CHAPTER 18

Custom Elements

In previous Windows development frameworks, custom controls played a central role. But in WPF, the emphasis has shifted. Custom controls are still a useful way to build custom widgets that you can share between applications, but they're no longer a requirement when you want to enhance and customize core controls. (To understand how remarkable this change is, it helps to point out that this book's predecessor, *Pro .NET 2.0 Windows Forms and Custom Controls* in *C#*, had nine complete chapters about custom controls and additional examples in other chapters. But in this book, you've made it to Chapter 18 without a single custom control sighting!)

WPF de-emphasizes custom controls because of its support for styles, content controls, and templates. These features give every developer several ways to refine and extend standard controls without deriving a new control class. Here are your possibilities:

> *Styles*: You can use a style to painlessly reuse a combination of control properties. You can even apply effects by using triggers.
>
> *Content controls*: Any control that derives from ContentControl supports nested content. Using content controls, you can quickly create compound controls that aggregate other elements. (For example, you can transform a button into an image button or a list box into an image list.)
>
> *Control templates*: All WPF controls are *lookless*, which means they have hardwired functionality but their appearance is defined separately through the control template. Replace the default template with something new, and you can revamp basic controls such as buttons, check boxes, radio buttons, and even windows.
>
> *Data templates*: All ItemsControl-derived classes support data templates, which allow you to create a rich list representation of some type of data object. Using the right data template, you can display each item by using a combination of text, images, and even editable controls, all in a layout container of your choosing.

If possible, you should pursue these avenues before you decide to create a custom control or another type of custom element. That's because these solutions are simpler, easier to implement, and often easier to reuse.

So, when *should* you create a custom element? Custom elements aren't the best choice when you want to fine-tune the appearance of an element, but they do make sense when you want to change its underlying functionality. For example, there's a reason that WPF has separate classes for the TextBox and

PasswordBox classes. They handle key presses differently, store their data internally in a different way, interact with other components such as the clipboard differently, and so on. Similarly, if you want to design a control that has its own distinct set of properties, methods, and events, you'll need to build it yourself.

In this chapter, you'll learn how to create custom elements and how to make them into first-class WPF citizens. That means you'll outfit them with dependency properties and routed events to get support for essential WPF services such as data binding, styles, and animation. You'll also learn how to create a *lookless* control—a template-driven control that allows the control consumer to supply different visuals for greater flexibility.

Understanding Custom Elements in WPF

Although you can code a custom element in any WPF project, you'll usually want to place custom elements in a dedicated class library (DLL) assembly. That way, you can share your work with multiple WPF applications.

To make sure you have the right assembly references and namespace imports, you should choose the Custom Control Library (WPF) project type when you create your application in Visual Studio. Inside your class library, you can create as many or as few controls as you like.

■ **Tip** As with all class library development, it's often a good practice to place both your class library and the application that uses your class library in the same Visual Studio solution. That way, you can easily modify and debug both pieces at once.

The first step in creating a custom control is choosing the right base class to inherit from. Table 18-1 lists some commonly used classes for creating custom controls, and Figure 18-1 shows where they fit into the element hierarchy.

Table 18-1. *Base Classes for Creating a Custom Element*

Name	Description
FrameworkElement	This is the lowest level you'll typically use when creating a custom element. Usually, you'll take this approach only if you want to draw your content from scratch by overriding OnRender()and using the System.Windows.Media. DrawingContext. It's similar to the approach you saw in Chapter 14, where a user interface was constructed by using Visual objects. The FrameworkElement class provides the basic set of properties and events for elements that aren't intended to interact with the user.
Control	This is the most common starting point when building a control from scratch. It's the base class for all user-interactive widgets. The Control class adds properties for setting the background and foreground, as well as the font and alignment of content. The control class also places itself into the tab order (through the IsTabStop property) and introduces the notion of double-clicking (through the MouseDoubleClick and PreviewMouseDoubleClick events). But most important, the Control class defines the Template property that allows its appearance to be swapped out with a customized element tree for endless flexibility.

ContentControl	This is the base class for controls that can display a single piece of arbitrary content. That content can be an element or a custom object that's used in conjunction with a template. (The content is set through the Content property, and an optional template can be provided in the ContentTemplate property.) Many controls wrap a specific, limited type of content (such as a string of text in a text box). Because these controls don't support all elements, they shouldn't be defined as content controls.
UserControl	This is a content control that can be configured using a design-time surface. Although a user control isn't that different from an ordinary content control, it's typically used when you want to quickly reuse an unchanging block of user interface in more than one window (rather than create a true stand-alone control that can be transported from one application to another).
ItemsControl or Selector	ItemsControl is the base class for controls that wrap a list of items but don't support selection, while Selector is the more specialized base class for controls that do support selection. These classes aren't often used to create custom controls, because the data-templating features of the ListBox, ListView, and TreeView provide a great deal of flexibility.
Panel	This is the base class for controls with layout logic. A layout control can hold multiple children and arranges them according to specific layout semantics. Often panels include attached properties that can be set on the children to configure how the children are arranged.
Decorator	This is the base class for elements that wrap another element and provide a graphical effect or specific feature. Two prominent examples are the Border, which draws a line around an element, and the Viewbox, which scales its content dynamically using a transform. Other decorators include the chrome classes used to give the familiar border and background to common controls such as the button.
A specific control class	If you want to introduce a refinement to an existing control, you can derive directly from that control. For example, you can create a TextBox with built-in validation logic. However, before you take this step, consider whether you could accomplish the same thing by using event-handling code or a separate component. Both approaches allow you to decouple your logic from the control and reuse it with other controls.

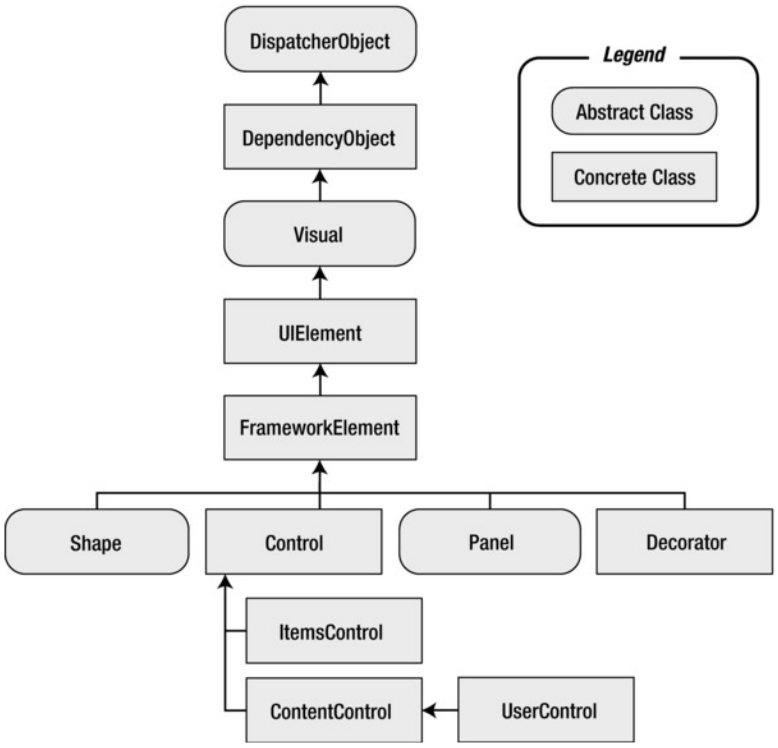


Figure 18-1. *Element and control base classes*

■ **Note** Although you can create a custom element that isn't a control, most custom elements you create in WPF will be controls—that is to say they'll be able to receive focus, and they'll interact with the user's key presses and mouse actions. For that reason, the terms *custom elements* and *custom controls* are sometimes used interchangeably in WPF development.

In this chapter, you'll see a user control, a lookless color picker that derives directly from the Control class, a lookless FlipPanel that uses visual states, a custom layout panel, and a custom-drawn element that derives from FrameworkElement and overrides OnRender(). Many of the examples are quite lengthy. Although you'll walk through almost all of the code in this chapter, you'll probably want to follow up by downloading the samples and playing with the custom controls yourself.

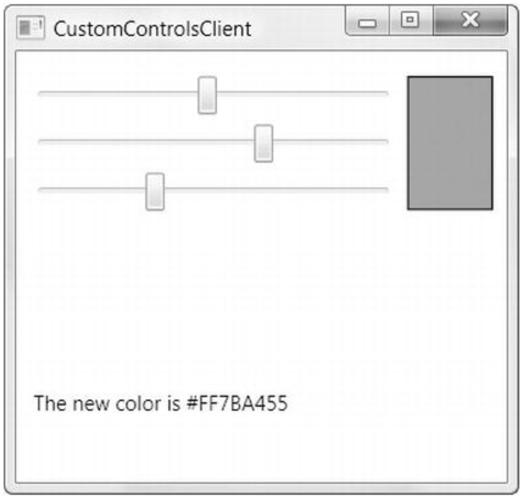
Building a Basic User Control

A good way to get started with custom controls is to take a crack at creating a straightforward user control. In this section, we'll begin by creating a basic color picker. Later, you'll see how to refactor this control into a more capable template-based control.

Creating a basic color picker is easy. However, creating a custom color picker is still a worthy exercise. Not only does it demonstrate a variety of important control-building concepts, but it also gives you a practical piece of functionality.

You could create a custom dialog box for your color picker. But if you want to create a color picker that you can integrate into different windows, a custom control is a far better choice. The most straightforward type of custom control is a *user control*, which allows you to assemble a combination of elements in the same way as when you design a window or page. Because the color picker appears to be little more than a fairly straightforward grouping of existing controls with added functionality, a user control seems like a perfect choice.

