15

Templates

Properties and styles determine a control's appearance and behavior. For example, a `Slider` control's `TickFrequency`, `TickPlacement`, `Background`, and `Width` properties help determine its appearance, while its `Minimum`, `Maximum`, `LargeChange`, and `IsEnabled` properties help determine its behavior.

In contrast, *templates* determine a control's structure. They determine what components make up the control and how those components interact to provide the control's features.

This chapter describes templates in general terms and shows how you can build templates of your own to change the way existing controls work.

TEMPLATE OVERVIEW

If you look closely at Figure 15-1, you can see that a `Slider` control has a bunch of parts including:

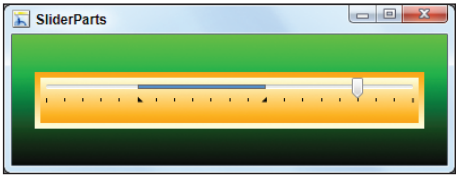

FIGURE 15-1

- A border
- Tick marks
- A background
- Clickable areas on the background (basically anywhere between the top of the control and its tick marks vertically) that change the current value
- A Thumb indicating the current value that you can drag back and forth
- Selection indicators (the little black triangles) that indicate a selected range

These features are provided by the pieces that make up the `Slider`. By default, a `Slider` is made up of a multitude of `Border`, `Grid`, `TickBar`, `Track`, `RepeatButton`, `Rectangle`, `Thumb`, `Canvas`, and `Path` controls, together with many `brushes`, `transformations`, `styles`, and `triggers`.

A *template* determines what the pieces are that make up a control. It determines the control's components together with their styles, triggers, and everything else that is needed by the control. As an analogy, consider a car. Its *properties* are easily changed — things like its color,

upholstery, and vanity plate (e.g., *WPF FAN*). Its *template* defines the things it is made of — for example, its chassis, number of doors, and engine.

No matter what combination of properties and components you pick, however, it has certain standard car-like features such as turning on, accelerating, decelerating, turning off, and costing way too much to insure. As you drive down the street, you will see hundreds of combinations, but they are all easily recognizable as cars.

[OK, there may be a few that are hard to recognize such as the MULE robotic logistics vehicle (`www.botmag.com/articles/mule.shtml`), the Terrafugia Transition flyable-car/roadable-plane (`www.theregister.co.uk/2008/07/29/terrafugia_transition_on_show_oshkosh`), or the Toyota PM (`www.toyota.com/concept-vehicles/pm.html`), which looks more like a Star Wars pod racer than a car, but I have yet to see any of these on the road.]

Note that the components influence the behavior of the car. A hybrid has great fuel efficiency but slow acceleration, while a 12-cylinder sports car has great acceleration but poor mileage. Similarly, the components that make up a control can change the way it behaves.

Because the controls in the template determine the control's appearance, WPF controls are sometimes called *lookless*. By creating your own template for an existing control such as a `Button` or `CheckBox`, you can give the control a new appearance and behavior.

WORK WARNING

Building a template can be a lot of work. When you build a template, you take responsibility for most of the control's behavior. You cannot make a `Button` use a diamond-shaped polygon for its surface and expect it to automatically do everything that a normal `Button` does. If you decide to use a template to make a diamond-shaped `Button`, then you need to build most of the `Button`'s behaviors yourself.

The `Button` still provides some very basic features such as raising a `Click` event when the user clicks it, but you need to handle things such as changing the `Button`'s appearance when the mouse is over it, when the user presses the mouse, when the mouse moves off it, and so forth.

CONTENTPRESENTER

If you're assembling a new control from components of your own choosing, how do you handle the essential features of the control? For example, if you're building a template for `Label` controls, perhaps displaying the text inside a `Border` with a beveled edge, how do you know what text to display?

The answer is the `ContentProvider`. The `ContentProvider` is an object that WPF provides to display whatever it is that the control should display. You can place the `ContentProvider` in whatever control hierarchy you build for the template, and it will display the content.

For example, the following code shows an extremely simple `Label` template:


```
<Window.Resources>
    <ControlTemplate x:Key="temSimpleLabel" TargetType="Label">
        <Border BorderBrush="Red" BorderThickness="1">
            <ContentPresenter/>
        </Border>
    </ControlTemplate>
</Window.Resources>
```

SimpleLabelTemplate

The template's name is `temSimpleLabel`, and it applies to `Label` controls. The template contains a `Border` control that displays a red border and that holds the `ContentPresenter`.

The following code shows how the program might use this template. This code creates a `Label`. Its last attribute sets the control's `Template` property to the previously created template.


```
<Label Margin="5" Content="No Template"
 HorizontalContentAlignment="Right"
 VerticalContentAlignment="Center"
 BorderBrush="Yellow" BorderThickness="2"
 Foreground="{StaticResource brForeground}"
 Background="{StaticResource brBackground}"
 Template="{StaticResource temBorderLabel}"
/>
```

SimpleLabelTemplate

The SimpleLabelTemplate example program shown in Figure 15-2 displays two `Labels`. The one on the left uses no template, while the one on the right uses the `temSimpleLabel` template.

