Naming a User Control

In the example shown here, the top-level UserControl is assigned a name (colorPicker). This allows you to write straightforward data binding expressions that bind to properties in the custom user control class. However, this technique raises an obvious question. Namely, what happens when you create an instance of the user control in a window (or page) and assign a new name to it?

Fortunately, this situation works without a hitch, because the user control performs its initialization before that of the containing window. First, the user control is initialized, and its data bindings are connected. Next, the window is initialized, and the name that's set in the window markup is applied to the user control. The data binding expressions and event handlers in the window can now use the window-defined name to access the user control, and everything works the way you'd expect.

Although this sounds straightforward, you might notice a couple of quirks if you use code that examines the UserControl.Name property directly. For example, if you examine the Name property in an event handler in the user control, you'll see the new name that was applied by the window. Similarly, if you don't set a name in the window markup, the user control will retain the original name from the user control markup. You'll then see this name if you examine the Name property in the window code.

Neither of these quirks represents a problem, but a better approach would be to avoid naming the user control in the user control markup and use the Binding.RelativeSource property to search up the element tree until you find the UserControl parent. Here's the lengthier syntax that does this:

```
<Slider Name="sliderRed" Minimum="0" Maximum="255"
     Value="{Binding Path=Red,
               RelativeSource={RelativeSource FindAncestor,
                               AncestorType={x:Type UserControl}}
             }">
</Slider>
```

You'll see this approach later, when you build a template-based control in the section "Refactoring the Color Picker Markup."

Using the Control

Now that you've completed the control, using it is easy. To use the color picker in another window, you need to begin by mapping the assembly and .NET namespace to an XML namespace, as shown here:

```
<Window x:Class="CustomControlsClient.ColorPickerUserControlTest"
 xmlns:lib="clr-namespace:CustomControls;assembly=CustomControls" ... >
```

Using the XML namespace you've defined and the user control class name, you can create your user control exactly as you create any other type of object in XAML markup. You can also set its properties and attach event handlers directly in the control tag, as shown here:

```
<lib:ColorPickerUserControl Name="colorPicker" Color="Beige"
 ColorChanged="colorPicker_ColorChanged"></lib:ColorPickerUserControl>
```

Because the Color property uses the Color data type and the Color data type is decorated with a TypeConverter attribute, WPF knows to use the ColorConverter to change the string color name into the corresponding Color object before setting the Color property.

The code that handles the ColorChanged event is straightforward:

```
private void colorPicker_ColorChanged(object sender,
  RoutedPropertyChangedEventArgs<Color> e)
{
    lblColor.Text = "The new color is " + e.NewValue.ToString();
}
```

This completes your custom control. However, there's still one frill worth adding. In the next section, you'll enhance the color picker with support for WPF's command feature.

Command Support

Many controls have baked-in command support. You can add this to your controls in two ways:

- Add command bindings that link your control to specific commands. That way, your control can respond to a command without the help of any external code.
- Create a new RoutedUICommand object for your command as a static field in your control. Then, add a command binding for that command. This allows your control to automatically support commands that aren't already defined in the basic set of command classes that you learned about in Chapter 9.

In the following example, you'll use the first approach to add support for the ApplicationCommands.Undo command.

■ **Tip** For more information about commands and how to create custom RoutedUICommand objects, refer to Chapter 9.

To support an Undo feature in the color picker, you need to track the previous color in a member field:

```
private Color? previousColor;
```

It makes sense to make this field nullable, because when the control is first created, there shouldn't be a previous color set. (You can also clear the previous color programmatically after an action that you want to make irreversible.)

When the color is changed, you simply need to record the old value. You can take care of this task by adding this line to the end of the OnColorChanged() method:

```
colorPicker.previousColor = (Color)e.OldValue;
```

Now you have the infrastructure in place that you need to support the Undo command. All that's left is to create the command binding that connects your control to the command and handle the CanExecute and Executed events.

The best place to create command bindings is when the control is first created. For example, the following code uses the color picker's constructor to add a command binding to the ApplicationCommands.Undo command:

```
public ColorPicker()
{
    InitializeComponent();
    SetUpCommands();
}

private void SetUpCommands()
{
    // Set up command bindings.
    CommandBinding binding = new CommandBinding(ApplicationCommands.Undo,
      UndoCommand_Executed, UndoCommand_CanExecute);

    this.CommandBindings.Add(binding);
}
```

To make your command functional, you need to handle the CanExecute event and allow the command as long as there is a previous value:

```
private void UndoCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = previousColor.HasValue;
}
```

Finally, when the command is executed, you can swap in the new color.

```
private void UndoCommand_Executed(object sender, ExecutedRoutedEventArgs e)
{
    this.Color = (Color)previousColor;
}
```

You can trigger the Undo command in two different ways. You can use the default Ctrl+Z key binding when an element in the user control has focus, or you can add a button to the client that triggers the command, like this one:

```
<Button Command="Undo" CommandTarget="{Binding ElementName=colorPicker}">
  Undo
</Button>
```

Either way, the current color is abandoned and the previous color is applied.