A typical color picker allows a user to select a color by clicking somewhere in a color gradient or specifying individual red, green, and blue components. Figure 18-2 shows the basic color picker you'll create in this section (at the top of the window). It consists of three Slider controls for adjusting color components, along with a Rectangle that shows a preview of the selected color.



The new color is #FF7BA455

Figure 18-2. *A color picker user control*

■ **Note** The user control approach has one significant flaw—it limits your ability to customize the appearance of your color picker to suit different windows, applications, and uses. Fortunately, it's not much harder to step up to a more template-based control, as you'll see a bit later.

Defining Dependency Properties

The first step in creating the color picker is to add a user control to your custom control library project. When you do, Visual Studio creates a XAML markup file and a corresponding custom class to hold your initialization and event-handling code. This is the same experience as when you create a new window or page—the only difference is that the top-level container is the UserControl class.

```
public partial class ColorPicker : System.Windows.Controls.UserControl
{ ... }
```

The easiest starting point is to design the public interface that the user control exposes to the outside world. In other words, it's time to create the properties, methods, and events that the control consumer (the application that uses the control) will rely on to interact with the color picker.

The most fundamental detail is the Color property—after all, the color picker is nothing more than a specialized tool for displaying and choosing a color value. To support WPF features such as data binding, styles, and animation, writeable control properties are almost always dependency properties.

As you learned in Chapter 4, the first step to creating a dependency property is to define a static field for it, with the word *Property* added to the end of your property name:

```
public static DependencyProperty ColorProperty;
```

The Color property will allow the control consumer to set or retrieve the color value programmatically. However, the sliders in the color picker also allow the user to modify one aspect of the current color. To implement this design, you could use event handlers that respond when a slider value is changed and update the Color property accordingly. But it's cleaner to wire the sliders up by using data binding. To make this possible, you need to define each of the color components as a separate dependency property:

```
public static DependencyProperty RedProperty;
public static DependencyProperty GreenProperty;
public static DependencyProperty BlueProperty;
```

Although the Color property will store a System.Windows.Media.Color object, the Red, Green, and Blue properties will store individual byte values that represent each color component. (You could also add a slider and a property for managing the alpha value, which allows you to create a partially transparent color, but this example doesn't add this detail.)

Defining the static fields for your properties is just the first step. You also need a static constructor in your user control that registers them, specifying the property name, the data type, and the control class that owns the property. As you learned in Chapter 4, this is the point where you can opt in to specific property features (such as value inheritance) by passing a FrameworkPropertyMetadata object with the right flags set. It's also the point where you can attach callbacks for validation, value coercion, and property change notifications.

In the color picker, you have just one consideration—you need to attach callbacks that respond when the various properties are changed. That's because the Red, Green, and Blue properties are really a different representation of the Color property, and if one property changes, you need to make sure the others stay synchronized.

Here's the static constructor code that registers the four dependency properties of the color picker:

```
static ColorPicker()
{
    ColorProperty =  DependencyProperty.Register(
      "Color", typeof(Color), typeof(ColorPicker),
      new FrameworkPropertyMetadata(Colors.Black,
        new PropertyChangedCallback(OnColorChanged)));

     RedProperty = DependencyProperty.Register(
      "Red", typeof(byte), typeof(ColorPicker),
      new FrameworkPropertyMetadata(
        new PropertyChangedCallback(OnColorRGBChanged)));

     GreenProperty = DependencyProperty.Register(
       "Green", typeof(byte), typeof(ColorPicker),
       new FrameworkPropertyMetadata(
         new PropertyChangedCallback(OnColorRGBChanged)));
     BlueProperty = DependencyProperty.Register(
       "Blue", typeof(byte), typeof(ColorPicker),
      new FrameworkPropertyMetadata(
        new PropertyChangedCallback(OnColorRGBChanged)));
}
```

Now that you have your dependency properties defined, you can add standard property wrappers that make them easier to access and usable in XAML:

```
public Color Color
{
    get { return (Color)GetValue(ColorProperty); }
    set { SetValue(ColorProperty, value); }
}

public byte Red
{
    get { return (byte)GetValue(RedProperty); }
    set { SetValue(RedProperty, value); }
}

public byte Green
{
    get { return (byte)GetValue(GreenProperty); }
    set { SetValue(GreenProperty, value); }
}

public byte Blue
{
    get { return (byte)GetValue(BlueProperty); }
    set { SetValue(BlueProperty, value); }
}
```

Remember, the property wrappers shouldn't contain any logic, because properties may be set and retrieved directly by using the SetValue() and GetValue() methods of the base DependencyObject class. For example, the property synchronization logic in this example is implemented by using callbacks that fire when the property changes through the property wrapper or a direct SetValue() call.

The property change callbacks are responsible for keeping the Color property consistent with the Red, Green, and Blue properties. Whenever the Red, Green, or Blue property is changed, the Color property is adjusted accordingly:

```
private static void OnColorRGBChanged(DependencyObject sender,
  DependencyPropertyChangedEventArgs e)
{
    ColorPicker colorPicker = (ColorPicker)sender;
    Color color = colorPicker.Color;

    if (e.Property == RedProperty)
        color.R = (byte)e.NewValue;
    else if (e.Property == GreenProperty)
        color.G = (byte)e.NewValue;
    else if (e.Property == BlueProperty)
        color.B = (byte)e.NewValue;

    colorPicker.Color = color;
}
```

and when the Color property is set, the Red, Green, and Blue values are also updated:

```
private static void OnColorChanged(DependencyObject sender,
  DependencyPropertyChangedEventArgs e)
{
    Color newColor = (Color)e.NewValue;
    Color oldColor = (Color)e.OldValue;

    ColorPicker colorPicker = (ColorPicker)sender;
    colorPicker.Red = newColor.R;
    colorPicker.Green = newColor.G;
    colorPicker.Blue = newColor.B;
}
```

Despite its appearances, this code won't cause an infinite series of calls as each property tries to change the other. That's because WPF doesn't allow reentrancy in the property change callbacks. For example, if you change the Color property, the OnColorChanged() method will be triggered. The OnColorChanged() method will modify the Red, Green, and Blue properties, triggering the OnColorRGBChanged() callback three times (once for each property). However, the OnColorRBGChanged() will not trigger the OnColorChanged() method again.

■ **Tip** It might occur to you to use the coercion callbacks discussed in Chapter 4 to deal with the color properties. However, this approach isn't appropriate. Property coercion callbacks are designed for properties that are interrelated and may override or influence one another. They don't make sense for properties that expose the same data in different ways. If you used property coercion in this example, it would be possible to set different values in the Red, Green, and Blue properties and have that color information *override* the Color property. The behavior you really want is to set the Red, Green, and Blue properties and use that color information to permanently *change* the value of the Color property.

Defining Routed Events

You might also want to add routed events that can be used to notify the control consumer when something happens. In the color picker example, it's useful to have an event that fires when the color is changed. Although you could define this event as an ordinary .NET event, using a routed event allows you to provide event bubbling and tunneling, so the event can be handled in a higher-level parent, such as the containing window.

As with the dependency properties, the first step to defining a routed event is to create a static property for it, with the word *Event* added to the end of the event name:

```
public static readonly RoutedEvent ColorChangedEvent;
```

You can then register the event in the static constructor. At this point, you specify the event name, the routing strategy, the signature, and the owning class:

```
ColorChangedEvent = EventManager.RegisterRoutedEvent(
  "ColorChanged", RoutingStrategy.Bubble,
  typeof(RoutedPropertyChangedEventHandler<Color>), typeof(ColorPicker));
```

Rather than going to the work of creating a new delegate for your event signature, you can sometimes reuse existing delegates. The two useful delegates are RoutedEventHandler (for a routed event that doesn't pass along any extra information) and RoutedPropertyChangedEventHandler (for a routed event that provides the old and new values after a property has been changed). The RoutedPropertyChangedEventHandler, which is used in the previous example, is a generic delegate that's parameterized by type. As a result, you can use it with any property data type without sacrificing type safety.

After you've defined and registered the event, you need to create a standard .NET event wrapper that exposes your event. This event wrapper can be used to attach (and remove) event listeners:

```
public event RoutedPropertyChangedEventHandler<Color> ColorChanged
{
    add { AddHandler(ColorChangedEvent, value); }
    remove { RemoveHandler(ColorChangedEvent, value); }
}
```

The final detail is the code that raises the event at the appropriate time. This code must call the RaiseEvent() method that's inherited from the base DependencyObject class.

In the color picker example, you simply need to add these lines of code to the end of the OnColorChanged() method:

```
Color oldColor = (Color)e.OldValue;
RoutedPropertyChangedEventArgs<Color> args =
  new RoutedPropertyChangedEventArgs<Color>(oldColor, newColor);
args.RoutedEvent = ColorPicker.ColorChangedEvent;

colorPicker.RaiseEvent(args);
```

Remember, the OnColorChanged() callback is triggered whenever the Color property is modified, either directly or by modifying the Red, Green, and Blue color components.

Adding Markup

Now that your user control's public interface is in place, all you need is the markup that creates the control's appearance. In this case, a basic Grid is all that's needed to bring together the three Slider controls and the Rectangle with the color preview. The trick is the data-binding expressions that tie these controls to the appropriate properties, with no event-handling code required.

All in all, four data-binding expressions are at work in the color picker. The three sliders are bound to the Red, Green, and Blue properties and are allowed to range from 0 to 255 (the acceptable values for a byte). The Rectangle.Fill property is set using a SolidColorBrush, and the Color property of that brush is bound to the Color property of the user control.