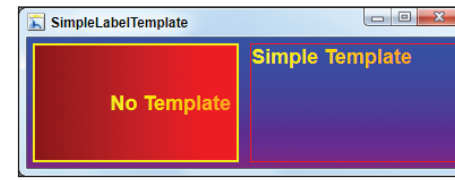

FIGURE 15-2

TEMPLATE BINDING

If you compare the two `Labels` in Figure 15-2, you'll see that even this simple example has some potential problems. Because the template's `Border` control includes explicit values for its `BorderBrush` and `BorderThickness` properties, it overrides any values set in the code that creates the `Label`. The `Border` control also doesn't specify a `Background`, so it uses its default transparent background.

This means the templated control doesn't display the correct background or border. It also doesn't honor the requested `HorizontalContentAlignment` and `VerticalContentAlignment` values.

Fortunately, a template can learn about some of the properties set on the client control by using a *template binding*. For example, the following code fragment sets the `Background` property for a piece of the template to the value set for the control's `Background` property:

```
Background="{TemplateBinding Background}"
```

Template bindings let the template honor values set for the control where appropriate while overriding other values to achieve the appearance you desire.

The following code shows a better version of the `Label` template that honors several of the control's background and foreground properties:


```
<ControlTemplate x:Key="temBetterLabel" TargetType="Label">
    <Border
     Background="{TemplateBinding Background}"
     BorderBrush="{TemplateBinding BorderBrush}"
     BorderThickness="{TemplateBinding BorderThickness}">
        <ContentPresenter Margin="4"
         HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
         VerticalAlignment="{TemplateBinding VerticalContentAlignment}"/>
    </Border>
</ControlTemplate>
```

BetterLabelTemplate

In this template, the `Border` control mimics the client control's `Background`, `BorderBrush`, and `BorderThickness` properties. The `ContentPresenter` sets its `HorizontalAlignment` and `VerticalAlignment` properties to the client control's `HorizontalContentAlignment` and `VerticalContentAlignment` values so that the result is properly aligned within the control.

The BetterLabelTemplate example program shown in Figure 15-3 uses this template to display a `Label` that looks much more like one that has no template.

So now that you can create a template `Label` that looks like a regular `Label`, what's the point? If you just want a `Label` that looks like a `Label`, use a `Label`!

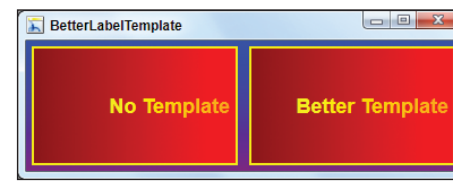

FIGURE 15-3

The point is that you don't have to copy every feature of the original control. You can add bitmap effects, rotate the label, insert an image, and make other changes. For example, the following section describes two `Label` templates that add features not provided by the normal `Label` control.

CHANGING CONTROL APPEARANCE

Of course, you won't always want your template to match exactly the appearance of a control without a template. If you did, you wouldn't bother going to all the trouble of making a template.

Your template will override properties, implement new behaviors, and build the template from controls other than those used by the original control to provide a unique experience.

The InterestingLabelTemplates example program shown in Figure 15-4 demonstrates two more interesting `Label` templates. The first draws a double border around its text if the `Label` specifies `BorderBrush` and `BorderThickness` properties. The second can display text wrapped across multiple lines.

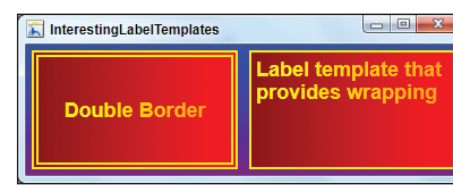

FIGURE 15-4

The following code shows how the InterestingLabelTemplates program displays its `Label` with a double border:


```
<ControlTemplate x:Key="temDoubleBorderLabel" TargetType="Label">
    <Border Background="{TemplateBinding Background}"
     BorderBrush="{TemplateBinding BorderBrush}"
     BorderThickness="{TemplateBinding BorderThickness}">
         <Border Margin="2" Background="Transparent"
          BorderBrush="{TemplateBinding BorderBrush}"
          BorderThickness="{TemplateBinding BorderThickness}">
              <ContentPresenter Margin="2"
               HorizontalAlignment="Center"
               VerticalAlignment="{TemplateBinding VerticalContentAlignment}"/>
        </Border>
    </Border>
</ControlTemplate>
```

InterestingLabelTemplates

This template displays a `Border` control that matches the client control's `Background`, `BorderBrush`, and `BorderThickness` properties.

Inside that is another `Border` control with its `Margin` set to 2, so it sits inside the first `Border`. Its `Background` is set to `Transparent`, so it doesn't cover the background used by the outer `Border`, although the inner `Border` also obeys the client's `BorderBrush` and `BorderThickness` properties.

Finally, inside the inner `Border`, the `ContentPresenter` displays the client's content as before.

The following code shows how the program displays its second `Label` with wrapped text:


```
<ControlTemplate x:Key="temWrappedLabel" TargetType="Label">
    <Grid>
        <Border
         Background="{TemplateBinding Background}"
         BorderBrush="{TemplateBinding BorderBrush}"
         BorderThickness="{TemplateBinding BorderThickness}">
            <TextBlock Name="txtbContent"
             Margin="4"
             TextWrapping="Wrap"
             Text="{TemplateBinding ContentPresenter.Content}"/>
        </Border>
    </Grid>
</ControlTemplate>
```

InterestingLabelTemplates

This template displays a `Border` as before. The `Border` contains a `TextBlock` with `TextWrapping = True`, so it wraps its content if necessary. The `TextBlock`'s `Text` property is set to the `ContentPresenter`'s `Content` property.