■ **Tip** The current example stores just one level of undo information. However, it's easy to create an undo stack that stores a series of values. You just need to store Color values in the appropriate type of collection. The Stack collection in the System.Collections.Generic namespaces is a good choice, because it implements a last-in first-out approach that makes it easy to grab the most recent Color object when performing an undo operation.

More Robust Commands

The technique described earlier is a perfectly legitimate way to connect commands to controls, but it's not the technique that's used in WPF elements and professional controls. These elements use a more robust approach and attach static command handlers using the CommandManager.RegisterClassCommandBinding() method.

The problem with the implementation shown in the previous example is that it uses the public CommandBindings collection. This makes it a bit fragile, because the client can modify the CommandBindings collection freely. This isn't possible if you use the RegisterClassCommandBinding() method. This is the approach that WPF controls use. For example, if you look at the CommandBindings collection of a TextBox, you won't find any of the bindings for hardwired commands such as Undo, Redo, Cut, Copy, and Paste, because these are registered as class bindings.

The technique is fairly straightforward. Instead of creating the command binding in the instance constructor, you must create the command binding in the static constructor, using code like this:

```
CommandManager.RegisterClassCommandBinding(typeof(ColorPicker),
  new CommandBinding(ApplicationCommands.Undo,
    UndoCommand_Executed, UndoCommand_CanExecute));
```

Although this code hasn't changed much, there's an important shift. Because the UndoCommand_Executed() and UndoCommand_CanExecute() methods are referred to in the constructor, they must both be static methods. To retrieve instance data (such as the current color and the previous color information), you need to cast the event sender to a ColorPicker object and use it.

Here's the revised command handling code:

```
private static void UndoCommand_CanExecute(object sender,
  CanExecuteRoutedEventArgs e)
{
    ColorPicker colorPicker = (ColorPicker)sender;
    e.CanExecute = colorPicker.previousColor.HasValue;
}

private static void UndoCommand_Executed(object sender,
  ExecutedRoutedEventArgs e)
{
    ColorPicker colorPicker = (ColorPicker)sender;
    Color currentColor = colorPicker.Color;
    colorPicker.Color = (Color)colorPicker.previousColor;
}
```

Incidentally, this technique isn't limited to commands. If you want to hardwire event handling logic into your control, you can use a class event handler with the EventManager.RegisterClassHandler() method. Class event handlers are always invoked before instance event handlers, allowing you to easily suppress events.

A Closer Look at User Controls

User controls provide a fairly painless but somewhat limited way to create a custom control. To understand why, it helps to take a closer look at how user controls work.

Behind the scenes, the UserControl class works a lot like the ContentControl class from which it derives. In fact, it has just a few key differences:

- The UserControl class changes some default values. Namely, it sets IsTabStop and Focusable to false (so it doesn't occupy a separate place in the tab order), and it sets HorizontalAlignment and VerticalAlignment to Stretch (rather than Left and Top) so it fills the available space.
- The UserControl class applies a new control template that consists of a Border element that wraps a ContentPresenter. The ContentPresenter holds the content you add using markup.
- The UserControl class changes the source of routed events. When events bubble or tunnel from controls inside the user control to elements outside the user control, the source changes to point to the user control rather than the original element. This gives you a bit more encapsulation. (For example, if you handle the UIElement.MouseLeftButtonDown event in the layout container that holds the color picker, you'll receive an event when you click the Rectangle inside. However, the source of this event won't be the Rectangle element but the ColorPicker object that contains the Rectangle. If you create the same color picker as an ordinary content control, this isn't the case—it's up to you to intercept the event in your control, handle it, and reraise it.)

The most significant difference between user controls and other types of custom controls is the way that a user control is designed. Like all controls, user controls have a control template. However, you'll rarely change this template—instead, you'll supply the markup as part of your custom user control class, and this markup is processed using the InitializeComponent() method when the control is created. On the other hand, a lookless control has no markup—everything it needs is in the template.

An ordinary ContentControl has the following stripped-down template:

```
<ControlTemplate TargetType="ContentControl">
  <ContentPresenter
   ContentTemplate="{TemplateBinding ContentControl.ContentTemplate}"
   Content="{TemplateBinding ContentControl.Content}" />
</ControlTemplate>
```

This template does little more than fill in the supplied content and apply the optional content template. Properties such as Padding, Background, HorizontalAlignment, and VerticalAlignment won't have any effect unless you explicitly bind to it.