Here's the complete markup:

```
<UserControl x:Class="CustomControls.ColorPicker"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Name="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 Name="sliderRed" Minimum="0" Maximum="255"
     Value="{Binding ElementName=colorPicker,Path=Red}"></Slider>
    <Slider Grid.Row="1" Name="sliderGreen" Minimum="0" Maximum="255"
     Value="{Binding ElementName=colorPicker,Path=Green}"></Slider>
    <Slider Grid.Row="2" Name="sliderBlue" Minimum="0" Maximum="255"
     Value="{Binding ElementName=colorPicker,Path=Blue}"></Slider>

    <Rectangle Grid.Column="1" Grid.RowSpan="3"
     Width="50" Stroke="Black" StrokeThickness="1">
      <Rectangle.Fill>
        <SolidColorBrush Color="{Binding ElementName=colorPicker,Path=Color}">
        </SolidColorBrush>
      </Rectangle.Fill>
    </Rectangle>

  </Grid>
</UserControl>
```

The markup for a user control plays the same role as the control template for a lookless control. If you want to make some of the details in your markup configurable, you can use binding expressions that link them to control properties. For example, currently the Rectangle's width is hard-coded at 50 units. However, you could replace this detail with a data-binding expression that pulls the value from a dependency property in your user control. That way, the control consumer could modify that property to choose a different width. Similarly, you could make the stroke color and thickness variable. However, if you want to make a control with real flexibility, you're much better off to create a lookless control and define the markup in a template, as described later in this chapter.

Occasionally, you might choose to use binding expressions to repurpose one of the core properties that's already defined in your control. For example, the UserControl class uses its Padding property to add space between the outer edge and the inner content that you define. (This detail is implemented through the control template for the UserControl.) However, you could also use the Padding property to set the spacing around each slider, as shown here:

```
<Slider Name="sliderRed" Minimum="0" Maximum="255"
  Margin="{Binding ElementName=colorPicker,Path=Padding}"
  Value="{Binding ElementName=colorPicker,Path=Red}"></Slider>
```

Similarly, you could grab the border settings for the Rectangle from the BorderThickness and BorderBrush properties of the UserControl. Once again, this is a shortcut that may make perfect sense for creating simple controls but can be improved by introducing additional properties (for example, SliderMargin, PreviewBorderBrush, and PreviewBorderThickness) or creating a full-fledged template-based control.

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.

Supporting Commands

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 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 by 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.

Taking 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 by 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)));
```

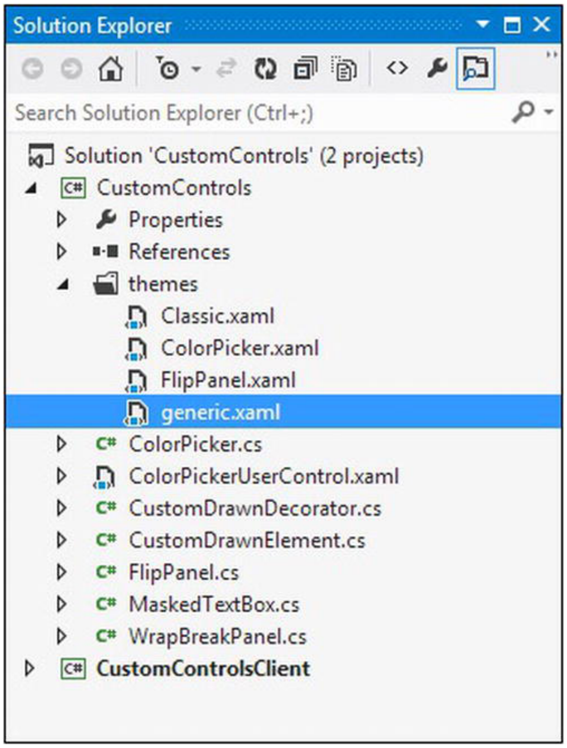
You could supply a different type if you want to use the template of another control class, but you'll almost always create a specific style for each one of your custom controls.

Refactoring the Color Picker Markup

After you've 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 ä New Folder.
2. Name the new folder Themes.
3. Right-click the Themes folder, and choose Add ä 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.



```
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'CustomControls' (2 projects)
◢ CustomControls
    ▷ Properties
    ▷ References
    ◢ themes
        Classic.xaml
        ColorPicker.xaml
        FlipPanel.xaml
        generic.xaml
    ▷ ColorPicker.cs
    ▷ ColorPickerUserControl.xaml
    ▷ CustomDrawnDecorator.cs
    ▷ CustomDrawnElement.cs
    ▷ FlipPanel.cs
    ▷ MaskedTextBox.cs
    ▷ WrapBreakPanel.cs
▷ CustomControlsClient
```

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

THEME-SPECIFIC STYLES AND GENERIC.XAML

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 a legacy WPF feature: *Windows theme support*.

The original goal of Windows theme support was to let developers create customized versions of their controls to match different Windows themes. This goal made the most sense on old Windows XP computers, which used themes to control the overall color scheme of Windows applications. When Windows Vista was released, it introduced the Aero theme, which effectively replaced the old theme choices. The versions of Windows that followed haven't changed that state of affairs, and so the Windows theming feature in WPF (which was never much used) is now universally ignored.

The bottom line this: WPF developers creating applications today always use a generic.xaml file to set their default control styles. The name of the generic.xaml file (and the Themes folder in which it's placed) is a holdover from the past.

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 (such as 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 (such as 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 a 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.

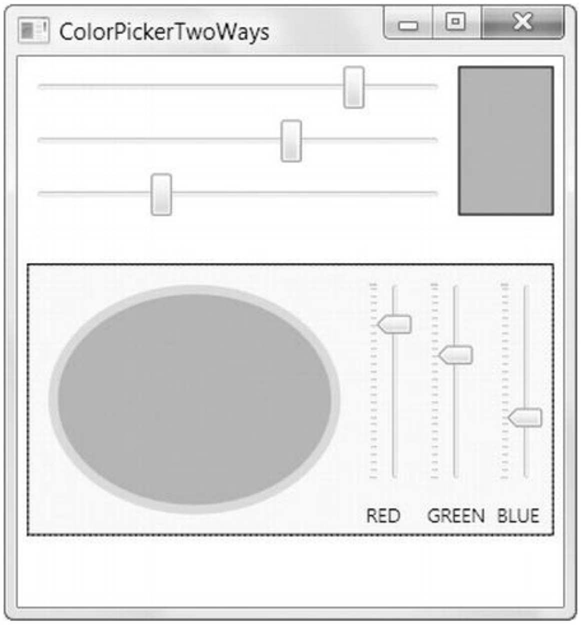


Figure 18-4. *A color picker custom control with two different templates*

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
{ ... }
```

FINDING A CONTROL'S DEFAULT STYLE

Every control has a default style. 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 (such as 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 whether it has focus, whether the mouse is on top, whether 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 uses 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.

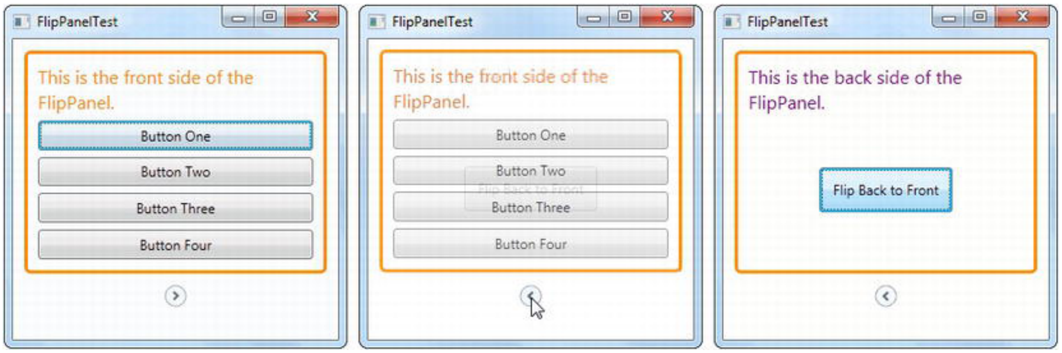


Figure 18-5. *Flipping the FlipPanel*

You can perform the flipping programmatically (by setting a property named IsFlipped), or the user can flip the panel by 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 content regions.

Although you can create a custom control by deriving from a control class such as 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 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 be only 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>

    <!-- This the flip button. -->
    <ToggleButton Grid.Row="1" x:Name="FlipButton" Margin="0,10,0,0">
    </ToggleButton>

  </Grid>
</ControlTemplate>
```

When you create a default control template, it's best to avoid hard-coding details that the control consumer may want to customize. Instead, you need to use template-binding expressions. In this example, you set several properties by using template-binding expressions: BorderBrush, BorderThickness, CornerRadius, Background, FrontContent, and BackContent. To set the default value for these properties (and thereby ensure that you get the right visual even if the control consumer doesn't set them), you must add additional setters to your control's default style.

The Flip Button

The control template shown in the previous example includes a ToggleButton. However, it uses the ToggleButton's default appearance, which makes the ToggleButton look like an ordinary button, complete with the traditional shaded background. This isn't suitable for the FlipPanel.

Although you can place any content you want inside the ToggleButton, the FlipPanel requires a bit more. It needs to do away with the standard background and change the appearance of the elements inside depending on the state of the ToggleButton. As you saw earlier in Figure 18-5, the ToggleButton points the way the content will be flipped (right initially, when the front faces forward, and left when the back faces forward). This makes the purpose of the button clearer.

To create this effect, you need to design a custom control template for the ToggleButton. This control template can include the shape elements that draw the arrow you need. In this example, the ToggleButton is drawn using an Ellipse element for the circle and a Path element for the arrow, both of which are placed in a single-cell Grid:

```
<ToggleButton Grid.Row="1" x:Name="FlipButton" RenderTransformOrigin="0.5,0.5"
 Margin="0,10,0,0" Width="19" Height="19">
  <ToggleButton.Template>
    <ControlTemplate>
      <Grid>
        <Ellipse Stroke="#FFA9A9A9" Fill="AliceBlue"></Ellipse>
        <Path Data="M1,1.5L4.5,5 8,1.5" Stroke="#FF666666" StrokeThickness="2"
         HorizontalAlignment="Center" VerticalAlignment="Center"></Path>
      </Grid>
    </ControlTemplate>
  </ToggleButton.Template>
</ToggleButton>
```

The ToggleButton needs one more detail—a RotateTransform that turns the arrow away from one side to point at the other. This RotateTransform will be used when you create the state animations:

```
<ToggleButton.RenderTransform>
  <RotateTransform x:Name="FlipButtonTransform" Angle="-90"></RotateTransform>
</ToggleButton.RenderTransform>
```

Defining the State Animations

The state animations are the most interesting part of the control template. They're the ingredients that provide the flipping behavior. They're also the details that are most likely to be changed if a developer creates a custom template for the FlipPanel.

To define state groups, you must add the VisualStateManager.VisualStateGroups element in the root element of your control template, as shown here:

```
<ControlTemplate TargetType="{x:Type local:FlipPanel}">
  <Grid>
    <VisualStateManager.VisualStateGroups>
      ...
    </VisualStateManager.VisualStateGroups>

    ...
  </Grid>
</ControlTemplate>
```

■ **Note** To add the VisualStateManager element to a template, your template must use a layout panel. This layout panel holds both the visuals for your control and the VisualStateManager, which is invisible. The VisualStateManager defines storyboards with the animations that the control can use at the appropriate time to alter its appearance.

Inside the VisualStateGroups element, you can create the state groups by using appropriately named VisualStateGroup elements. Inside each VisualStateGroup, you add a VisualState element for each visual state. In the case of the FlipPanel, there is one group that contains two visual states:

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="ViewStates">
    <VisualState x:Name="Normal">
      ...
    </VisualState>
  </VisualStateGroup>

  <VisualStateGroup x:Name="FocusStates">
    <VisualState x:Name="Flipped">
      ...
    </VisualState>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Each state corresponds to a storyboard with one or more animations. If these storyboards exist, they're triggered at the appropriate times. (If they don't, the control should degrade gracefully, without raising an error.)

In the default control template, the animations use a simple fade to change from one content region to the other and use a rotation to flip the ToggleButton arrow around to point in the other direction. Here's the markup that takes care of both tasks:

```
<VisualState x:Name="Normal">
  <Storyboard>
    <DoubleAnimation Storyboard.TargetName="BackContent"
     Storyboard.TargetProperty="Opacity" To="0" Duration="0" ></DoubleAnimation>
  </Storyboard>
</VisualState>

<VisualState x:Name="Flipped">
  <Storyboard>
    <DoubleAnimation Storyboard.TargetName="FlipButtonTransform"
     Storyboard.TargetProperty="Angle" To="90" Duration="0"></DoubleAnimation>
    <DoubleAnimation Storyboard.TargetName="FrontContent"
     Storyboard.TargetProperty="Opacity" To="0" Duration="0"></DoubleAnimation>
  </Storyboard>
</VisualState>
```

You'll notice that the visual states set the animation duration to 0, which means the animation applies its effect instantaneously. This might seem a little odd—after all, don't you need a more gradual change to notice the animated effect?

In fact, this design is perfectly correct, because visual states are meant to indicate how the control looks while it's in the appropriate state. For example, a flipped panel simply shows its background content while in the flipped state. The flipping process is a *transition* that happens just before the FlipControl enters the flipped state, not part of the state itself. (This distinction between states and transitions is important, because some controls *do* have animations that run during a state. For example, think of the button example from Chapter 17 that featured the pulsing background color while the mouse hovers over it.)

Defining the State Transitions

A transition is an animation that starts from the current state and ends at the new state. One of the advantages of the transition model is that you don't need to create the storyboard for this animation. For example, if you add the markup shown here, WPF creates a 0.7-second animation to change the opacity of the FlipPanel, creating the pleasant fade effect you want:

```
<VisualStateGroup x:Name="ViewStates">
  <VisualStateGroup.Transitions>
    <VisualTransition GeneratedDuration="0:0:0.7"></VisualTransition>
  </VisualStateGroup.Transitions>

  <VisualState x:Name="Normal">
    ...
  </VisualState>

  <VisualState x:Name="Flipped">
    ...
  </VisualState>
</VisualStateGroup>
```

Transitions apply to state groups. When you define a transition, you must add it to the VisualStateGroup.Transitions collection. This example uses the simplest sort of transition: a *default transition*, which applies to all the state changes for that group.

A default transition is convenient, but it's a one-size-fits-all solution that's not always suitable. For example, you may want the FlipPanel to transition at different speeds depending on which state it's entering. To set this up, you need to define multiple transitions, and you need to set the To property to specify when the transition will come into effect.

For example, if you have these transitions

```
<VisualStateGroup.Transitions>
  <VisualTransition To="Flipped" GeneratedDuration="0:0:0.5" />
  <VisualTransition To="Normal" GeneratedDuration="0:0:0.1" />
</VisualStateGroup.Transitions>
```

the FlipPanel will switch to the Flipped state in 0.5 seconds, and it will enter the Normal state in 0.1 seconds.

This example shows transitions that apply when entering specific states, but you can also use the From property to create a transition that applies when leaving a state, and you can use the To and From properties in conjunction to create even more specific transitions that apply only when moving between two specific states. When applying transitions, WPF looks through the collection of transitions to find the most specific one that applies, and it uses only that one.

For even more control, you can create custom transition animations that take the place of the automatically generated transitions WPF would normally use. You may create a custom transition for several reasons. Here are some examples: to control the pace of the animation with a more sophisticated animation, to use an animation easing, to run several animations in succession, or to play a sound at the same time as an animation.

To define a custom transition, you place a storyboard with one or more animations inside the VisualTransition element. In the FlipPanel example, you can use custom transitions to make sure the ToggleButton arrow rotates itself quickly, while the fade takes place more gradually.

```
<VisualStateGroup.Transitions>
  <VisualTransition GeneratedDuration="0:0:0.7" To="Flipped">
    <Storyboard>
      <DoubleAnimation Storyboard.TargetName="FlipButtonTransform"
       Storyboard.TargetProperty="Angle" To="90"
       Duration="0:0:0.2"></DoubleAnimation>
    </Storyboard>
  </VisualTransition>
  <VisualTransition GeneratedDuration="0:0:0.7" To="Normal">
    <Storyboard>
      <DoubleAnimation Storyboard.TargetName="FlipButtonTransform"
       Storyboard.TargetProperty="Angle" To="-90"
       Duration="0:0:0.2"></DoubleAnimation>
    </Storyboard>
  </VisualTransition>
</VisualStateGroup.Transitions>
```

■ **Note** When you use a custom transition, you must still set the VisualTransition.GeneratedDuration property to match the duration of your animation. Without this detail, the VisualStateManager can't use your transition, and it will apply the new state immediately. (The actual time value you use still has no effect on your custom transition, because it applies only to automatically generated animations.)

Unfortunately, many controls will require custom transitions, and writing them is tedious. You still need to keep the zero-length state animations, which also creates some unavoidable duplication of details between your visual states and your transitions.

Wiring Up the Elements

Now that you've polished off a respectable control template, you need to fill in the plumbing in the FlipPanel control to make it work.

The trick is the OnApplyTemplate(), which you also used to set bindings in the ColorPicker. The OnApplyTemplate() method for the FlipPanel retrieves the ToggleButton for the FlipButton and FlipButtonAlternate parts and attaches event handlers to each so it can react when the user clicks to flip the control. Finally, the OnApplyTemplate() method ends by calling a custom method named ChangeVisualState(), which ensures that the control's visuals match its current state:

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

    // Wire up the ToggleButton.Click event.
    ToggleButton flipButton = base.GetTemplateChild("FlipButton") as ToggleButton;
    if (flipButton != null) flipButton.Click += flipButton_Click;

    // Allow for two flip buttons if needed (one for each side of the panel).
    ToggleButton flipButtonAlternate =
      base.GetTemplateChild("FlipButtonAlternate") as ToggleButton;
    if (flipButtonAlternate != null) flipButtonAlternate.Click += flipButton_Click;

    // Make sure the visuals match the current state.
    this.ChangeVisualState(false);
}
```

■ **Tip** When calling GetTemplateChild(), you need to indicate the string name of the element you want. To avoid possible errors, you can declare this string as a constant in your control. You can then use that constant in the TemplatePart attribute and when calling GetTemplateChild().

Here's the very simple event handler that allows the user to click the ToggleButton and flip the panel:

```
private void flipButton_Click(object sender, RoutedEventArgs e)
{
    this.IsFlipped = !this.IsFlipped;
    ChangeVisualState(true);
}
```

Fortunately, you don't need to manually trigger the state animations. Nor do you need to create or trigger the transition animations. Instead, to change from one state to another, you call the static VisualStateManager.GoToState() method. When you do, you pass in a reference to the control object that's changing state, the name of the new state, and a Boolean value that determines whether a transition is shown. This value should be true when it's a user-initiated change (for example, when the user clicks the ToggleButton) but false when it's a property setting (for example, if the markup for your page sets the initial value of the IsFlipped property).

Dealing with all the different states a control supports can become messy. To avoid scattering GoToState() calls throughout your control code, most controls add a custom method such as the ChangeVisualState() method in the FlipPanel. This method has the responsibility of applying the correct state in each state group. The code inside uses one `if` block (or switch statement) to apply the current state in each state group. This approach works because it's completely acceptable to call GoToState() with the name of the current state. In this situation, when the current state and the requested state are the same, nothing happens.