Note that this only works if the `ContentPresenter` is trying to display text. For example, if you build the client `Label` control so that it contains a `Button` as shown in the following code, then the `TextBlock` doesn't display anything:


```
<Label Margin="5"
 HorizontalContentAlignment="Right"
 VerticalContentAlignment="Center"
 BorderBrush="Yellow" BorderThickness="2"
 Foreground="{StaticResource brForeground}"
 Background="{StaticResource brBackground}"
 Template="{StaticResource temWrappedLabel}"
>
    <Button Content="Click Me"/>
</Label>
```

InterestingLabelTemplates

TEMPLATE EVENTS

The `Label` control used in the previous example is one of the simplest controls. It mostly just sits there looking pretty without bothering to interact with the user.

But more complicated controls like `Button`, `CheckBox`, and `Slider` must perform all sorts of stunts as the mouse moves, presses, drags, and releases over them.

To make a template control respond to events, you can add property and event triggers to the template much as you added them to styles in Chapters 13 and 14.

In addition to events caused by user actions such as moving or pressing the mouse, controls must respond to changes in state. For example, although a `Label` mostly just sits around doing nothing, it should also change its appearance when it is disabled. If you don't need to display complex animations, then it can simply respond with `Setters` in a property `Trigger` that runs when the `IsEnabled` property is `False`.

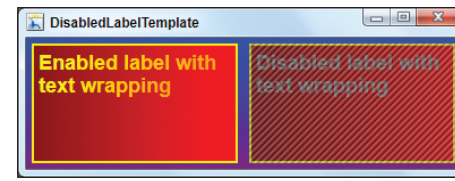

FIGURE 15-5

The DisabledLabelTemplate example program shown in Figure 15-5 uses a template that gives a disabled `Label` a distinctive appearance.

The following code shows the template that gives the disabled `Label` its appearance:


```
<ControlTemplate x:Key="temWrappedLabel" TargetType="Label">
    <Grid>
        <Border Name="brdMain"
         Background="{TemplateBinding Background}"
         BorderBrush="{TemplateBinding BorderBrush}"
         BorderThickness="{TemplateBinding BorderThickness}">
            <TextBlock Name="txtbContent"
             Margin="4"
             TextWrapping="Wrap"
             Text="{TemplateBinding ContentPresenter.Content}"/>
        </Border>
        <Canvas Name="canDisabled" Opacity="0">
```

```
            <Canvas.Background>
                <LinearGradientBrush StartPoint="0,0" EndPoint="3,3"
                 MappingMode="Absolute"
                 SpreadMethod="Repeat">
                    <GradientStop Color="LightGray" Offset="0"/>
                    <GradientStop Color="Black" Offset="1"/>
                </LinearGradientBrush>
            </Canvas.Background>
        </Canvas>
    </Grid>
    <ControlTemplate.Triggers>
        <Trigger Property="IsEnabled" Value="False">
            <Setter TargetName="canDisabled"
             Property="Opacity" Value="0.5"/>
            <Setter TargetName="txtbContent"
             Property="Foreground" Value="Gray"/>
        </Trigger>
    </ControlTemplate.Triggers>
</ControlTemplate>
```

DisabledLabelTemplate

This version of the `Template` starts with a `Grid` control that contains a `Border` and a `Canvas`. The `Border` holds a `TextBlock` that displays the control's `ContentPresenter` as before. The `Canvas` covers the `Border`, is filled with a linear gradient brush, and initially has `Opacity = 0` so it is invisible.

The template's Triggers section contains a property trigger that activates when the control's `IsEnabled` property is `False`. When that happens, the trigger sets the `Canvas`'s `Opacity` property to 0.5 so it partially obscures the control's content. It also changes the `TextBlock`'s `Foreground` to Gray.

TEMPLATE TRICKS 1

Using a control with `Opacity = 0` is a common and particularly useful template trick. The template can use it to display something new, cover something old, or, as in this example, partially obscure whatever lies behind it.

You can use a translucent white control to wash out whatever is behind, a translucent black control to darken whatever is behind, or an opaque control to cover the background controls completely.

TEMPLATE TRICKS 2

It's easier for triggers to manipulate the template's controls and other objects if you give those objects names. In this example, the `TextBlock` is named `txtbContent` and the translucent `Canvas` is named `canDisabled`, so it's easy for the triggers to control them. If you'll need to animate it, give it a name.

The following sections describe some much more complex templates that change the way `Buttons` work.

GLASS BUTTON

The GlassButton example program shown in Figure 15-6 uses a template to give its buttons a glassy appearance.

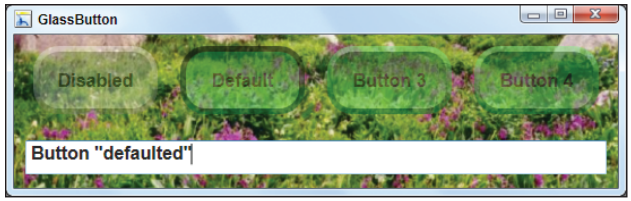

FIGURE 15-6

The disabled button on the left looks washed-out and doesn't respond to the user.