The UserControl has a similar template with a few more niceties. Most obviously, it adds a Border element and binds its properties to the BorderBrush, BorderThickness, Background, and Padding properties of the user control to make sure they have some meaning. Additionally, the ContentPresenter inside binds to the alignment properties.

```
<ControlTemplate TargetType="UserControl">
  <Border BorderBrush="{TemplateBinding Border.BorderBrush}"
   BorderThickness="{TemplateBinding Border.BorderThickness}"
   Background="{TemplateBinding Panel.Background}" SnapsToDevicePixels="True"
   Padding="{TemplateBinding Control.Padding}">

    <ContentPresenter
     HorizontalAlignment="{TemplateBinding Control.HorizontalContentAlignment}"
     VerticalAlignment="{TemplateBinding Control.VerticalContentAlignment}"
     SnapsToDevicePixels="{TemplateBinding UIElement.SnapsToDevicePixels}"
     ContentTemplate="{TemplateBinding ContentControl.ContentTemplate}"
     Content="{TemplateBinding ContentControl.Content}" />

  </Border>
</ControlTemplate>
```

Technically, you could change the template of a user control. In fact, you could move all your markup into the template, with only slight readjusting. But there's really no reason to take this step—if you want a more flexible control that separates the visual look from the interface that's defined by your control class, you'd be much better off creating a custom lookless control, as described in the next section.

Creating a Lookless Control

The goal of user controls is to provide a design surface that supplements the control template, giving you a quicker way to define the control at the price of future flexibility. This causes a problem if you're happy with the functionality of a user control, but you need to tailor its visual appearance. For example, imagine you want to use the same color picker but give it a different "skin" that blends better into an existing application window. You may be able to change some aspects of the user control through styles, but parts of it are locked away inside, hard-coded into the markup. For example, there's no way to move the preview rectangle to the left side of the sliders.

The solution is to create a lookless control—a control that derives from one of the control base classes but doesn't have a design surface. Instead, this control places its markup into a default template that can be replaced at will without disturbing the control logic.

Refactoring the Color Picker Code

Changing the color picker into a lookless control isn't too difficult. The first step is easy—you simply need to change the class declaration, as shown here:

```
public class ColorPicker : System.Windows.Controls.Control
{ ... }
```

In this example, the ColorPicker class derives from Control. FrameworkElement isn't suitable, because the color picker does allow user interaction and the other higher-level classes don't accurately describe the color picker's behavior. For example, the color picker doesn't allow you to nest other content inside, so the ContentControl class isn't appropriate.

The code inside the ColorPicker class is the same as the code for the user control (aside from the fact that you must remove the call to InitializeComponent() in the constructor). You follow the same approach to define dependency properties and routed events. The only difference is that you need to tell WPF that you will be providing a new style for your control class. This style will provide the new control template. (If you don't take this step, you'll continue whatever template is defined in the base class.)

To tell WPF that you're providing a new style, you need to call the OverrideMetadata() method in the static constructor of your class. You call this method on the DefaultStyleKeyProperty, which is a dependency property that defines the default style for your control. The code you need is as follows:

```
DefaultStyleKeyProperty.OverrideMetadata(typeof(ColorPicker),
  new FrameworkPropertyMetadata(typeof(ColorPicker)));
```

You could supply a different type if you want to use the template of another control class, but you'll almost always create a specific style for each one of your custom controls.

Refactoring the Color Picker Markup

Once you've added the added the call to OverrideMetadata, you simply need to plug in the right style. This style needs to be placed in a resource dictionary named `generic.xaml`, which must be placed in a Themes subfolder in your project. That way, your style will be recognized as the default style for your control. Here's how to add the generic.xaml file:

1. Right-click the class library project in the Solution Explorer, and choose Add ➤ New Folder.
2. Name the new folder Themes.
3. Right-click the Themes folder, and choose Add ➤ New Item.
4. In the Add New Item dialog box, pick the XML file template, enter the name generic.xaml, and click Add.

Figure 18-3 shows the generic.xaml file in the Themes folder.

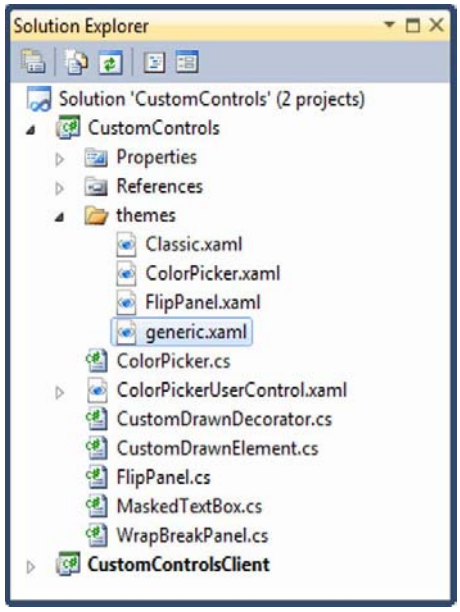

Figure 18-3. *A WPF application and class library*

Often, a custom control library has several controls. To keep their styles separate for easier editing, the generic.xaml file often uses resource dictionary merging. The following markup shows a generic.xaml file that pulls in the resources from the ColorPicker.xaml resource dictionary in the same Themes subfolder of a control library named CustomControls:

```
<ResourceDictionary
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" >
  <ResourceDictionary.MergedDictionaries>
    <ResourceDictionary Source="/CustomControls;component/themes/ColorPicker.xaml">
    </ResourceDictionary>
  </ResourceDictionary.MergedDictionaries>

</ResourceDictionary>
```

Your custom control style must use the TargetType attribute to attach itself to the color picker automatically. Here's the basic structure of the markup that appears in the ColorPicker.xaml file:

```
<ResourceDictionary
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 xmlns:local="clr-namespace:CustomControls">
  <Style TargetType="{x:Type local:ColorPicker}">
    ...
  </Style>
</ResourceDictionary>
```

You can use your style to set any properties in the control class (whether they're inherited from the base class or new properties you've added). However, the most useful task that your style performs is to apply a new template that defines the default visual appearance of your control.