Here's the code for the FlipPanel's version of the ChangeVisualState() method:

```
private void ChangeVisualState(bool useTransitions)
{
    if (!IsFlipped)
    {
        VisualStateManager.GoToState(this, "Normal", useTransitions);
    }
    else
    {
        VisualStateManager.GoToState(this, "Flipped", useTransitions);
    }
}
```

Usually, you call the ChangeVisualState() method (or your equivalent) in the following places:

- After initializing the control at the end of the OnApplyTemplate() method.
- When reacting to an event that represents a state change, such as a mouse movement or a click of the ToggleButton.
- When reacting to a property change or a method that's triggered through code. (For example, the IsFlipped property setter calls ChangeVisualState() and always supplies true, thereby showing the transition animations. If you want to give the control consumer the choice of not showing the transition, you can add a Flip() method that takes the same Boolean parameter you pass to ChangeVisualState().

As written, the FlipPanel control is remarkably flexible. For example, you can use it without a ToggleButton and flip it programmatically (perhaps when the user clicks a different control). Or, you can include one or two flip buttons in the control template and allow the user to take control.

Using the FlipPanel

Now that you've completed the control template and code for the FlipPanel, you're ready to use it in an application. Assuming you've added the necessary assembly reference, you can then map an XML prefix to the namespace that holds your custom control:

```
<Window x:Class="FlipPanelTest.Page"
  xmlns:lib="clr-namespace:FlipPanelControl;assembly=FlipPanelControl" ... >
```

Next, you can add instances of the FlipPanel to your page. Here's an example that creates the FlipPanel shown earlier in Figure 18-5, using a StackPanel full of elements for the front content region and a Grid for the back:

```
<lib:FlipPanel x:Name="panel" BorderBrush="DarkOrange"
 BorderThickness="3" CornerRadius="4" Margin="10">
  <lib:FlipPanel.FrontContent>
    <StackPanel Margin="6">
      <TextBlock TextWrapping="Wrap" Margin="3" FontSize="16"
       Foreground="DarkOrange">This is the front side of the FlipPanel.</TextBlock>
      <Button Margin="3" Padding="3" Content="Button One"></Button>
      <Button Margin="3" Padding="3" Content="Button Two"></Button>
      <Button Margin="3" Padding="3" Content="Button Three"></Button>
      <Button Margin="3" Padding="3" Content="Button Four"></Button>
    </StackPanel>
  </lib:FlipPanel.FrontContent>

  <lib:FlipPanel.BackContent>
    <Grid Margin="6">
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition></RowDefinition>
      </Grid.RowDefinitions>
      <TextBlock TextWrapping="Wrap" Margin="3" FontSize="16"
       Foreground="DarkMagenta">This is the back side of the FlipPanel.</TextBlock>
      <Button Grid.Row="2" Margin="3" Padding="10" Content="Flip Back to Front"
```

```
        HorizontalAlignment="Center" VerticalAlignment="Center"
        Click="cmdFlip_Click"></Button>
    </Grid>
  </lib:FlipPanel.BackContent>
</lib:FlipPanel>
```

When clicked, the button on the back side of the FlipPanel programmatically flips the panel:

```
private void cmdFlip_Click(object sender, RoutedEventArgs e)
{
    panel.IsFlipped = !panel.IsFlipped;
}
```

This has the same result as clicking the ToggleButton with the arrow, which is defined as part of the default control template.

Using a Different Control Template

Custom controls that have been designed properly are extremely flexible. In the case of the FlipPanel, you can supply a new template to change the appearance and placement of the ToggleButton and the animated effects that are used when flipping between the front and back content regions.

Figure 18-6 shows one such example. Here, the flip button is placed in a special bar that's at the bottom of the front side and the top of the back side. And when the panel flips, it doesn't turn its content like a sheet of paper. Instead, it squares the front content into nothingness at the top of the panel while simultaneously expanding the back content underneath. When the panel flips the other way, the back content squishes back down, and the front content expands from the top. For even more visual pizzazz, the content that's being squashed is also blurred with the help of the BlurEffect class.

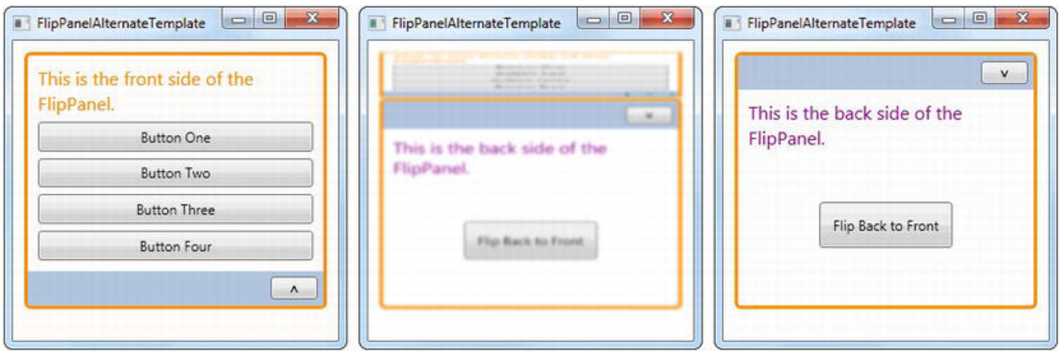

FlipPanelAlternateTemplate
This is the front side of the FlipPanel.
Button One
Button Two
Button Three
Button Four
FlipPanelAlternateTemplate
This is the back side of the FlipPanel.
Flip Back to Front
FlipPanelAlternateTemplate
This is the back side of the FlipPanel.
Flip Back to Front

__Figure 18-6.__ The FlipPanel with a different control template

Here's the portion of the template that defines the front content region:

```
<Border BorderBrush="{TemplateBinding BorderBrush}"
 BorderThickness="{TemplateBinding BorderThickness}"
 CornerRadius="{TemplateBinding CornerRadius}"
 Background="{TemplateBinding Background}">

  <Border.RenderTransform>
```

```
    <ScaleTransform x:Name="FrontContentTransform"></ScaleTransform>
  </Border.RenderTransform>
  <Border.Effect>
    <BlurEffect x:Name="FrontContentEffect" Radius="0"></BlurEffect>
  </Border.Effect>

  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition></RowDefinition>
      <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>

    <ContentPresenter Content="{TemplateBinding FrontContent}"></ContentPresenter>
    <Rectangle Grid.Row="1" Stretch="Fill" Fill="LightSteelBlue"></Rectangle>
    <ToggleButton Grid.Row="1" x:Name="FlipButton" Margin="5" Padding="15,0"
     Content="^" FontWeight="Bold" FontSize="12" HorizontalAlignment="Right">
    </ToggleButton>
  </Grid>
</Border>
```

The back content region is almost the same. It consists of a Border that contains a ContentPresenter element, and it includes its own ToggleButton placed at the right edge of the shaded rectangle. It also defines the all-important ScaleTransform and BlurEffect on the Border, which is what the animations use to flip the panel.

Here are the animations that flip the panel. To see all the markup, refer to the downloadable code for this chapter.

```
<VisualState x:Name="Flipped">
  <Storyboard>
    <DoubleAnimation Storyboard.TargetName="FrontContentTransform"
     Storyboard.TargetProperty="ScaleY" To="0" ></DoubleAnimation>

    <DoubleAnimation Storyboard.TargetName="FrontContentEffect"
     Storyboard.TargetProperty="Radius" To="30" ></DoubleAnimation>

    <DoubleAnimation Storyboard.TargetName="BackContentTransform"
     Storyboard.TargetProperty="ScaleY" To="1" ></DoubleAnimation>

    <DoubleAnimation Storyboard.TargetName="BackContentEffect"
     Storyboard.TargetProperty="Radius" To="0" ></DoubleAnimation>
  </Storyboard>
</VisualState>
```

Because the animation that changes the front content region runs at the same time as the animation that changes the back content region, you don't need a custom transition to manage them.

Creating Custom Panels

So far, you've seen how to develop two custom controls from scratch, with a custom ColorPicker and FlipPanel. In the following sections, you'll consider two more specialized options: deriving a custom Panel and building a custom-drawn control.

Creating a custom panel is a specific but relatively common subset of custom control development. As you learned in Chapter 3, panels host one or more children and implement specific layout logic to arrange them appropriately. Custom panels are an essential ingredient if you want to build your own system for tear-off toolbars or dockable windows. Custom panels are often useful when creating composite controls that need a specific nonstandard layout, like fancy docking toolbars.

You're already familiar with the basic types of panels that WPF includes for organizing content (such as the StackPanel, DockPanel, WrapPanel, Canvas, and Grid). You've also seen that some WPF elements use their own custom panels (such as the TabPanel, ToolBarOverflowPanel, and VirtualizingPanel). You can find many more examples of custom panels online. Here are some worth exploring:

- A custom Canvas that allows its children to be dragged with no extra event-handling code (`http://tinyurl.com/9s324ud`)
- Two panels that implement fish-eye and fanning effects on a list of items (`http://tinyurl.com/965bqt3`)
- A panel that uses a frame-based animation to transition from one layout to another (`http://tinyurl.com/95sdzgx`)

In the next sections, you'll learn how to create a custom panel, and you'll consider two straightforward examples—a basic Canvas clone and an enhanced version of the WrapPanel.

The Two-Step Layout Process

Every panel uses the same plumbing: a two-step process that's responsible for sizing and arranging children. The first stage is the *measure* pass, and it's at this point that the panel determines how large its children want to be. The second stage is the *layout* pass, and it's at this point that each control is assigned its bounds. Two steps are required because the panel might need to take into account the desires of all its children before it decides how to partition the available space.