The second button labeled *Default* has a different border from that of the other buttons. If no other button has the focus when the user presses the [Enter] key, that `Button` fires its `Click` event. In Figure 15-6, the `TextBox` has the focus, so pressing the [Enter] key will fire the default `Button`.

DESIGNATED DEFAULT

Just because a `Button`'s `IsDefault` property is `True`, that doesn't mean that the `Button` always fires when the user presses [Enter]. If the focus is on another `Button`, then the [Enter] key fires that `Button` instead of the default.

Also, when the default `Button` has the focus, it behaves like any other `Button` with the focus, so it is not acting as the default `Button` at that time.

When a `Button` is acting as the default, it is said to be *defaulted*. You (or, more importantly, your triggers) can see whether a `Button` is defaulted by checking its `IsDefaulted` property.

Figure 15-7 shows the program when the mouse is over Button 3. The button under the mouse becomes less transparent. Notice that the focus is still in the `TextBox` (you can see the caret in Figure 15-7), so the default button still shows its distinctive border.

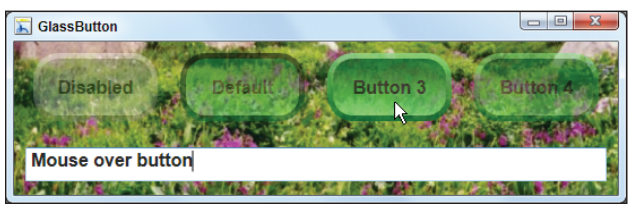

FIGURE 15-7

Figure 15-8 shows the program when the user presses the mouse down on Button 3.

FIGURE 15-8

At this point, the pressed button is opaque. Pressing the button also moves focus to that `button`. Because focus is now on Button 3, the default button is no longer defaulted. In fact, no button is defaulted right now. Button 3 has the focus so it will fire if the user presses [Enter] but it is not defaulted so it won't display the default border even after the user releases the mouse.

If you drag the mouse off the button while it is still pressed, the button returns to its focused "mouse over" appearance. If you then release the mouse, no mouse click occurs.The following three sections describe the glass button's `Template`. The first describes the `Template` at a high level, explaining the controls the `Template` uses and how they fit together. The two sections that follow describe the `Template`'s `Styles` and `Triggers`.

This program is fairly long so the complete code isn't shown in these sections. You can download the example program from the book's web site to see the details.

Glass Button Template Overview

The following code snippet shows the `Template`'s main sections and the controls that it uses.


```
<ControlTemplate x:Key="temGlassButton" TargetType="Button">
    <ControlTemplate.Resources>
        ... Template Styles omitted here...
    </ControlTemplate.Resources>

    <Grid Name="grdMain" ClipToBounds="True" Opacity="0.5"
     Width="{TemplateBinding Width}"
     Height="{TemplateBinding Height}">
        <Rectangle Name="rectMain"/>

        <ContentPresenter
         VerticalAlignment="Center"
         HorizontalAlignment="Center"/>
    </Grid>

    <!--  Behaviors.  -->
```

```
    <ControlTemplate.Triggers>
        ... Template triggers omitted here...
    </ControlTemplate.Triggers>
</ControlTemplate>
```

GlassButton

The `Template`'s controls are relatively simple. The `Template` contains a `Grid` that holds a `Rectangle` and the `ContentPresenter`. The `ContentPresenter`'s attributes center it on the `Button`, but all of the other interesting properties are set in the `Template`'s `Styles`.

Glass Button Styles

The following code shows the `Styles` defined in the template's Resources section. The `Button` looks differently when it is in the three states (normal, defaulted, and disabled). To make the code easier to understand, the template uses three different `Styles` for those states. The code also includes a base `Style` from which the others inherit.


```
<!--  Base style that sets corner radii and stroke thickness.  -->
<Style x:Key="styBase" TargetType="Rectangle">
    <Setter Property="RadiusX" Value="20"/>
    <Setter Property="RadiusY" Value="20"/>
    <Setter Property="StrokeThickness" Value="5"/>
</Style>

<!--  Style for "normal" status.  -->
<Style TargetType="Rectangle"
 BasedOn="{StaticResource styBase}">
    <Setter Property="Fill">
        <Setter.Value>
            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Color="DarkGreen" Offset="0"/>
                <GradientStop Color="LightGreen" Offset="1"/>
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
    <Setter Property="Stroke">
        <Setter.Value>
            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Color="DarkGreen" Offset="1"/>
                <GradientStop Color="LightGreen" Offset="0"/>
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
</Style>

<!--  Style when IsDefaulted.  -->
<Style x:Key="styIsDefaulted" TargetType="Rectangle"
 BasedOn="{StaticResource styBase}">
    <Setter Property="Fill">
        <Setter.Value>
            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Color="DarkGreen" Offset="0"/>
```

```
                <GradientStop Color="LightGreen" Offset="1"/>
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
    <Setter Property="Stroke">
        <Setter.Value>
            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Color="DarkGreen" Offset="1"/>
                <GradientStop Color="Black" Offset="0"/>
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
</Style>

<!--  Style when disabled.  -->
<Style x:Key="styDisabled" TargetType="Rectangle"
 BasedOn="{StaticResource styBase}">
    <Setter Property="Opacity" Value="0.75"/>
    <Setter Property="Fill">
        <Setter.Value>
            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Color="Gray" Offset="0"/>
                <GradientStop Color="White" Offset="1"/>
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
    <Setter Property="Stroke">
        <Setter.Value>
            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Color="Gray" Offset="1"/>
                <GradientStop Color="White" Offset="0"/>
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
</Style>
```

GlassButton

The first `Style` is a base `Style` that sets the `Rectangle`'s `RadiusX`, `RadiusY`, and `StrokeThickness` properties. These properties are the same for all of the `Button`'s states.