It's fairly easy to convert ordinary markup (like that used by the color picker) into a control template. Keep these considerations in mind:

- When creating binding expressions that link to properties in the parent control class, you can't use the ElementName property. Instead, you need to use the RelativeSource property to indicate that you want to bind to the parent control. If one-way data binding is all that you need, you can usually use the lightweight TemplateBinding markup extension instead of the full-fledged Binding.
- You can't attach event handlers in the control template. Instead, you'll need to give your elements recognizable names and attach event handlers to them programmatically in the control constructor.
- Don't name an element in a control template unless you want to attach an event handler or interact with it programmatically. When naming an element you want to use, give it a name in the form PART_*ElementName*.

With these considerations in mind, you can create the following template for the color picker. The most important changed details are highlighted in bold.

```
<Style TargetType="{x:Type local:ColorPicker}">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type local:ColorPicker}">
        <Grid>
          <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
          </Grid.RowDefinitions>
          <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
          </Grid.ColumnDefinitions>

          <Slider Minimum="0" Maximum="255"
           Margin="{TemplateBinding Padding}"
           Value="{Binding Path=Red,
                   RelativeSource={RelativeSource TemplatedParent}}">
          </Slider>
          <Slider Grid.Row="1" Minimum="0" Maximum="255"
           Margin="{TemplateBinding Padding}"
           Value="{Binding Path=Red,
                   RelativeSource={RelativeSource TemplatedParent}}">
          </Slider>
          <Slider Grid.Row="2" Minimum="0" Maximum="255"
           Margin="{TemplateBinding Padding}"
           Value="{Binding Path=Red,
                   RelativeSource={RelativeSource TemplatedParent}}">
```

```
          </Slider>

          <Rectangle Grid.Column="1" Grid.RowSpan="3"
           Margin="{TemplateBinding Padding}"
           Width="50" Stroke="Black" StrokeThickness="1">
            <Rectangle.Fill>
              <SolidColorBrush
               Color="{Binding Path=Color,
                   RelativeSource={RelativeSource TemplatedParent}}">
              </SolidColorBrush>
            </Rectangle.Fill>
          </Rectangle>
        </Grid>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

As you'll notice, some binding expressions have been replaced with the TemplateBinding extension. Others still use the Binding extension but have the RelativeSource set to point to the template parent (the custom control). Although both TemplateBinding and Binding with a RelativeSource of TemplatedParent are for the same purpose—extracting data from the properties of your custom control—the lighter-weight TemplateBinding is always appropriate. It won't work if you need two-way binding (as with the sliders) or when binding to the property of a class that derives from Freezable (like the SolidColorBrush).

Streamlining the Control Template

As it stands, the color picker control template fills in everything you need, and you can use it in the same way that you use the color picker user control. However, it's still possible to simplify the template by removing some of the details.

Currently, any control consumer that wants to supply a custom template will be forced to add a slew of data binding expressions to ensure that the control continues to work. This isn't difficult, but it is tedious. Another option is to configure all the binding expressions in the initialization code of the control itself. This way, the template doesn't need to specify these details.

■ **Note** This is the same technique you use when attaching event handlers to the elements that make up a custom control. You attach each event handler programmatically, rather than use event attributes in the template.

Adding Part Names

For this system to work, your code needs to be able to find the elements it needs. WPF controls locate the elements they need by name. As a result, your element names become part of the public interface of your control and need suitably descriptive names. By convention, these names begin with the text *PART_* followed by the element name. The element name uses initial caps, just like a property name.

PART_RedSlider is a good choice for a required element name, while PART_sldRed, PART_redSlider, and RedSlider are all poor choices.

For example, here's how you would prepare the three sliders for programmatic binding, by removing the binding expression from the Value property and adding a PART_ name:

```
<Slider Name="PART_RedSlider" Minimum="0" Maximum="255"
 Margin="{TemplateBinding Padding}"></Slider>
<Slider Grid.Row="1" Name="PART_GreenSlider" Minimum="0" Maximum="255"
 Margin="{TemplateBinding Padding}"></Slider>
<Slider Grid.Row="2" Name="PART_BlueSlider" Minimum="0" Maximum="255"
 Margin="{TemplateBinding Padding}"></Slider>
```

Notice that the Margin property still uses a binding expression to add padding, but this is an optional detail that can easily be left out of custom template (which may choose to hard-code the padding or use a different layout).

To ensure maximum flexibility, the Rectangle isn't given a name. Instead, the SolidColorBrush inside is given a name. That way, the color preview feature can be used with any shape or an arbitrary element, depending on the template.