You add the logic for these two steps by overriding the oddly named MeasureOverride() and ArrangeOverride() methods, which are defined in the FrameworkElement class as part of the WPF layout system. The odd names represent that the MeasureOverride() and ArrangeOverride() methods replace the logic that's defined in the MeasureCore() and ArrangeCore() methods that are defined in the UIElement class. These methods are *not* overridable.

MeasureOverride()

The first step is to determine how much space each child wants by using the MeasureOverride() method. However, even in the MeasureOverride() method, children aren't given unlimited room. At a bare minimum, children are confined to fit in the space that's available to the panel. Optionally, you might want to limit them more stringently. For example, a Grid with two proportionally sized rows will give children half the available height. A StackPanel will offer the first element all the space that's available, then offer the second element whatever's left, and so on.

Every MeasureOverride() implementation is responsible for looping through the collection of children and calling the Measure() method of each one. When you call the Measure() method, you supply the bounding box—a Size object that determines the maximum available space for the child control. At the end of the MeasureOverride() method, the panel returns the space it needs to display all its children and their desired sizes.

Here's the basic structure of the MeasureOverride() method, without the specific sizing details:

```
protected override Size MeasureOverride(Size constraint)
{
    // Examine all the children.
    foreach (UIElement element in base.InternalChildren)
    {
        // Ask each child how much space it would like, given the
        // availableSize constraint.
        Size availableSize = new Size(...);
        element.Measure(availableSize);
        // (You can now read element.DesiredSize to get the requested size.)
    }

    // Indicate how much space this panel requires.
    // This will be used to set the DesiredSize property of the panel.
    return new Size(...);
}
```

The Measure() method doesn't return a value. After you call Measure() on a child, that child's DesiredSize property provides the requested size. You can use this information in your calculations for future children (and to determine the total space required for the panel).

You *must* call Measure() on each child, even if you don't want to constrain the child's size or use the DesiredSize property. Many elements will not render themselves until you've called Measure(). If you want to give a child free reign to take all the space it wants, pass a Size object with a value of Double.PositiveInfinity for both dimensions. (The ScrollViewer is one element that uses this strategy, because it can handle any amount of content.) The child will then return the space it needs for all its content. Otherwise, the child will normally return the space it needs for its content or the space that's available—whichever is smaller.

At the end of the measuring process, the layout container must return its desired size. In a simple panel, you might calculate the panel's desired size by combining the desired size of every child.

■ **Note** You can't simply return the constraint that's passed to the MeasureOverride() method for the desired size of your panel. Although this seems like a good way to take all the available size, it runs into trouble if the container passes in a Size object with Double.PositiveInfinity for one or both dimensions (which means "take all the space you want"). Although an infinite size is allowed as a sizing constraint, it's not allowed as a sizing result, because WPF won't be able to figure out how large your element should be. Furthermore, you really shouldn't take more space than you need. Doing so can cause extra whitespace and force elements that occur after your layout panel to be bumped further down the window.

Attentive readers may have noticed that there's a close similarity between the Measure() method that's called on each child and the MeasureOverride() method that defines the first step of the panel's layout logic. In fact, the Measure() method triggers the MeasureOverride() method. Thus, if you place one layout container inside another, when you call Measure(), you'll get the total size required for the layout container and all its children.

■ **Tip** One reason the measuring process goes through two steps—a Measure() method that triggers the MeasureOverride() method—is to deal with margins. When you call Measure(), you pass in the total available space.

When WPF calls the MeasureOverride() method, it automatically reduces the available space to take margin space into account (unless you've passed in an infinite size).

ArrangeOverride()

After every element has been measured, it's time to lay them out in the space that's available. The layout system calls the ArrangeOverride() method of your panel, and the panel calls the Arrange()method of each child to tell it how much space it's been allotted. (As you can probably guess, the Arrange() method triggers the ArrangeOverride() method, much as the Measure() method triggers the MeasureOverride() method.)

When measuring items with the Measure() method, you pass in a Size object that defines the bounds of the available space. When placing an item with the Arrange() method, you pass in a System.Windows. Rect object that defines the size *and* position of the item. At this point, it's as though every element is placed with Canvas-style X and Y coordinates that determine the distance between the top-left corner of your layout container and the element.

■ **Note** Elements (and layout panels) are free to break the rules and attempt to draw outside of their allocated bounds. For example, in Chapter 12 you saw how the Line can overlap adjacent items. However, ordinary elements should respect the bounds they're given. Additionally, most containers will clip children that extend outside their bounds.

Here's the basic structure of the ArrangeOverride() method, without the specific sizing details:

```
protected override Size ArrangeOverride(Size arrangeSize)
{
    // Examine all the children.
    foreach (UIElement element in base.InternalChildren)
    {
        // Assign the child its bounds.
        Rect bounds = new Rect(...);
        element.Arrange(bounds);
        // (You can now read element.ActualHeight and element.ActualWidth
        //  to find out the size it used.)
    }

    // Indicate how much space this panel occupies.
    // This will be used to set the ActualHeight and ActualWidth properties
    // of the panel.
    return arrangeSize;
}
```

When arranging elements, you can't pass infinite sizes. However, you can give an element its desired size by passing in the value from its DesiredSize property. You can also give an element *more* space than it requires. In fact, this happens frequently. For example, a vertical StackPanel gives a child as much height as it requests but gives it the full width of the panel itself. Similarly, a Grid might use fixed or proportionally sized rows that are larger than the desired size of the element inside. And even if you've placed an element in a size-to-content container, that element can still be enlarged if an explicit size has been set using the Height and Width properties.

When an element is made larger than its desired size, the HorizontalAlignment and VerticalAlignment properties come into play. The element content is placed somewhere inside the bounds that it has been given.

Because the ArrangeOverride() method always receives a defined size (not an infinite size), you can return the Size object that's passed in to set the final size of your panel. In fact, many layout containers take this step to occupy all the space that's been given. (You aren't in danger of taking up space that could be needed for another control, because the measure step of the layout system ensures that you won't be given more space than you need unless that space is available.)

The Canvas Clone

The quickest way to get a grasp of these two methods is to explore the inner workings of the Canvas class, which is the simplest layout container. To create your own Canvas-style panel, you simply need to derive from Panel and add the MeasureOverride() and ArrangeOverride() methods shown next:

```
public class CanvasClone : System.Windows.Controls.Panel
{ ... }
```

The Canvas places children where they want to be placed and gives them the size they want. As a result, it doesn't need to calculate how the available space should be divided. That makes its MeasureOverride() method extremely simple. Each child is given infinite space to work with:

```
protected override Size MeasureOverride(Size constraint)
{
    Size size = new Size(double.PositiveInfinity, double.PositiveInfinity);
    foreach (UIElement element in base.InternalChildren)
    {
        element.Measure(size);
    }
    return new Size();
}
```

Notice that the MeasureOverride() returns an empty Size object, which means the Canvas doesn't request any space at all. It's up to you to specify an explicit size for the Canvas or place it in a layout container that will stretch it to fill the available space.

The ArrangeOverride() method is only slightly more involved. To determine the proper placement of each element, the Canvas uses attached properties (Left, Right, Top, and Bottom). As you learned in Chapter 4 (and as you'll see in the WrapBreakPanel next), attached properties are implemented with two helper methods in the defining class: a Get*Property*() and a Set*Property*() method.

The Canvas clone that you're considering is a bit simpler—it respects only the Left and Top attached properties (not the redundant Right and Bottom properties). Here's the code it uses to arrange elements:

```
protected override Size ArrangeOverride(Size arrangeSize)
{
    foreach (UIElement element in base.InternalChildren)
    {
        double x = 0;
        double y = 0;
        double left = Canvas.GetLeft(element);
        if (!DoubleUtil.IsNaN(left))
        {
            x = left;
        }
        double top = Canvas.GetTop(element);
        if (!DoubleUtil.IsNaN(top))
        {
            y = top;
        }
        element.Arrange(new Rect(new Point(x, y), element.DesiredSize));
    }
    return arrangeSize;
}
```

A Better Wrapping Panel

Now that you've examined the panel system in a fair bit of detail, it's worth creating your own layout container that adds something you can't get with the basic set of WPF panels. In this section, you'll see an example that extends the capabilities of the WrapPanel.

The WrapPanel performs a simple function that's occasionally quite useful. It lays out its children one after the other, moving to the next line after the width in the current line is used up. However, occasionally you need a way to force an immediate line break, so you can start a specific control on a new line. Although the stock WrapPanel doesn't provide this capability, it's fairly easy to add one if you create a custom control. All you need to do is add an attached property that requests a line break. Then the child elements in the panel can use this property to start a new line at the right spot.

The following listing shows a WrapBreakPanel that adds an attached LineBreakBeforeProperty. When set to true, this property causes an immediate line break before the element.

```
public class WrapBreakPanel : Panel
{
    public static DependencyProperty LineBreakBeforeProperty;

    static WrapBreakPanel()
    {
        FrameworkPropertyMetadata metadata = new FrameworkPropertyMetadata();
        metadata.AffectsArrange = true;
        metadata.AffectsMeasure = true;
        LineBreakBeforeProperty = DependencyProperty.RegisterAttached(
          "LineBreakBefore", typeof(bool), typeof(WrapBreakPanel), metadata);
    }
    ...
}
```

As with any dependency property, the LineBreakBefore property is defined as a static field and then registered in the static constructor for your class. The only difference is that you use the RegisterAttached() method rather than Register().

