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 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 why (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.

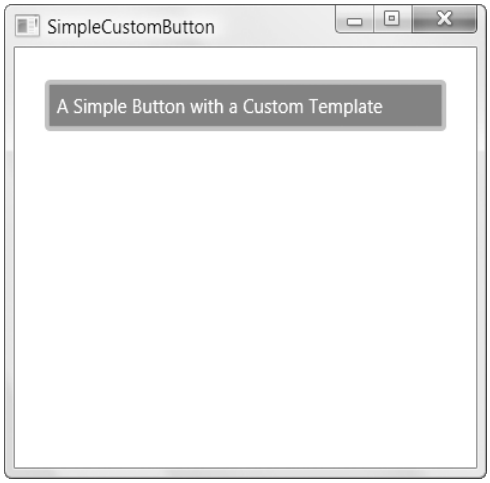

Figure 17-6. *A button with a customized control template*

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 only support 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, once 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 using a trigger. Although you could create a focus effect in many different 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 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.

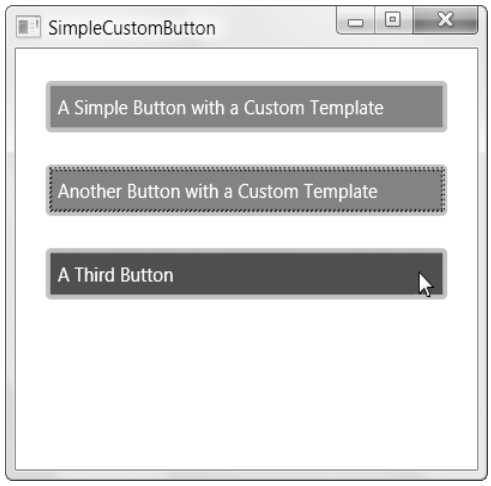

Figure 17-7. *Buttons with focus and mouseover support*

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 mover overtop. 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 using a separate one-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 ➤ 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 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 a number of different 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.

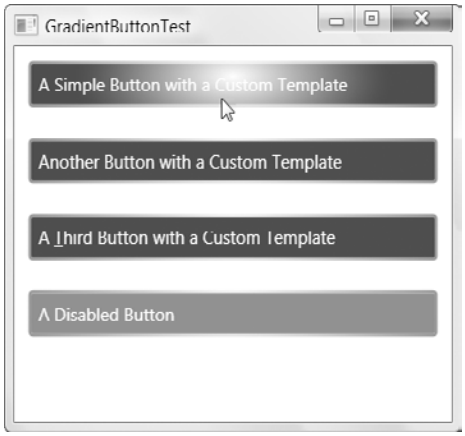

Figure 17-8. *A gradient button*

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 different 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 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 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:

- Your control represents a significant change in functionality. For example, you have a custom button, and that button adds new functionality that requires new properties or methods.
- You plan to distribute your control in a separate class library assembly so it can be used in (and customized for) a wide range of applications. In this situation, you need a higher level of standardization than is possible with control templates alone.

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 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 remarkably. 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://wpf.codeplex.com/Release/ProjectReleases.aspx?ReleaseId=14962`. You can preview the themes at `http://tinyurl.com/ylojdry`.

To use a theme, add the .xaml file that contains the resource dictionary to your project. For example, the 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.

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 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 using code like this:

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

You can also load a resource dictionary that's defined in another assembly 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 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 different pieces that style the ListBox, the ListBoxItem, and the various constituents of the ListBox (such as the scroll bar). The important piece is the style that changes the ListBoxItem template.

In this example, the ListBoxItem enlarges relatively slowly (over one 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 overtop and continues shrinking when the mouse is moved away.

■ **Tip** As always, the best way to get used to these different 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 Scroll Bar

There's one aspect of the list box that's remained out of touch: the scroll bar 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 scroll bars whenever they use the ScrollBar control, which may be exactly what you want. On the other hand, if you want to change only the scroll bar in the