```
<Rectangle Grid.Column="1" Grid.RowSpan="3"
 Margin="{TemplateBinding Padding}"
 Width="50" Stroke="Black" StrokeThickness="1">
  <Rectangle.Fill>
    <SolidColorBrush x:Name="PART_PreviewBrush"></SolidColorBrush>
  </Rectangle.Fill>
</Rectangle>
```

Manipulating Template Parts

You could connect your binding expressions when the control is initialized, but there's a better approach. WPF has a dedicated OnApplyTemplate() method that you should override if you need to search for elements in the template and attach event handlers or add data binding expressions. In that method, you can use the GetTemplateChild() method (which is inherited from FrameworkElement) to find the elements you need.

If you don't find an element that you want to work with, the recommended pattern is to do nothing. Optionally, you can add code that checks that the element, if present, is the correct type and raises an exception if it isn't. (The thinking here is that a missing element represents a conscious opting out of a specific feature, whereas an incorrect element type represents a mistake.)

Here's how you can connect the data binding expression for a single slider in the OnApplyTemplate() method:

```
public override void OnApplyTemplate()
{
    base.OnApplyTemplate();

    RangeBase slider = GetTemplateChild("PART_RedSlider") as RangeBase;
    if (slider != null)
    {
        // Bind to the Red property in the control, using a two-way binding.
        Binding binding = new Binding("Red");
        binding.Source = this;
```

```
        binding.Mode = BindingMode.TwoWay;
        slider.SetBinding(RangeBase.ValueProperty, binding);
    }
    ...
}
```

Notice that the code uses the System.Windows.Controls.Primitives.RangeBase class (from which Slider derives) instead of the Slider class. That's because the RangeBase class provides the minimum required functionality—in this case, the Value property. By making the code as generic as possible, the control consumer gains more freedom. For example, it's now possible to supply a custom template that uses a different RangeBase-derived control in place of the color sliders.

The code for binding the other two sliders is virtually identical. The code for binding the SolidColorBrush is slightly different, because the SolidColorBrush does not include the SetBinding() method (which is defined in the FrameworkElement class). One easy workaround is to create a binding expression for the ColorPicker.Color property, which uses the one-way-to-source direction. That way, when the color picker's color is changed, the brush is updated automatically.

```
SolidColorBrush brush = GetTemplateChild("PART_PreviewBrush") as SolidColorBrush;
if (brush != null)
{
    Binding binding = new Binding("Color");
    binding.Source = brush;
    binding.Mode = BindingMode.OneWayToSource;
    this.SetBinding(ColorPicker.ColorProperty, binding);
}
```

To see the benefit of this change in design, you need to create a control that uses the color picker but supplies a new control template. Figure 18-4 shows one possibility.

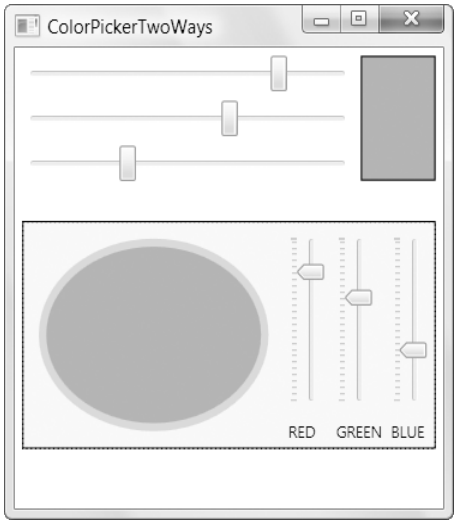

Figure 18-4. *A color picker custom control with two different templates*

Documenting Template Parts

There's one last refinement that you should make to the previous example. Good design guidelines suggest that you add the TemplatePart attribute to your control declaration to document what part names you use in your template and what type of control you use for each part. Technically, this step isn't required, but it's a piece of documentation that can help others who are using your class (and it can also be inspected by design tools that let you build customized control templates, such as Expression Blend).

Here are the TemplatePart attributes you should add to the ColorPicker control class:

```
[TemplatePart(Name="PART_RedSlider", Type=typeof(RangeBase))]
[TemplatePart(Name = "PART_BlueSlider", Type=typeof(RangeBase))]
[TemplatePart(Name="PART_GreenSlider", Type=typeof(RangeBase))]
public class ColorPicker : System.Windows.Controls.Control
{ ... }
```

Theme-Specific Styles and the Default Style

As you've seen, the ColorPicker gets its default control template from a file named generic.xaml, which is placed in a project folder named Themes. This slightly strange convention is actually part of the theme support that's built into WPF.

The Themes folder holds the default styles for the controls you create, customized for the different versions and themes of the Windows operating system. If you aren't interested in creating theme-specific styles, all you need is the generic.xaml file. This resource dictionary holds the fallback styles that are used for your controls if no theme-specific files are available.

If you *do* want to create controls that are aware of the current theme and vary themselves in minor or major ways, you simply need to add the right files to the Themes folder. Table 18-2 lists the themes you can set, and the file name you need to use for your resource dictionary. If you choose not to supply a file for a specific theme, your control falls back to the generic.xaml dictionary when that theme is active.

■ **Note** The theme-specific resource dictionaries are used to set the default control style (which should contain the default control template). However, no matter what default or theme-specific styles you use, you're always free to replace the control template by setting the Template property of a control object.

Table 18-2. *File Names for Theme-Specific Resource Dictionaries*

Operating System	Base Theme Name	Theme Color Name	File Name
Windows Vista or Windows 7 (default)	Aero	NormalColor	Aero.NormalColor.xaml
Windows XP (blue, the default)	Luna	NormalColor	Luna.NormalColor.xaml

Operating System	Base Theme Name	Theme Color Name	File Name
Windows XP (olive green)	Luna	Homestead	Luna.Homestead.xaml
Windows XP (silver)	Luna	Metallic	Luna.Metallic.xaml
Window XP Media Center Edition 2005	Royale	Normal	Royale.NormalColor.xaml
Windows XP (Zune, released separately)	Zune	NormalColor	Zune.NormalColor.xaml
Windows XP or Windows Vista	Classic		Classic.xaml

The set of defined themes is relatively small (although new ones may be added to this list in the future). Windows Vista and Windows 7 support only two themes from this list—the standard Aero theme and the legacy Windows Classic look.

■ **Tip** It doesn't matter if the user has saved a custom theme under a new name or applied a custom color scheme. All user-created themes are based on one of the themes in the list shown in Table 18-2. This detail determines the style for your control. If needed, you can access the currently configured current system colors (and even use them in your template) using the system resources that are exposed by the SystemColors class.

If you decide to create a theme-specific look for one of your controls, you need to start by creating the appropriate resource dictionaries with the right file names. However, this step isn't quite enough to get these styles working in your application. You also need to use the ThemeInfo attribute on your assembly to enable theme support.

The ThemeInfo attribute is an assembly-level attribute that takes two parameters in its constructor. The first configures theme-specific style support, and the second configures support for the generic.xaml fallback. When you create a new WPF project in Visual Studio, the ThemeInfo attribute is added to the AssemblyInfo.cs file that configures generic.xaml support but not theme-specific style. (You can find the AssemblyInfo.cs file under the Properties node in the Solution Explorer.)

By default, the ThemeInfo attribute looks like this:

```
[assembly: ThemeInfo(ResourceDictionaryLocation.None,
 ResourceDictionaryLocation.SourceAssembly)]
```

To enable theme-specific style support, you need to change the ThemeInfo attribute to this:

```
[assembly: ThemeInfo(ResourceDictionaryLocation.SourceAssembly,
 ResourceDictionaryLocation.SourceAssembly)]
```

Although None and SourceAssembly are the two most commonly used values from the ResourceDictionaryLocation enumeration, you can also use ExternalAssembly. In this case, WPF looks for an assembly with the file name *AssemblyName.ThemeName*.dll in the same folder as your application. For example, if you've created a library named CustomControls.dll, the resources for the Windows 7/Vista styles will be found in an assembly named CustomControls.Aero.dll. Windows XP styles will be in CustomControls.Luna.dll, CustomControls.Royale.dll, and so on. (Notice that the color part of the theme name isn't used. Instead, it's assumed that you'll put all the color-specific themes in one assembly for each base theme.) You've already seen this system when you considered the chrome classes in Chapter 17 that support controls like the Button. They use resources from assemblies with names like PresentationFramework.Aero.dll and PresentationFramework.Luna.dll.

Theme Styles versus Application Styles

Every control has a default style (or several theme-dependent default styles). You call DefaultStyleKeyProperty.OverrideMetadata() in the static constructor of your control class to indicate what default style your custom control should use. If you don't your control will simply use the default style that's defined for the control that your class derives from.

Contrary to what you might expect, the default theme style is not exposed through the Style property. All the controls in the WPF library return a null reference for their Style property.

Instead, the Style property is reserved for an *application style* (the type you learned to build in Chapter 11). If you set an application style, it's merged into the default theme style. If you set an application style that conflicts with the default style, the application style wins and overrides the property setter or trigger in the default style. However, the details you don't override remain. This is the behavior you want. It allows you to create an application style that changes just a few properties (for example, the text font in a button), without removing the other essential details that are supplied in the default theme style (like the control template).

Incidentally, you can retrieve the default style programmatically. To do so, you can use the FindResource() method to search up the resource hierarchy for a style that has the right element-type key. For example, if you want to find the default style that's applied to the Button class, you can use this code statement:

```
Style style = Application.Current.FindResource(typeof(Button));
```

Supporting Visual States

The ColorPicker control is a good example of control design. Because its behavior and its visual appearance are carefully separated, other designers can develop new templates that change its appearance dramatically.

One of the reasons the ColorPicker is so simple is that it doesn't have a concept of states. In other words, it doesn't distinguish its visual appearance based on if it has focus, if the mouse is overtop, if it's disabled, and so on. The FlipPanel control in the following example is a bit different.

The basic idea behind the FlipPanel is that it provides two surfaces to host content, but only one is visible at a time. To see the other content, you "flip" between the sides. You can customize the flipping

effect through the control template, but the default effect use a simple fade that transitions between the front and back (see Figure 18-5). Depending on your application, you could use the FlipPanel to combine a data-entry form with some helpful documentation, to provide a simple or a more complex view on the same data, or to fuse together a question and an answer in a trivia game.

Figure 18-5. *Flipping the FlipPanel*

You can perform the flipping programmatically (by setting a property named IsFlipped), or the user can flip the panel using a convenient button (unless the control consumer removes it from the template).

Clearly, the control template needs to specify two separate sections: the front and back content regions of the FlipPanel. However, there's an additional detail—namely, the FlipPanel needs a way to switch between its two states: flipped and not flipped. You could do the job by adding triggers to your template. One trigger would hide the front panel and show the second panel when a button is clicked, while the other would reverse these changes. Both could use any animations you like. But by using visual states, you clearly indicate to the control consumer that these two states are a required part of the template. Rather than writing triggers for the right property or event, the control consumer simply needs to fill in the appropriate state animations—a task that gets even easier with Expression Blend.

Starting the FlipPanel Class

Stripped down to its bare bones, the FlipPanel is surprisingly simple. It consists of two content regions that the user can fill with a single element (most likely, a layout container that contains an assortment of elements). Technically, that means the FlipPanel isn't a true panel, because it doesn't use layout logic to organize a group of child elements. However, this isn't likely to pose a problem, because the structure of the FlipPanel is clear and intuitive. The FlipPanel also includes a flip button that lets the user switch between the two different content regions.

Although you can create a custom control by deriving from a control class like ContentControl or Panel, the FlipPanel derives directly from the base Control class. If you don't need the functionality of a specialized control class, this is the best starting point. You shouldn't derive from the simpler FrameworkElement class unless you want to create an element without the standard control and template infrastructure:

```
public class FlipPanel : Control
{...}
```

The first order of business is to create the properties for the FlipPanel. As with almost all the properties in a WPF element, you should use dependency properties. Here's how FlipPanel defines the FrontContent property that holds the element that's displayed on the front surface:

```
public static readonly DependencyProperty FrontContentProperty =
  DependencyProperty.Register("FrontContent", typeof(object),
  typeof(FlipPanel), null);
```

Next, you need to add a traditional .NET property procedure that calls the base GetValue() and SetValue() methods to change the dependency property. Here's the property procedure implementation for the FrontContent property:

```
public object FrontContent
{
    get
    {
        return base.GetValue(FrontContentProperty);
    }
    set
    {
        base.SetValue(FrontContentProperty, value);
    }
}
```

The BackContent property is virtually identical:

```
public static readonly DependencyProperty BackContentProperty =
  DependencyProperty.Register("BackContent", typeof(object),
  typeof(FlipPanel), null);

public object BackContent
{
    get
    {
        return base.GetValue(BackContentProperty);
    }
    set
    {
        base.SetValue(BackContentProperty, value);
    }
}
```

You need to add just one more essential property: IsFlipped. This Boolean property keeps track of the current state of the FlipPanel (forward-facing or backward-facing) and lets the control consumer flip it programmatically:

```
public static readonly DependencyProperty IsFlippedProperty =
  DependencyProperty.Register("IsFlipped", typeof(bool), typeof(FlipPanel), null);

public bool IsFlipped
{
    get
    {
```

```
        return (bool)base.GetValue(IsFlippedProperty);
    }
    set
    {
        base.SetValue(IsFlippedProperty, value);
        ChangeVisualState(true);
    }
}
```

The IsFlipped property setter calls a custom method called ChangeVisualState(). This method makes sure the display is updated to match the current flip state (forward-facing or backward-facing). You'll consider the code that takes care of this task a bit later.

The FlipPanel doesn't need many more properties, because it inherits virtually everything it needs from the Control class. One exception is the CornerRadius property. Although the Control class includes BorderBrush and BorderThickness properties, which you can use to draw a border around the FlipPanel, it lacks the CornerRadius property for rounding square edges into a gentler curve, as the Border element does. Implementing the same effect in the FlipPanel is easy, provided you add the CornerRadius dependency property and use it to configure a Border element in the FlipPanel's default control template:

```
public static readonly DependencyProperty CornerRadiusProperty =
  DependencyProperty.Register("CornerRadius", typeof(CornerRadius),
  typeof(FlipPanel), null);

public CornerRadius CornerRadius
{
    get { return (CornerRadius)GetValue(CornerRadiusProperty); }
    set { SetValue(CornerRadiusProperty, value); }
}
```

You also need to add a style that applies the default template for the FlipPanel. You place this style in the generic.xaml resource dictionary, as you did when developing the ColorPicker. Here's the basic skeleton you need:

```
<Style TargetType="{x:Type local:FlipPanel}">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="local:FlipPanel">
        ...
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

There's one last detail. To tell your control to pick up the default style from the generic.xaml file, you need to call the DefaultStyleKeyProperty.OverrideMetadata() method in the FlipPanel's static constructor:

```
DefaultStyleKeyProperty.OverrideMetadata(typeof(FlipPanel),
  new FrameworkPropertyMetadata(typeof(FlipPanel)));
```

Choosing Parts and States

Now that you have the basic structure in place, you're ready to identify the parts and states that you'll use in the control template.