The FrameworkPropertyMetadata object for the LineBreakBefore property specifically indicates that it affects the layout process. As a result, a new layout pass will be triggered whenever this property is set.

Attached properties aren't wrapped by normal property wrappers, because they aren't set in the same class that defines them. Instead, you need to provide two static methods that can use the DependencyObject.SetValue() method to set this property on any arbitrary element. Here's the code that you need for the LineBreakBefore property:

```
public static void SetLineBreakBefore(UIElement element, Boolean value)
{
    element.SetValue(LineBreakBeforeProperty, value);
}
public static Boolean GetLineBreakBefore(UIElement element)
{
    return (bool)element.GetValue(LineBreakBeforeProperty);
}
```

The only remaining detail is to take this property into account when performing the layout logic. The layout logic of the WrapBreakPanel is based on the WrapPanel. During the measure stage, elements are arranged into lines so that the panel can calculate the total space it needs. Each element is added into the current line unless it's too large or the LineBreakBefore property is set to true. Here's the full code:

```
protected override Size MeasureOverride(Size constraint)
{
    Size currentLineSize = new Size();
    Size panelSize = new Size();

    foreach (UIElement element in base.InternalChildren)
    {
        element.Measure(constraint);
        Size desiredSize = element.DesiredSize;

        if (GetLineBreakBefore(element) ||
          currentLineSize.Width + desiredSize.Width > constraint.Width)
        {
            // Switch to a new line (either because the element has requested it
            // or space has run out).
            panelSize.Width = Math.Max(currentLineSize.Width, panelSize.Width);
            panelSize.Height += currentLineSize.Height;
            currentLineSize = desiredSize;

            // If the element is too wide to fit using the maximum width
            // of the line, just give it a separate line.
            if (desiredSize.Width > constraint.Width)
            {
                panelSize.Width = Math.Max(desiredSize.Width, panelSize.Width);
                panelSize.Height += desiredSize.Height;
                currentLineSize = new Size();
            }
        }
        else
        {
            // Keep adding to the current line.
            currentLineSize.Width += desiredSize.Width;

            // Make sure the line is as tall as its tallest element.
            currentLineSize.Height = Math.Max(desiredSize.Height,
              currentLineSize.Height);
        }
    }

    // Return the size required to fit all elements.
    // Ordinarily, this is the width of the constraint, and the height
    // is based on the size of the elements.
    // However, if an element is wider than the width given to the panel,
    // the desired width will be the width of that line.
    panelSize.Width = Math.Max(currentLineSize.Width, panelSize.Width);
    panelSize.Height += currentLineSize.Height;
    return panelSize;
}
```

The key detail in this code is the test that checks the LineBreakBefore property. This implements the additional logic that's not provided in the ordinary WrapPanel.

The code for ArrangeOverride() is almost the same but slightly more tedious. The difference is that the panel needs to determine the maximum height of the line (which is determined by the tallest element) before it begins laying out that line. That way, each element can be given the full amount of available space, which takes into account the full height of the line. This is the same process that's used to lay out an ordinary WrapPanel. To see the full details, refer to the downloadable code examples for this chapter.

Using the WrapBreakPanel is easy. Here's some markup that demonstrates that the WrapBreakPanel correctly separates lines and calculates the right desired size based on the size of its children:

```
<StackPanel>
  <StackPanel.Resources>
    <Style TargetType="{x:Type Button}">
      <Setter Property="Margin" Value="3"></Setter>
      <Setter Property="Padding" Value="3"></Setter>
    </Style>
  </StackPanel.Resources>

  <TextBlock Padding="5" Background="LightGray">
    Content above the WrapBreakPanel.
  </TextBlock>
  <lib:WrapBreakPanel>
    <Button>No Break Here</Button>
    <Button>No Break Here</Button>
    <Button>No Break Here</Button>
    <Button>No Break Here</Button>
    <Button lib:WrapBreakPanel.LineBreakBefore="True" FontWeight="Bold">
     Button with Break
    </Button>
    <Button>No Break Here</Button>
    <Button>No Break Here</Button>
    <Button>No Break Here</Button>
    <Button>No Break Here</Button>
  </lib:WrapBreakPanel>
  <TextBlock Padding="5" Background="LightGray">
    Content below the WrapBreakPanel.
  </TextBlock>
</StackPanel>
```

Figure 18-7 shows how this markup is interpreted.

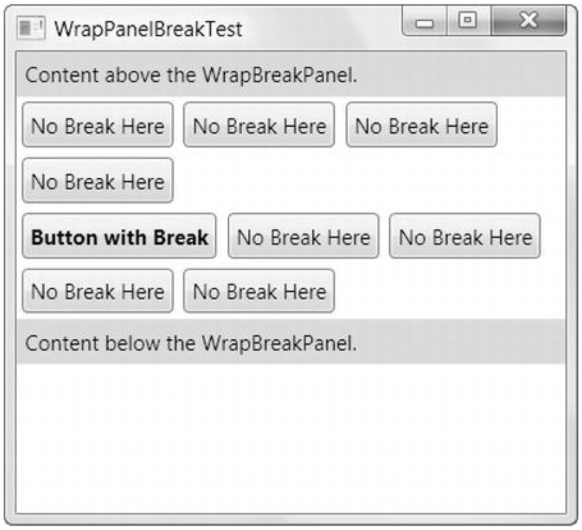


Figure 18-7. *The WrapBreakPanel*

Custom-Drawn Elements

In the previous section, you began to explore the inner workings of WPF elements—namely, the MeasureOverride() and ArrangeOverride() methods that allow every element to plug into WPF's layout system. In this section, you'll delve a bit deeper and consider how elements render themselves.

Most WPF elements use *composition* to create their visual appearance. In other words, a typical element builds itself out of other, more fundamental elements. You've seen this pattern at work throughout this chapter. For example, you define the composite elements of a user control by using markup that's processed in the same way as the XAML in a custom window. You define the visual tree for a custom control by using a control template. And when creating a custom panel, you don't need to define any visual details at all. The composite elements are provided by the control consumer and added to the Children collection.

Of course, composition can take you only so far. Eventually, some class needs to take responsibility for drawing content. In WPF, this point is a long way down the element tree. In a typical window, the rendering is performed by individual bits of text, shapes, and bitmaps, rather than high-level elements.

The OnRender() Method

To perform custom rendering, an element must override the OnRender() method, which is inherited from the base UIElement class. The OnRender() method doesn't necessarily replace composition—some controls use OnRender() to paint a visual detail and use composition to layer other elements over it. Two examples are the Border class, which draws its border in the OnRender() method, and the Panel class, which draws its background in the OnRender() method. Both the Border and Panel support child content, and this content is rendered on top of the custom-drawn details.

The OnRender() method receives a DrawingContext object, which provides a set of useful methods for drawing content. You first learned about the DrawingContext class in Chapter 14, when you used it to draw the content for a Visual object. The key difference when performing drawing in the OnRender() method is that you don't explicitly create and close the DrawingContext. That's because several different OnRender() methods could conceivably use the same DrawingContext. For example, a derived element might perform some custom drawing and call the OnRender() implementation in the base class to draw additional content. This works because WPF automatically creates the DrawingContext object at the beginning of this process and closes it when it's no longer needed.

■ **Note** Technically, the OnRender() method doesn't actually *draw* your content to the screen. Instead, it draws your content to the DrawingContext object, and WPF then caches that information. WPF determines when your element needs to be repainted and paints the content that you created with the DrawingContext. This is the essence of WPF's retained graphics system—you define the content, and it manages the painting and refreshing process seamlessly.

The most surprising detail about WPF rendering is that so few classes actually do it. Most classes are built out of other simpler classes, and you need to dig quite a way down the element tree of a typical control before you discover a class that actually overrides OnRender(). Here are some that do:

The TextBlock class: Wherever you place text, there's a TextBlock object using OnRender() to draw it.

The Image class: The Image class overrides OnRender() to paint its image content by using the DrawingContext.DrawImage() method.

The MediaElement class: The MediaElement overrides OnRender() to draw a frame of video, if it's being used to play a video file.

The shape classes: The base Shape class overrides OnRender() to draw its internally stored Geometry object, with the help of the DrawingContext. DrawGeometry() method. This Geometry object could represent an ellipse, a rectangle, or a more complex path composed of lines and curves, depending on the specific Shape-derived class. Many elements use shapes to draw small visual details.

The chrome classes: Classes such as ButtonChrome and ListBoxChrome draw the outer appearance of a common control and place the content you specify inside. Many other Decorator-derived classes, such as Border, also override OnRender().

The panel classes: Although the content of a panel is supplied by its children, the OnRender() method takes care of drawing a rectangle with the background color if the Background property is set.

Often, the OnRender() implementation is deceptively simple. For example, here's the rendering code for any Shape-derived class:

```
protected override void OnRender(DrawingContext drawingContext)
{
    this.EnsureRenderedGeometry();
    if (this._renderedGeometry != Geometry.Empty)
    {
        drawingContext.DrawGeometry(this.Fill, this.GetPen(),
          this._renderedGeometry);
    }
}
```

Remember, overriding OnRender() isn't the only way to render content and add it to your user interface. You can also create a DrawingVisual object and add that visual to a UIElement by using the AddVisualChild() method (and implementing a few other details, as described in Chapter 14). You can then call DrawingVisual.RenderOpen() to retrieve a DrawingContext for your DrawingVisual and use it to render its content.

Some elements use this strategy in WPF to display some graphical detail on top of other element content. For example, you'll see it with drag-and-drop indicators, error indicators, and focus boxes. In all these cases, the DrawingVisual approach allows the element to draw content *over* other content, rather than *under* it. But for the most part, rendering takes place in the dedicated OnRender() method.

