And here's the code at the end of the DispatcherTimer.Tick event handler, which checks whether an adjustment is needed and makes the appropriate changes:

```
    ...
    // Perform an "adjustment" when needed.
    if ((DateTime.Now.Subtract(lastAdjustmentTime).TotalSeconds >
      secondsBetweenAdjustments))
    {
        lastAdjustmentTime = DateTime.Now;

        secondsBetweenBombs -= secondsBetweenBombsReduction;
        secondsToFall -= secondsToFallReduction;

        // (Technically, you should check for 0 or negative values.
        // However, in practice these won't occur because the game will
        // always end first.)

        // Set the timer to drop the next bomb at the appropriate time.
        bombTimer.Interval = TimeSpan.FromSeconds(secondsBetweenBombs);

        // Update the status message.
        lblRate.Text = String.Format("A bomb is released every {0} seconds.",
          secondsBetweenBombs);
        lblSpeed.Text = String.Format("Each bomb takes {0} seconds to fall.",
          secondsToFall);
    }
}
```

With this code in place, there's enough functionality to drop bombs at an ever-increasing rate. However, the game still lacks the code that responds to dropped and saved bombs.

Intercepting a Bomb

The user saves a bomb by clicking it before it reaches the bottom of the Canvas. Because each bomb is a separate instance of the Bomb user control, intercepting mouse clicks is easy—all you need to do is handle the MouseLeftButtonDown event, which fires when any part of the bomb is clicked (but doesn't fire if you click somewhere in the background, such as around the edges of the bomb circle).

When a bomb is clicked, the first step is to get the appropriate bomb object and set its IsFalling property to indicate that it's no longer falling. (The IsFalling property is used by the event handler that deals with completed animations.)

```
private void bomb_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    // Get the bomb.
    Bomb bomb = (Bomb)sender;
    bomb.IsFalling = false;

    // Record the bomb's current (animated) position.
    double currentTop = Canvas.GetTop(bomb);
    ...
```

The next step is to find the storyboard that controls the animation for this bomb so it can be stopped. To find the storyboard, you need to look it up in the collection that this game uses for tracking. Currently, WPF doesn't include any standardized way to find the animations that are acting on a given element.

```
    ...
    // Stop the bomb from falling.
    Storyboard storyboard = storyboards[bomb];
    storyboard.Stop();
    ...
```

After a button is clicked, another set of animations moves the bomb off the screen, throwing it up and left or right (depending on which side is closest). Although you could create an entirely new storyboard to implement this effect, the BombDropper game clears the current storyboard that's being used for the bomb and adds new animations to it. When this process is completed, the new storyboard is started:

```
    ...
    // Reuse the existing storyboard, but with new animations.
    // Send the bomb on a new trajectory by animating Canvas.Top
    // and Canvas.Left.
    storyboard.Children.Clear();

    DoubleAnimation riseAnimation = new DoubleAnimation();
    riseAnimation.From = currentTop;
    riseAnimation.To = 0;
    riseAnimation.Duration = TimeSpan.FromSeconds(2);

    Storyboard.SetTarget(riseAnimation, bomb);
    Storyboard.SetTargetProperty(riseAnimation, new PropertyPath("(Canvas.Top)"));
    storyboard.Children.Add(riseAnimation);

    DoubleAnimation slideAnimation = new DoubleAnimation();
    double currentLeft = Canvas.GetLeft(bomb);

    // Throw the bomb off the closest side.
    if (currentLeft < canvasBackground.ActualWidth / 2)
    {
        slideAnimation.To = -100;
    }
    else
    {
        slideAnimation.To = canvasBackground.ActualWidth + 100;
    }
    slideAnimation.Duration = TimeSpan.FromSeconds(1);
    Storyboard.SetTarget(slideAnimation, bomb);
    Storyboard.SetTargetProperty(slideAnimation, new PropertyPath("(Canvas.Left)"));
    storyboard.Children.Add(slideAnimation);

    // Start the new animation.
    storyboard.Duration = slideAnimation.Duration;
    storyboard.Begin();
}
```

Now the game has enough code to drop bombs and bounce them off the screen when the user saves them. However, to keep track of what bombs are saved and which ones are dropped, you need to react to the Storyboard.Completed event that fires at the end of an animation.