Clearly, the FlipPanel requires two states:

- **Normal.** This storyboard ensures that only the front content is visible. The back content is flipped, faded, or otherwise shuffled out of view.
- **Flipped.** This storyboard ensures that only the back content is visible. The front content is animated out of the way.

In addition, you need two parts:

- **FlipButton.** This is the button that, when clicked, changes the view from the front to the back (or vice versa). The FlipPanel provides this service by handling this button's events.
- **FlipButtonAlternate.** This is an optional element that works in the same way as the FlipButton. Its inclusion allows the control consumer to use two different approaches in a custom control template. One option is to use a single flip button outside the flippable content region. The other option is to place a separate flip button on both sides of the panel, in the flippable region.

■ **Note** Keen eyes will notice a confusing design choice here. Unlike the custom ColorPicker, the named parts in the FlipPanel don't use the PART_ naming syntax (as in PART_FlipButton). That's because the PART_ naming system was introduced before the visual state model. With the visual state model, the conventions have changed to favor simpler names, although this is still an emerging standard, and it could change in the future. In the meantime, your custom controls should be fine as long as they use the TemplatePart attribute to point out all the named parts.

You could also add parts for the front content and back content regions. However, the FlipPanel control doesn't need to manipulate these regions directly, as long as the template includes an animation that hides or shows them at the appropriate time. (Another option is to define these parts so you can explicitly change their visibility in code. That way, the panel can still change between the front and back content region even if no animations are defined, by hiding one section and showing the other. For simplicity's sake, the FlipPanel doesn't go to these lengths.)

To advertise the fact that the FlipPanel uses these parts and states, you should apply the TemplatePart attribute to your control class, as shown here:

```
[TemplateVisualState(Name = "Normal", GroupName="ViewStates")]
[TemplateVisualState(Name = "Flipped", GroupName = "ViewStates")]
[TemplatePart(Name = "FlipButton", Type = typeof(ToggleButton))]
[TemplatePart(Name = "FlipButtonAlternate", Type = typeof(ToggleButton))]
public class FlipPanel : Control
{ ... }
```

The FlipButton and FlipButtonAlternate parts are restricted—each one can only be a ToggleButton or an instance of a ToggleButton-derived class. (As you may remember from Chapter 6, the ToggleButton is a clickable button that can be in one of two states. In the case of the FlipPanel control, the ToggleButton states correspond to normal front-forward view or a flipped back-forward view.)

■ **Tip** To ensure the best, most flexible template support, use the least-specialized element type that you can. For example, it's better to use FrameworkElement than ContentControl, unless you need some property or behavior that ContentControl provides.

The Default Control Template

Now, you can slot these pieces into the default control template. The root element is a two-row Grid that holds the content area (in the top row) and the flip button (in the bottom row). The content area is filled with two overlapping Border elements, representing the front and back content, but only one of the two is ever shown at a time.

To fill in the front and back content regions, the FlipPanel uses the ContentPresenter. This technique is virtually the same as in the custom button example, except you need two ContentPresenter elements, one for each side of the FlipPanel. The FlipPanel also includes a separate Border element wrapping each ContentPresenter. This lets the control consumer outline the flippable content region by setting a few straightforward properties on the FlipPanel (BorderBrush, BorderThickness, Background, and CornerRadius), rather than being forced to add a border by hand.

Here's the basic skeleton for the default control template:

```
<ControlTemplate TargetType="{x:Type local:FlipPanel}">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"></RowDefinition>
      <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>

    <!-- This is the front content. -->
    <Border BorderBrush="{TemplateBinding BorderBrush}"
     BorderThickness="{TemplateBinding BorderThickness}"
     CornerRadius="{TemplateBinding CornerRadius}"
     Background="{TemplateBinding Background}">
      <ContentPresenter Content="{TemplateBinding FrontContent}">
      </ContentPresenter>
    </Border>

    <!-- This is the back content. -->
    <Border BorderBrush="{TemplateBinding BorderBrush}"
     BorderThickness="{TemplateBinding BorderThickness}"
     CornerRadius="{TemplateBinding CornerRadius}"
     Background="{TemplateBinding Background}">
      <ContentPresenter Content="{TemplateBinding BackContent}">
      </ContentPresenter>
    </Border>
```