Since the `Button`'s "normal" `Style` is an unnamed `Rectangle Style`, it always applies unless some other `Style` overrides it. It sets the control's `Fill` property to a `LinearGradientBrush` that shades from dark green to light green. The `Style` then sets the `Stroke` property to a brush that does the opposite: It shades from light green to dark green. (This remarkably simple technique gives the `Button` an easy 3D appearance.)

The "defaulted" `Style` uses the same `Fill` property but makes the `Stroke` brush shade from dark green to black. (The difference is actually fairly subtle. You might want to experiment with larger changes, perhaps adding a completely black outline.)

The "disabled" `Style` sets the `Rectangle`'s `Opacity` property to 0.75, so it is translucent. It also changes the `Rectangle`'s `Fill` and `Stroke` properties to shade between gray and white.

Glass Button Triggers

The following code shows the `Template`'s `Triggers`. In response to events and changes in the control's properties, the `Triggers` set new property values and apply the `Styles`.


```
<!--  Mouse over.  -->
<Trigger Property="IsMouseOver" Value="True">
    <Setter TargetName="grdMain" Property="Opacity"
     Value="0.75"/>
</Trigger>

<!--  Focus.  -->
<Trigger Property="IsFocused" Value="True">
    <Setter TargetName="grdMain" Property="Opacity"
     Value="0.75"/>
</Trigger>

<!--  Defaulted.  -->
<Trigger Property="IsDefaulted" Value="True">
    <Setter TargetName="rectMain" Property="Style"
     Value="{StaticResource styIsDefaulted}"/>
</Trigger>

<!--  Pressed. This comes after Focus so it gets precedence.  -->
<Trigger Property="IsPressed" Value="True">
    <Setter TargetName="grdMain" Property="Opacity"
     Value="1"/>
</Trigger>

<!--  Disabled. This comes last so it gets ultimate precedence.  -->
<Trigger Property="IsEnabled" Value="False">
    <Setter TargetName="rectMain" Property="Style"
     Value="{StaticResource styDisabled}"/>
</Trigger>
```

GlassButton

When the control's `IsMouseOver` property is `True`, the first trigger sets the `Grid`'s `Opacity` property to 0.75. This is more opaque than the original value of 0.5, so the control becomes more solid.

When the control receives the focus, the second trigger also sets the `Grid`'s `Opacity` property to 0.75.

When the control's `IsDefaulted` property is `True`, the next trigger sets the `Rectangle`'s `Style` to the "defaulted" `Style`.

When the `IsPressed` property is `True`, the following trigger sets the `Grid`'s `Opacity` to 1, making it fully opaque.

Finally, when the control's `IsEnabled` property is `False`, the last trigger sets the `Rectangle`'s `Style` to the "disabled" `Style`. The `Button` control automatically stops interacting with the user, so you don't need to worry about that.

IMPORTANT ORDER

Notice that the order of the template's triggers is important. Triggers that are defined later are applied later — if two triggers are active at the same time, the second trigger overrides the first.

In this example, the `IsPressed` trigger must come after the `IsMouseOver` trigger. Otherwise, when the user pressed the mouse on the `Button`, the `IsPressed` trigger would occur first. But at that point, since the mouse would be over the `Button`, the `IsMouseOver` trigger would also apply and would override the `IsPressed` trigger so that the user would never see the `Button` look pressed.

Similarly the `IsEnabled` trigger comes last so it overrides all other triggers. If the button is disabled, it should never display any of the other appearances.

ELLIPSE BUTTON

The EllipseButton example program shown in Figure 15-9 uses a template to make an elliptical button that is very different from the glass button described in the preceding section.

FIGURE 15-9

The disabled button on the left is paler than the others and doesn't respond to the user.

The defaulted button is brighter than the others and displays a yellow highlight along its border.

Figure 15-10 shows the program when the mouse is over Button 3. The button under the mouse is even brighter than the defaulted button and displays a yellow glow under its text.

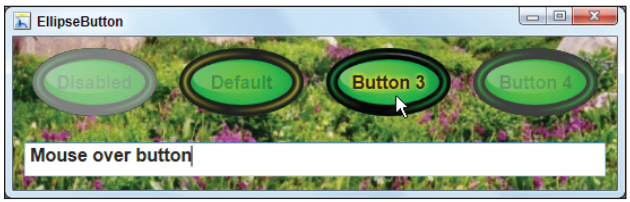

FIGURE 15-10

It's difficult to see, but the button under the mouse in Figure 15-10 also displays an extra white highlight in its border roughly above the number 3. Every second, that highlight makes a trip around the button's circumference to draw the user's attention to the button. It's a small highlight, so the effect is fairly subtle.