Counting Bombs and Cleaning Up

As you've seen, the BombDropper uses storyboards in two ways: to animate a falling bomb and to animate a defused bomb. You could handle the completion of these storyboards with different event handlers, but to keep things simple, the BombDropper uses just one. It tells the difference between an exploded bomb and a rescued bomb by examining the Bomb.IsFalling property.

```
// End the game when 5 bombs have fallen.
private int maxDropped = 5;

private void storyboard_Completed(object sender, EventArgs e)
{
    ClockGroup clockGroup = (ClockGroup)sender;

    // Get the first animation in the storyboard, and use it to find the
    // bomb that's being animated.
    DoubleAnimation completedAnimation =
      (DoubleAnimation)clockGroup.Children[0].Timeline;
    Bomb completedBomb = (Bomb)Storyboard.GetTarget(completedAnimation);

    // Determine if a bomb fell or flew off the Canvas after being clicked.
    if (completedBomb.IsFalling)
    {
        droppedCount++;
    }
    else
    {
        savedCount++;
    }
    ...
```

Either way, the code then updates the display test to indicate how many bombs have been dropped and saved:

```
    ...
        // Update the display.
    lblStatus.Text = String.Format("You have dropped {0} bombs and saved {1}.",
      droppedCount, savedCount);
    ...
```

At this point, the code checks to see whether the maximum number of dropped bombs has been reached. If it has, the game ends, the timer is stopped, and all the bombs and storyboards are removed:

```
    ...
    // Check if it's game over.
    if (droppedCount >= maxDropped)
    {
        bombTimer.Stop();
        lblStatus.Text += "\r\n\r\nGame over.";

        // Find all the storyboards that are underway.
        foreach (KeyValuePair<Bomb, Storyboard> item in storyboards)
        {
            Storyboard storyboard = item.Value;
            Bomb bomb = item.Key;

            storyboard.Stop();
            canvasBackground.Children.Remove(bomb);
        }

        // Empty the tracking collection.
        storyboards.Clear();

        // Allow the user to start a new game.
        cmdStart.IsEnabled = true;
    }
    else
    {
        // Clean up just this bomb, and let the game continue.
        Storyboard storyboard = (Storyboard)clockGroup.Timeline;
        storyboard.Stop();

        storyboards.Remove(completedBomb);
        canvasBackground.Children.Remove(completedBomb);
    }
}
```

This completes the code for BombDropper game. However, you can make plenty of refinements. Some examples include the following:

- **Animate a bomb explosion effect.** This effect can make the flames around the bomb twinkle or send small pieces of shrapnel flying across the Canvas.
- **Animate the background.** This change is easy, and it adds pizzazz. For example, you can create a linear gradient that shifts up, creating an impression of movement, or one that transitions between two colors.
- **Add depth.** It's easier than you think. The basic technique is to give the bombs different sizes. Bombs that are bigger should have a higher ZIndex, ensuring that they overlap smaller bombs, and should be given a shorter animation time, ensuring that they fall faster. You can also make the bombs partially transparent, so as one falls, the others behind it are visible.
- **Add sound effects.** In Chapter 26, you'll learn to use sound and other media in WPF. You can use well-timed sound effects to punctuate bomb explosions or rescued bombs.

- **Use animation easing.** If you want bombs to accelerate as they fall, bounce off the screen, or wiggle more naturally, you can add easing functions to the animations used here. And, as you'd expect, easing functions can be constructed in code just as easily as in XAML.
- **Fine-tune the parameters.** You can provide more dials to tweak behavior (for example, variables that set how the bomb times, trajectories, and frequencies are altered as the game processes). You can also inject more randomness (for example, allowing saved bombs to bounce off the Canvas in slightly different ways).

The Last Word

In this chapter, you learned the techniques needed to make practical animations and integrate them into your applications. The only missing ingredient is the eye candy—in other words, making sure the animated effects are as polished as your code.

As you've seen over the past two chapters, the animation model in WPF is surprisingly full-featured. However, getting the result you want isn't always easy. If you want to animate separate portions of your interface as part of a single animated "scene," you're often forced to write a fair bit of markup with interdependent details that aren't always clear. In more complex animations, you may be forced to hard-code details and fall back to code to perform calculations for the ending value of animation. And if you need fine-grained control over an animation, such as when modeling a physical particle system, you'll need to control every step of the way using frame-based animation.

The future of WPF animation promises higher-level classes that are built on the basic plumbing you've learned about in this chapter. Ideally, you'll be able to plug animations into your application simply by using prebuilt animation classes, wrapping your elements in specialized containers, and setting a few attached properties. The actual implementation that generates the effect you want—whether it's a smooth dissolve between two images or a series of animated fly-ins that builds a window—will be provided for you.

