

```
[TemplateVisualState(Name="Normal", GroupName="CommonStates")]
[TemplateVisualState(Name="MouseOver", GroupName="CommonStates")]
[TemplateVisualState(Name="Pressed", GroupName="CommonStates")]
[TemplateVisualState(Name="Disabled", GroupName="CommonStates")]
[TemplateVisualState(Name="Unfocused", GroupName="FocusStates")]
[TemplateVisualState(Name="Focused", GroupName="FocusStates")]
public class Button : ButtonBase
{ ... }
```

States are placed together in *groups*. Groups are mutually exclusive, which means a control has one state in each group. For example, the button shown here has two state groups: CommonStates and FocusStates. At any given time, the button has one of the states from the CommonStates group *and* one of the states from the FocusStates group.

For example, if you tab to the button, its states will be Normal (from CommonStates) and Focused (from FocusStates). If you then move the mouse over the button, its states will be MouseOver (from

CommonStates) and Focused (from FocusStates). Without state groups, you'd have trouble dealing with this situation. You'd either be forced to make some states dominate others (so a button in the MouseOver state would lose its focus indicator) or need to create many more states (such as FocusedNormal, UnfocusedNormal, FocusedMouseOver, UnfocusedMouseOver, and so on).

At this point, you can already see the appeal of the visual states model. From the template, it's immediately clear that a control template needs to address six state possibilities. You also know the name of each state, which is the only essential detail. You don't need to know what properties the Button class provides or understand the inner workings of the control. Best of all, if you use Expression Blend, you'll get enhanced design-time support when creating control templates for a control that supports visual states. Blend will show you the named parts and visual states the control supports (as defined with the TemplatePart and TemplateVisualState attributes), and you can then add the corresponding elements and storyboards.

In the next chapter, you'll see a custom control named the FlipPanel that puts the visual state model into practice.

The Last Word

In this chapter, you learned how to use basic template-building techniques to re-skin core WPF controls like the button, without being forced to reimplement any core button functionality. These custom buttons support all the normal button behavior—you can tab from one to the next, you can click them to fire an event, you can use access keys, and so on. Best of all, you can reuse your button template throughout your application and still replace it with a whole new design at a moment's notice.

So, what more do you need to know before you can skin all the basic WPF controls? To get the snazzy look you probably want, you might need to spend more time studying the details of WPF drawing (Chapter 12 and Chapter 13) and animation (Chapter 15 and Chapter 16). It might surprise you to know that you can use the shapes and brushes you've already learned about to build sophisticated controls with glass-style blurs and soft glow effects. The secret is in combining multiple layers of shapes, each with a different gradient brush. The best way to get this sort of effect is to learn from the control template examples others have created. Here are two good examples to check out:

- There are plenty of handcrafted, shaded buttons with glass and soft glow effects on the Web. You can find an old (but still useful) tutorial that walks you through the process of creating a snazzy glass button in Expression Blend at `http://tinyurl.com/3bk26g`.
- An MSDN Magazine article about control templates provides examples of templates that incorporate simple drawings in innovative ways. For example, a CheckBox is replaced by an up-down lever, a slider is rendered with a three-dimensional tab, a ProgressBar is changed into a thermometer, and so on. Check it out at `http://msdn.microsoft.com/magazine/cc163497.aspx`.