ANIMATION ADVICE

Motion is one of the most attention-grabbing effects you can add to a program, but it can also be the most distracting and annoying. A button that flashes bright colors or continually changes size while the mouse is over it would really annoy users. The moving highlight that the EllipseButton program displays is subtle so the effect isn't too bad, but be careful. Keep animations like this one subtle or make them play only once — for example, when the mouse first enters the button, so you don't drive your users crazy. In extreme cases, rapidly flashing lights can even induce seizures in some people so don't use areas that flash brightly, particularly at frequencies between 2 and 55 Hz. Better still, give users a way to disable these sorts of animations.

Also note that users with special needs such as color vision deficiency or visual impairment may not see subtle animations very well or at all. Don't rely solely on subtle animations to give the user information.

Figure 15-11 shows the program when the user presses the mouse down on a button.

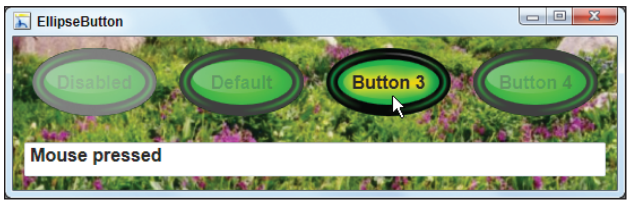

FIGURE 15-11

When you press a button, its background shifts slightly, and it displays a larger glow under its text.

If you drag the mouse off the button while it is still pressed, the button returns to its focused "mouse over" appearance. If you then release the mouse, no mouse click occurs.

The following two sections describe the ellipse button's `Template`. The first section describes the `Template` at a high level, explaining the controls the `Template` uses and how they fit together. The next section describes the `Template`'s `Triggers`.

This program is fairly long so the complete code isn't shown in these sections. You can download the example program from the book's web site to see the details.

Ellipse Button Controls

Figure 15-12 shows the template's control structure. The controls' labels are shown in left-to-right and top-to-bottom order, so you can tell which controls are defined by the XAML code before the others. For example, the "Inner surface" is defined in the XAML code before the "Outer edge."

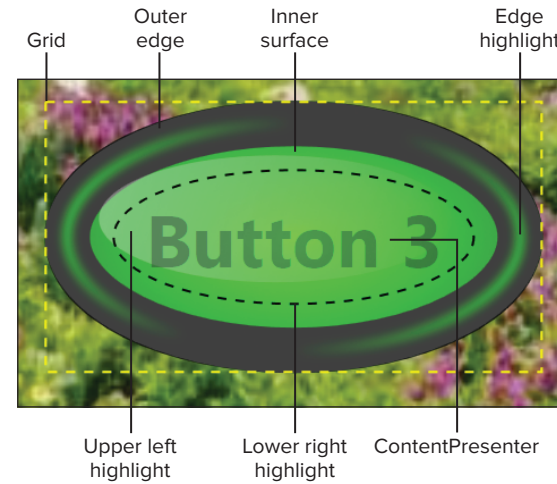

FIGURE 15-12

A `Grid` (shown as a yellow dashed box) contains all of the other controls, most of which are `Ellipses`.

The inner surface is an ellipse filled with a brush that shades from lime to green. It lies beneath all of the other visible controls.

The outer edge is an ellipse with a transparent center and a black edge. Because its `StrokeThickness` is 10, it forms a wide band around the control.

The edge highlight is an ellipse with `Margin = 4` and `StrokeThickness = 4`, forming a band within the outer edge. It has a transparent center. Its `Stroke` property is a gradient brush that shades from lime on the left, to transparent in the middle, to lime again on the right; thus this ellipse makes two highlights on the edge. Its `BitmapEffect` property is set to a `BlurBitmapEffect` object, so it's fuzzy, giving the edge a rounded, 3D appearance.

The sparkle highlight isn't shown in Figure 15-12. It is similar to the green edge highlight except that it's white and only has one visible piece instead of two. It is only visible when the mouse is over the control and it is animated.

The upper-left highlight is an ellipse filled with a brush that shades from white to transparent. Since its `Margin` property is set to `12,12,20,20`, it is offset a bit toward the upper left.

The lower-right highlight is shown in Figure 15-12 as a dashed ellipse because it has `Opacity = 0`, making in invisible. It is displayed when the user presses the button. If you look closely, you can see it in Figure 15-11. Like the upper-left highlight, this ellipse shades from white to transparent but is centered with a `Margin` value of 15.

The `ContentPresenter` is centered in the `Template`'s `Grid`.

The final `Template` control is a light gray `Ellipse` that covers everything. Its `Opacity` is 0.3 and thus it tones down the colors of all of the other controls.

Many of the controls have `Opacity` less than 1, so they are semitransparent. When the control changes state — for example, when the mouse is over the button or the user presses the button — the template's `Triggers` change the `Opacity` of the controls to give the `Button` a different appearance.

Ellipse Button Triggers

When events occur, the template's triggers make appropriate changes to the control's appearance. Mostly these changes involve changing `Opacity` values to make some controls more visible while hiding others.