CHAPTER 17

■ ■ ■

Control Templates

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

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

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

■ **What's New** WPF 4 adds a new *visual state* model to help you restyle controls more easily. This model was originally introduced by WPF's "little brother," the browser-based application development platform Silverlight 3. However, the visual state model hasn't been fully incorporated into the WPF world in this release. Although it's there for you to use when you design your own controls (see Chapter 18), the standard set of WPF controls doesn't yet support it. For a high-level discussion of the visual state manager, see the "Visual States" section later in this chapter.

Understanding Logical Trees and Visual Trees

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

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

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

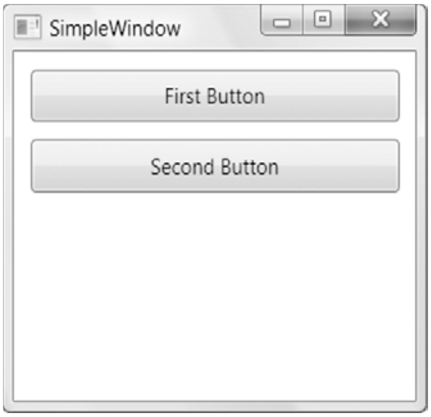


Figure 17-1. A window with three elements

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

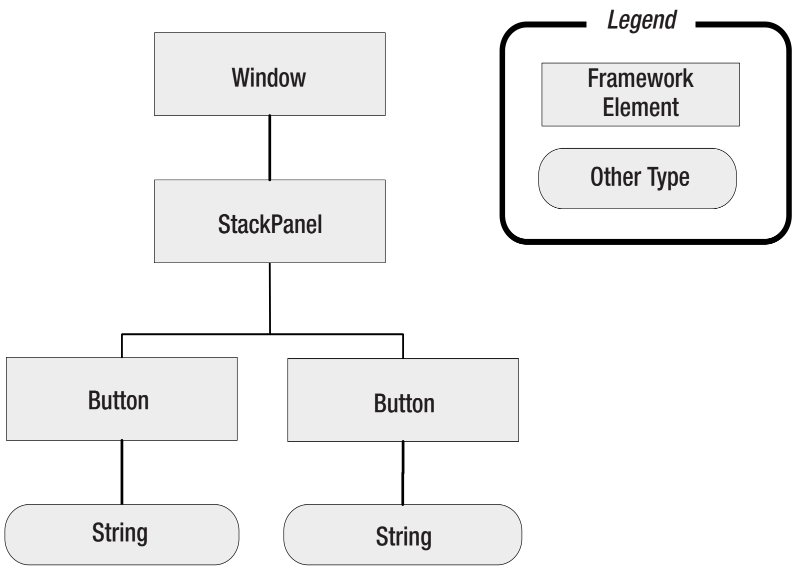


Figure 17-2. *The logical tree for SimpleWindow*

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

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

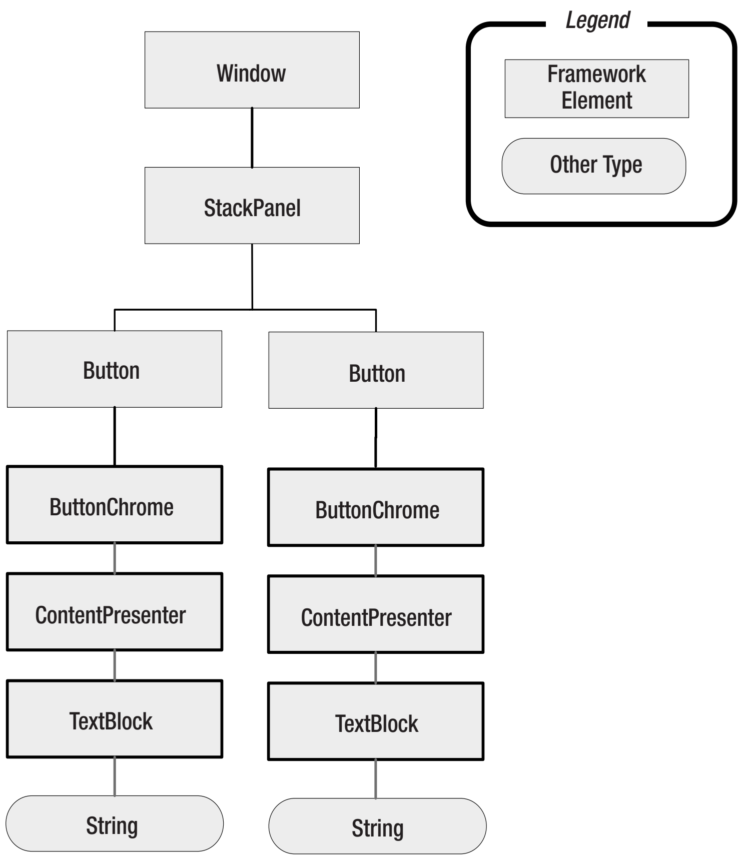


Figure 17-3. *The visual tree for SimpleWindow*

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

■ **Note** It's important to realize that there is more than one possible way to expand a logical tree into a visual tree. Details like the styles you've used, the properties you've set, your operating system (Windows XP or Windows 7/Vista), and your current Windows theme can affect the way a visual tree is composed. For instance, in the previous example, the button holds text content, and as a result, it automatically creates a nested TextBlock element. But as you know, the Button control is a content control, so it can hold any other element you want to use, as long as you nest it inside the button.