Evaluating Custom Drawing

When you create your own custom elements, you may choose to override OnRender() to draw custom content. You might override OnRender() in an element that contains content (most commonly, a Decorator-derived class) so you can add a graphical embellishment around that content. Or you might override OnRender() in an element that doesn't have any nested content so that you can draw its full visual appearance. For example, you might create a custom element that draws a small graphical detail, which you can then use in another control through composition. One example in WPF is the TickBar element, which draws the tick marks for a Slider. The TickBar is embedded in the visual tree of a Slider through the Slider's default control template (along with a Border and a Track that includes two RepeatButton controls and a Thumb).

The obvious question is when to use the comparatively low-level OnRender() approach and when to use composition with other classes (such as the Shape-derived elements) to draw what you need. To decide, you need to evaluate the complexity of the graphics you need and the interactivity you want to provide.

For example, consider the ButtonChrome class. In WPF's implementation of the ButtonChrome class, the custom rendering code takes various properties into account, including RenderDefaulted, RenderMouseOver, and RenderPressed. The default control template for the Button uses triggers to set these properties at the appropriate time, as you saw in Chapter 17. For example, when the mouse moves over the button, the Button class uses a trigger to set the ButtonChrome.RenderMouseOver property to true.

Whenever the RenderDefaulted, RenderMouseOver, or RenderPressed property is changed, the ButtonChrome calls the base InvalidateVisual() method to indicate that its current appearance is no longer valid. WPF then calls the ButtonChrome.OnRender() method to get its new graphical representation.

If the ButtonChrome class used composition, this behavior would be more difficult to implement. It's easy enough to create the standard appearance for the ButtonChrome class by using the right elements, but it's more work to modify it when the button's state changes. You'd need to dynamically change the nested elements that compose the ButtonChrome class or—if the appearance changes more dramatically—you'd be forced to hide one element and show another one in its place.

Most custom elements won't need custom rendering. But if you need to render complex visuals that change significantly when properties are changed or certain actions take place, the custom rendering approach just might be easier to use and more lightweight.

Creating a Custom-Drawn Element

Now that you know how the OnRender() method works and when to use it, the last step is to consider a custom control that demonstrates it in action.

The following code defines an element named CustomDrawnElement that demonstrates a simple effect. It paints a shaded background by using the RadialGradientBrush. The trick is that the highlight point where the gradient starts is set dynamically, so it follows the mouse. Thus, as the user moves the mouse over the control, the white glowing center point follows, as shown in Figure 18-8.

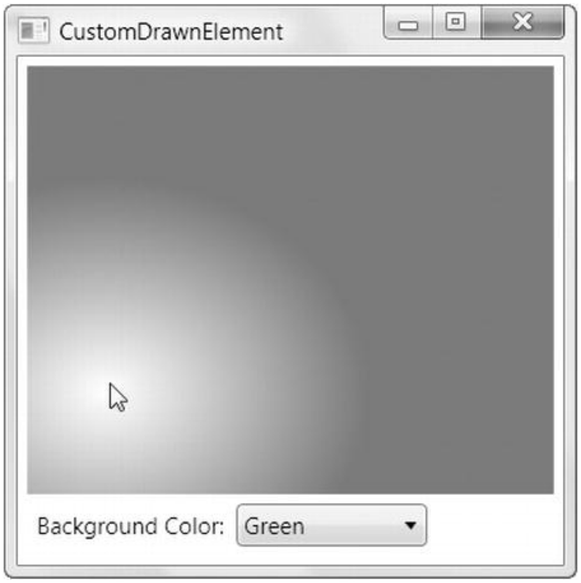


Figure 18-8. *A custom-drawn element*

The CustomDrawnElement doesn't need to contain any child content, so it derives directly from FrameworkElement. It allows only a single property to be set—the background color of the gradient. (The foreground color is hard-coded to be white, although you could easily change this detail.)

```
public class CustomDrawnElement : FrameworkElement
{
    public static DependencyProperty BackgroundColorProperty;

    static CustomDrawnElement()
    {
        FrameworkPropertyMetadata metadata =
          new FrameworkPropertyMetadata(Colors.Yellow);
        metadata.AffectsRender = true;
        BackgroundColorProperty =  DependencyProperty.Register("BackgroundColor",
        typeof(Color), typeof(CustomDrawnElement), metadata);
    }

    public Color BackgroundColor
    {
        get { return (Color)GetValue(BackgroundColorProperty); }
        set { SetValue(BackgroundColorProperty, value); }
    }
    ...
```

The BackgroundColor dependency property is specifically marked with the FrameworkPropertyMetadata.AffectRender flag. As a result, WPF will automatically call OnRender() whenever the color is changed. However, you also need to make sure OnRender() is called when the mouse moves to a new position. This is handled by calling the InvalidateVisual() method at the right times:

```
    ...
    protected override void OnMouseMove(MouseEventArgs e)
    {
        base.OnMouseMove(e);
        this.InvalidateVisual();
    }

    protected override void OnMouseLeave(MouseEventArgs e)
    {
        base.OnMouseLeave(e);
        this.InvalidateVisual();
    }
    ...
```

The only remaining detail is the rendering code. It uses the DrawingContext.DrawRectangle() method to paint the element's background. The ActualWidth and ActualHeight properties indicate the final rendered dimensions of the control.

```
    ...
    protected override void OnRender(DrawingContext dc)
    {
        base.OnRender(dc);

        Rect bounds = new Rect(0, 0, base.ActualWidth, base.ActualHeight);
        dc.DrawRectangle(GetForegroundBrush(), null, bounds);
    }
    ...
```

Finally, a private helper method named GetForegroundBrush() constructs the correct RadialGradientBrush based on the current position of the mouse. To calculate the center point, you need to convert the current position of the mouse over the element to a relative position from 0 to 1, which is what the RadialGradientBrush expects.

```
    ...
    private Brush GetForegroundBrush()
    {
        if (!IsMouseOver)
        {
            return new SolidColorBrush(BackgroundColor);
        }
        else
        {
            RadialGradientBrush brush = new RadialGradientBrush(
              Colors.White, BackgroundColor);

            // Get the position of the mouse in device-independent units,
            // relative to the control itself.
            Point absoluteGradientOrigin = Mouse.GetPosition(this);

            // Convert the point coordinates to proportional (0 to 1) values.
            Point relativeGradientOrigin = new Point(
              absoluteGradientOrigin.X / base.ActualWidth,
              absoluteGradientOrigin.Y / base.ActualHeight);

            // Adjust the brush.
            brush.GradientOrigin = relativeGradientOrigin;
            brush.Center = relativeGradientOrigin;

            return brush;
        }
    }
}
```

This completes the example.

Creating a Custom Decorator

As a general rule, you should never use custom drawing in a control. If you do, you violate the premise of WPF's lookless controls. The problem is that after you hardwire in some drawing logic, you've ensured that a portion of your control's visual appearance cannot be customized through the control template.

A much better approach is to design a separate element that draws your custom content (such as the CustomDrawnElement class in the previous example) and then use that element inside the default control template for your control. That's the approach used in many WPF controls, and you saw it at work in the Button control in Chapter 17.

It's worth quickly considering how you can adapt the previous example so that it can function as part of a control template. Custom-drawn elements usually play two roles in a control template:

- They draw some small graphical detail (such as the arrow on a scroll button).
- They provide a more detailed background or frame around another element.

The second approach requires a custom decorator. You can change the CustomDrawnElement into a custom-drawn element by making two small changes. First, derive it from Decorator:

```
public class CustomDrawnDecorator : Decorator
```

Next, override the OnMeasure() method to specify the required size. It's the responsibility of all decorators to consider their children, add the extra space required for their embellishments, and then return the combined size. The CustomDrawnDecorator doesn't need any extra space to draw a border. Instead, it simply makes itself as large as the content warrants by using this code:

```
protected override Size MeasureOverride(Size constraint)
{
    UIElement child = this.Child;
    if (child != null)
```

```
        {
            child.Measure(constraint);
            return child.DesiredSize;
        }
        else
        {
            return new Size();
        }
    }
```

After you've created your custom decorator, you can use it in a custom control template. For example, here's a button template that places the mouse-tracking gradient background behind the button content. It uses template bindings to make sure the properties for alignment and padding are respected.

```
<ControlTemplate x:Key="ButtonWithCustomChrome">
  <lib:CustomDrawnDecorator BackgroundColor="LightGreen">
    <ContentPresenter Margin="{TemplateBinding Padding}"
     HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
     VerticalAlignment="{TemplateBinding VerticalContentAlignment}"
     ContentTemplate="{TemplateBinding ContentControl.ContentTemplate}"
     Content="{TemplateBinding ContentControl.Content}"
     RecognizesAccessKey="True" />
  </lib:CustomDrawnDecorator>
</ControlTemplate>
```

You can now use this template to restyle your buttons with a new look. Of course, to make your decorator more practical, you'd probably want to vary its appearance when the mouse button is clicked. You can do this by using triggers that modify properties in your chrome class. Chapter 17 has a complete discussion of this design.

The Last Word

In this chapter, you took a detailed look at custom control development in WPF. You saw how to build basic user controls and extend existing WPF controls and how to create the WPF gold standard—a template-based lookless control. Finally, you considered custom drawing and how you can use custom-drawn content with a template-based control.

If you're planning to dive deeper into the world of custom control development, you'll find some excellent samples online. One good starting point is Microsoft's set of control customization samples that are available at `http://tinyurl.com/9jtk93x`. Another worthwhile download is the actively maintained Bag-o-Tricks sample project provided by Kevin Moore (a former program manager on the WPF team) at `http://tinyurl.com/95sdzgx`, which includes everything from basic date controls to a panel with built-in animation.