The `IsMouseOver` property trigger shown in the following code is the most interesting of the template's triggers:


```
<Trigger Property="IsMouseOver" Value="True">
    <Setter TargetName="ellUpperLeftHighlight" Property="Opacity" Value="1"/>
    <Setter TargetName="ellCover" Property="Opacity" Value="0"/>
    <Setter TargetName="cpContent" Property="Opacity" Value="1"/>
    <Setter TargetName="cpContent" Property="BitmapEffect"
     Value="{StaticResource bmeMouseOver}"/>
    <Setter TargetName="ellSparkle" Property="Opacity" Value="0.75"/>

    <!--  Start the sparkle animation.  -->
    <Trigger.EnterActions>
        <BeginStoryboard Name="begSparkle">
            <Storyboard BeginTime="0:0:1" RepeatBehavior="Forever" >
                <DoubleAnimationUsingKeyFrames
                 Duration="0:0:2"
                 Storyboard.TargetName="transSparkle"
                 Storyboard.TargetProperty="Angle">
                    <LinearDoubleKeyFrame
                     Value="0" KeyTime="0:0:0"/>
                    <LinearDoubleKeyFrame
                     Value="360" KeyTime="0:0:1"/>
                    <LinearDoubleKeyFrame
                     Value="360" KeyTime="0:0:2"/>
                </DoubleAnimationUsingKeyFrames>
            </Storyboard>
        </BeginStoryboard>
    </Trigger.EnterActions>

    <!--  Stop the sparkle animation.  -->
    <Trigger.ExitActions>
        <StopStoryboard BeginStoryboardName="begSparkle"/>
    </Trigger.ExitActions>
</Trigger>
```

EllipseButton

This code uses simple setters to do the following immediately when it starts:

- Make the upper-left highlight fully opaque instead of translucent.
- Make the light gray cover that tones down the other controls transparent so that all of the other controls have their full brightness.
- Make the `ContentPresenter` fully opaque instead of translucent.

- Give the `ContentPresenter` an `OuterGlowBitmapEffect` (defined in the template's Resources section).
- Make the white sparkle highlight visible with `Opacity = 0.75`. This makes the left edge highlight brighter than the right edge highlight and prepares the sparkle for the animation described next.

The `Trigger`'s `EnterActions` occur when the `IsMouseOver` property becomes `True`. This code begins a `Storyboard` that uses a `DoubleAnimationUsingKeyFrames` object to animate the sparkle highlight's brush. The brush has a `RotateTransform` named `transSparkle` that initially has `Angle = 0`, so the brush is not rotated. The animation makes `Angle` sweep from 0 to 360 degrees over a 1-second period. It holds `Angle` at 360 degrees for another second to make the animation pause. The `Storyboard` then repeats indefinitely.

The `Trigger`'s `ExitActions` occur when the `IsMouseOver` property becomes no longer `True` (in other words, becomes `False`). When that happens, the code stops the `Storyboard` that animates the sparkle brush.

The `Template`'s other triggers are much simpler. For example, the following code shows the `IsPressed` property trigger that executes when the user presses the button:


```
<Trigger Property="IsPressed" Value="True">
    <Setter TargetName="ellUpperLeftHighlight" Property="Opacity" Value="0"/>
    <Setter TargetName="ellLowerRightHighlight" Property="Opacity" Value="0.75"/>
    <Setter TargetName="cpContent" Property="BitmapEffect"
     Value="{StaticResource bmePressed}"/>
</Trigger>
```

EllipseButton

This code uses simple setters to do the following:

- Hide the upper-left highlight by setting `Opacity = 0`.
- Display the lower-right highlight by setting `Opacity = 0.75`.
- Give the `ContentPresenter` the `OuterGlowBitmapEffect` named `bmePressed`. This effect, which is defined in the template's Resources section, uses a larger `GlowSize` than the `bmeMouseOver` effect, so the glow behind the `ContentPresenter` is larger. You can see the difference if you carefully compare Figures 15-10 and 15-11.

The template's other triggers shown in the following code work similarly:


```
<!--  Defaulted.  -->
<Trigger Property="IsDefaulted" Value="True">
    <Setter TargetName="ellCover" Property="Opacity" Value="0.15"/>
    <Setter TargetName="ellEdgeHighlight" Property="Stroke"
     Value="{StaticResource brDefaultedEdgeHighlight}"/>
</Trigger>

<!--  Not defaulted.  -->
```

```
<Trigger Property="IsDefaulted" Value="False">
    <Setter TargetName="ellEdgeHighlight" Property="Stroke"
     Value="{StaticResource brEdgeHighlight}"/>
</Trigger>

<!--  Disabled. This comes last so it gets ultimate precedence.  -->
<Trigger Property="IsEnabled" Value="False">
    <Setter TargetName="ellCover" Property="Opacity" Value="0.6"/>
</Trigger>
```

EllipseButton

These triggers are fairly straightforward, giving controls new `Stroke` values and shuffling around `Opacity` values.

TEMPLATE TRICKS 3

The EllipseButton example demonstrates several useful template tricks including:

- Using `BlurBitmapEffect` to make an object appear three-dimensional (the outer edge with the blurred edge highlight on it)
- Using `Opacity` <1 to make translucent highlights
- Using brushes that blend from a color to transparent to make highlights fade away
- Using brushes that blend from a color to transparent and back to a color to make objects that are visible in multiple places
- Using different brushes for different control states (the normal vs. defaulted highlights)
- Using different `BitmapEffects` for different control states (the smaller glow for `IsMouseOver` and the larger glow for `IsPressed`)
- Animating changes to a brush's transformation

RESEARCHING CONTROL TEMPLATES

To effectively build templates, you need to learn what behaviors the control provides for you and what behaviors you need to provide for it. You also need to determine what events the control provides so that you know when you have a chance to make the control take action.