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

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

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

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

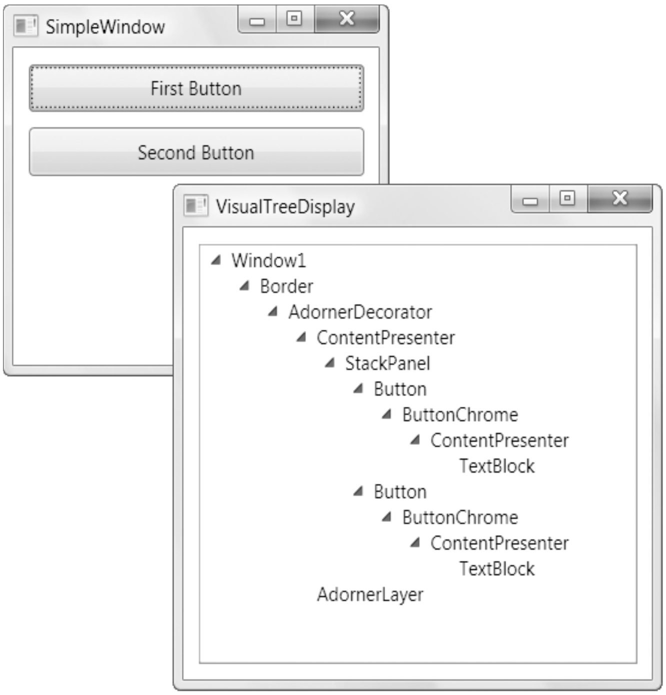
Table 17-1. *LogicalTreeHelper Methods*

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

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

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

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



```
◢ Window1
    ◢ Border
        ◢ AdornerDecorator
            ◢ ContentPresenter
                ◢ StackPanel
                    ◢ Button
                        ◢ ButtonChrome
                            ◢ ContentPresenter
                                TextBlock
                    ◢ Button
                        ◢ ButtonChrome
                            ◢ ContentPresenter
                                TextBlock
            AdornerLayer
```

Figure 17-4. Programmatically examining the visual tree

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

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

Here's the complete code for the VisualTreeDisplay window:

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

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

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

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

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

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

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

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

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

Understanding Templates

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Types of Templates

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

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

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

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

The Chrome Classes

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

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

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

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

```
<ControlTemplate TargetType="{x:Type ResizeGrip}" ... >
  <Grid Background="{TemplateBinding Panel.Background}" SnapsToDevicePixels="True">
    <Path Margin="0,0,2,2" Data="M9,0L11,0 11,11 0,11 0,9 3,9 3,6 6,6 6,3 9,3z"
     HorizontalAlignment="Right" VerticalAlignment="Bottom">
      <Path.Fill>
        <DrawingBrush ViewboxUnits="Absolute" TileMode="Tile" Viewbox="0,0,3,3"
```