For example, WPF provides a confusing assortment of mouse events including `Mouse.MouseEnter`, `IsMouseOver`, `MouseLeftButtonDown`, `Pressed`, and `Click`. If you're trying to write a `Button` template, which mouse events can you use to change the `Button`'s appearance? Which properties and behaviors does the `Button` provide for you, and which do you need to implement?

The Button templates described in the previous sections use the Button's IsMouseOver, IsPressed, IsEnabled, IsFocused, and IsDefaulted properties. The Button class provides these no matter what controls you add to the template to provide basic Button behavior.

As described in the "Template Binding" section earlier in this chapter, templates can also read some of the property values provided by the underlying control. For example, the Button class provides Background, BorderBrush, and BorderThickness properties that a template can read by using template bindings. Button also inherits properties such as Width and Height that you can also read with template bindings.

So, how do you learn what properties and template bindings are available?

One good source of information is Microsoft's "Control Styles and Templates" web page at msdn.microsoft.com/cc278075(VS.95).aspx. That page provides links to other pages that describe the features available to different control templates.

For example, the "Button Styles and Templates" page lists the Button's states and properties and tells where it gets them. For instance, the Pressed state (which you can read with the IsPressed property) tells when the button is pressed.

These web pages also show the default templates used by the controls. The Button control's default template is 84 lines long and fairly complicated. Some are much longer and much more complex.

In addition to using Microsoft's web pages, you can make a control tell you about its template. The ShowTemplate example program shown in Figure 15-13 displays the default template for a control. When you click on the "Show Template" button, the program displays the default template for the control named Target. In Figure 15-13, that control is the Slider in the upper-left corner. To see the template used by a different kind of control, replace the Slider with a different control, name it Target, and run the program.

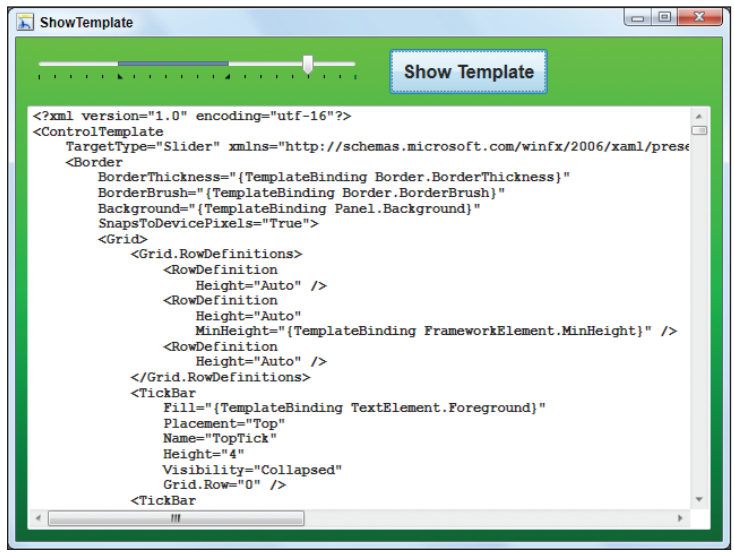

FIGURE 15-13

The ShowTemplate program uses the following code to display the `Target` control's template:


```
private void btnShowTemplate_Click(object sender, RoutedEventArgs e)
{
    XmlWriterSettings writer_settings = new XmlWriterSettings();
    writer_settings.Indent = true;
    writer_settings.IndentChars = "    ";
    writer_settings.NewLineOnAttributes = true;

    StringBuilder sb = new StringBuilder();
    XmlWriter xml_writer = XmlWriter.Create(sb, writer_settings);

    XamlWriter.Save(Target.Template, xml_writer);

    txtResult.Text = sb.ToString();
}
```

ShowTemplate

The key to this code is the `XamlWriter` class, which includes methods that extract XAML from a WPF object such as a control or template.

The code starts by initializing an `XmlWriterSettings` object to make the writer produce nicely formatted output. If you don't do this, the code comes out in one long line of XML without carriage returns or indentation.

The program then creates a `StringBuilder` to hold the result text. It uses the writer settings to create an `XmlWriter` attached to the `StringBuilder`.

The code then calls the `XamlWriter` class's static `Save` method to write a XAML representation of the `Target` control's `Template` property into the `StringBuilder`.

The program finishes by displaying the result in its `TextBox txtResult`.

After you find a control's default template, you can modify it to make your own template that changes the control's appearance. That lets you ensure that your template behaves the same way the default template does except in those places where you want changes.

SUMMARY

Properties and styles let you change a control's appearance in superficial ways. Templates let you change a control more fundamentally, altering the pieces that make up the control and changing the way it responds to events and changes in property values. By using templates, you can give your applications a distinctive look and feel.

The next chapter explains two topics closely related to styles and templates: themes and skins. Skins let an application change its entire appearance, sometimes radically. They let you change the application to suit your immediate need or even your mood.

Themes provide a unifying appearance across controls, windows, and even separate applications. By providing a common look and feel, themes can make even unrelated programs seem to fit together in a single system.