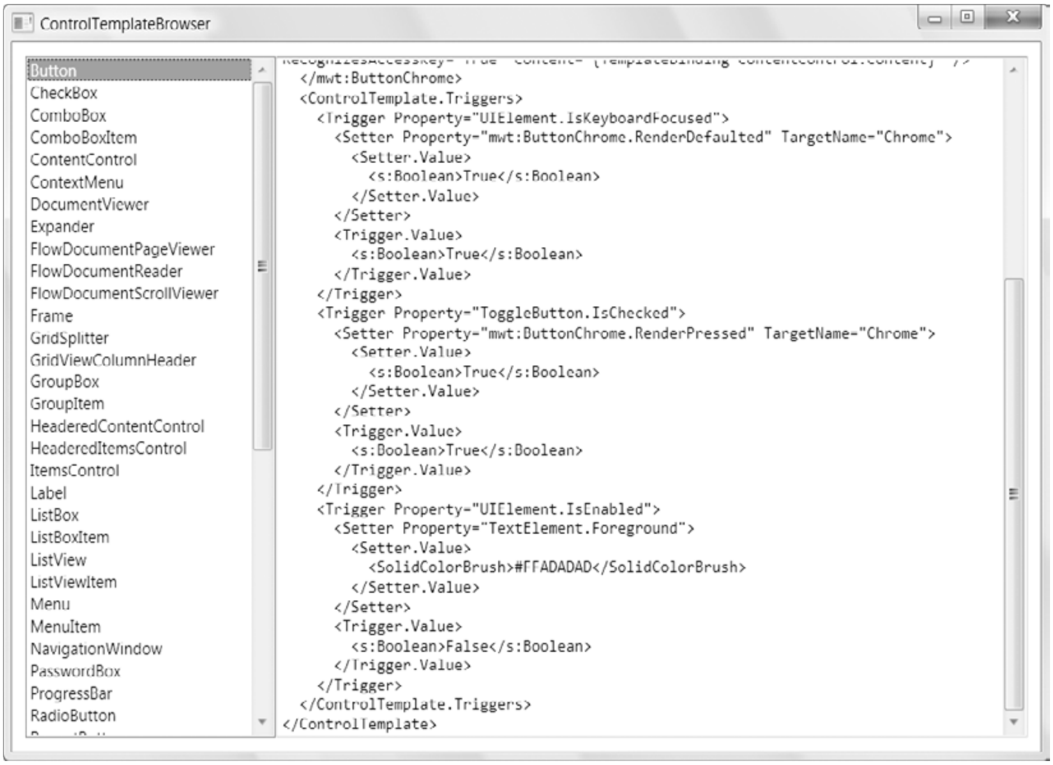
```
          Viewport="0,0,3,3" ViewportUnits="Absolute">
           <DrawingBrush.Drawing>
             <DrawingGroup>
               <DrawingGroup.Children>
                 <GeometryDrawing Geometry="M0,0L2,0 2,2 0,2z">
                   <GeometryDrawing.Brush>
                     <LinearGradientBrush EndPoint="1,0.75" StartPoint="0,0.25">
                       <LinearGradientBrush.GradientStops>
                         <GradientStop Offset="0.3" Color="#FFFFFFFF" />
                         <GradientStop Offset="0.75" Color="#FFBBC5D7" />
                         <GradientStop Offset="1" Color="#FF6D83A9" />
                       </LinearGradientBrush.GradientStops>
                     </LinearGradientBrush>
                   </GeometryDrawing.Brush>
                 </GeometryDrawing>
               </DrawingGroup.Children>
             </DrawingGroup>
           </DrawingBrush.Drawing>
         </DrawingBrush>
       </Path.Fill>
     </Path>
  </Grid>
</ControlTemplate>
```

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

Dissecting Controls

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

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



Button
CheckBox
ComboBox
ComboBoxItem
ContentControl
ContextMenu
DocumentViewer
Expander
FlowDocumentPageViewer
FlowDocumentReader
FlowDocumentScrollViewer
Frame
GridSplitter
GridViewColumnHeader
GroupBox
GroupItem
HeaderedContentControl
HeaderedItemsControl
ItemsControl
Label
ListBox
ListBoxItem
ListView
ListViewItem
Menu
MenuItem
NavigationWindow
PasswordBox
ProgressBar
RadioButton

```
RecognizesAccessKey="True" Content="{TemplateBinding ContentControl.Content}" />
  </mwt:ButtonChrome>
  <ControlTemplate.Triggers>
    <Trigger Property="UIElement.IsKeyboardFocused">
      <Setter Property="mwt:ButtonChrome.RenderDefaulted" TargetName="Chrome">
        <Setter.Value>
          <s:Boolean>True</s:Boolean>
        </Setter.Value>
      </Setter>
      <Trigger.Value>
        <s:Boolean>True</s:Boolean>
      </Trigger.Value>
    </Trigger>
    <Trigger Property="ToggleButton.IsChecked">
      <Setter Property="mwt:ButtonChrome.RenderPressed" TargetName="Chrome">
        <Setter.Value>
          <s:Boolean>True</s:Boolean>
        </Setter.Value>
      </Setter>
      <Trigger.Value>
        <s:Boolean>True</s:Boolean>
      </Trigger.Value>
    </Trigger>
    <Trigger Property="UIElement.IsEnabled">
      <Setter Property="TextElement.Foreground">
        <Setter.Value>
          <SolidColorBrush>#FFADADAD</SolidColorBrush>
        </Setter.Value>
      </Setter>
      <Trigger.Value>
        <s:Boolean>False</s:Boolean>
      </Trigger.Value>
    </Trigger>
  </ControlTemplate.Triggers>
</ControlTemplate>
```

Figure 17-5. Browsing WPF control templates

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Creating Control Templates

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

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

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

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