16

Themes and Skins

WPF lets you build applications that have engaging, distinctive appearances. By using relatively simple techniques such as drop shadows, partial transparency, and transparency masks, you can make an eye-catching interface that adds interest and excitement to even the most routine application.

Properties let you change the appearance of controls. They let you change visual characteristics such as a control's colors, size, location, and contents. Resources and styles let you package those changes so that you can easily apply them to many controls simultaneously.

Templates let you alter the way controls behave by changing the pieces that make up the controls. They let you change the appearance and behavior of `Buttons`, `ListBoxes`, `Menus`, and other controls in fundamental ways while still allowing them to perform their essential functions.

Themes and skins bring all of these ideas together to let you easily change the appearance and behavior of an entire application to suit the users' needs and moods.

THEMES

Some developers use the terms *theme* and *skin* interchangeably, but I make the distinction that a *skin* applies to a single application (or part of an application), and a *theme* applies to more than one application.

More precisely, a *theme* is a unifying plan that helps determine the appearance and behavior of more than one application. The most common themes are those provided by Windows. For example, Windows 7 provides the themes:

- Aero
- Classic
- Luna (Homestead, Metallic, and Normal versions)
- Royale

Using the System Theme

The ShowThemes example program shown in Figure 16-1 displays controls that use each of the themes that come with Windows 7. The differences are fairly subtle, so you'll need to look closely to see the changes in each theme.

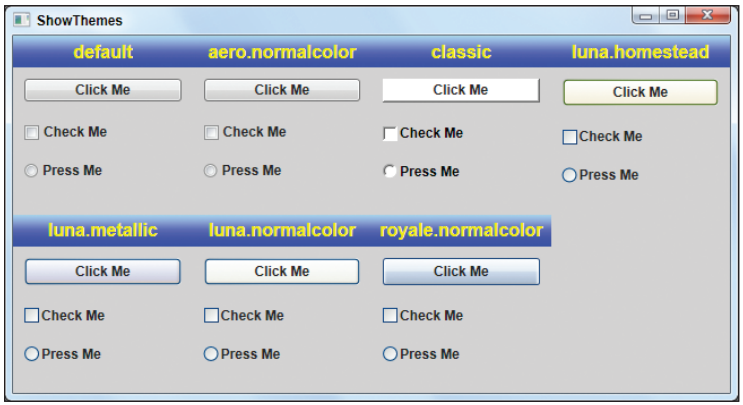

FIGURE 16-1

If you don't use properties, styles, or templates to change a control's appearance or behaviors, it uses the values defined by the system's current theme. In Figure 16-1, you can see that the controls in the "default" group have the same appearance as those that use the Aero theme. This is because the system had the Aero theme selected when I ran the program.

To change the system's theme in Windows 7, open the Control Panel. Under "Appearance and Personalization" click "Change the theme." On the dialog shown in Figure 16-2, click the theme that you want to use.

After taking the screenshot shown in Figure 16-1, I followed these steps to change the system's theme to Windows Classic without closing the ShowThemes program. The program automatically detected the change in the system theme and redrew itself appropriately.

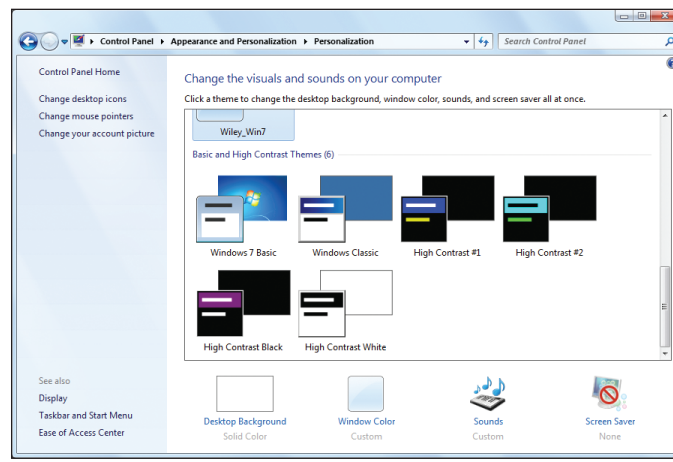

FIGURE 16-2

Figure 16-3 shows the result. If you look closely, you'll see that the controls in the default group now match those that use the Classic theme.

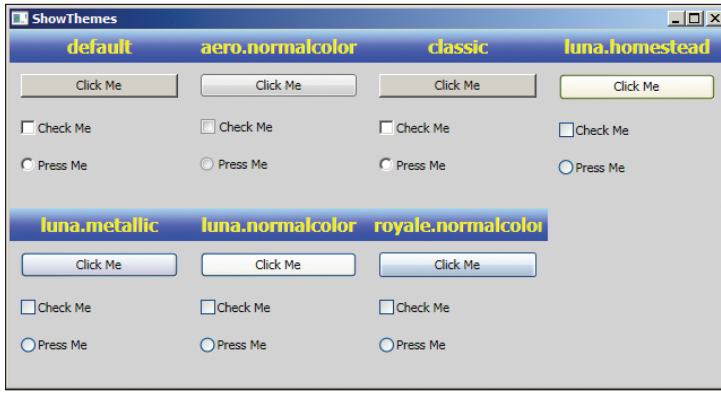

FIGURE 16-3

If you compare Figures 16-1 and 16-3 very closely, you'll also see that the controls in the ShowThemes program look a little different. In Figure 16-3, the window's upper corners are not as rounded, the title bar is darker, the border is no longer shaded with a light blue pattern, and the system icons in the form's upper-left and upper-right corners are different.

Using a Specific Theme

Normally you don't need to even think about themes. If you leave a control's appearance alone, it will automatically use the system's current theme and even change its appearance if you change the theme. If you really want to, however, you can select a specific theme.

CHANGING THEMES

This technique is probably more useful for testing an application to see what it looks like in a particular theme than it is for actually forcing the theme on the users.

Some users may select a particular theme for a good reason. For example, a visually impaired user may select a high-contrast theme to make programs easier to see. If you change the theme, that user may be unable to use your application.

If you don't really *need* a specific theme, you should let your program use the default.

To use a specific system theme in Visual Studio, begin a new WPF project. Open the Project menu and select "Add Reference." On the .NET tab, select the theme(s) that you want to use and click OK. Figure 16-4 shows the Add Reference dialog with the Aero, Classic, Luna, and Royale themes selected.

Next, in a resource dictionary, use a `MergedDictionaries` object to load the theme. The theme will apply to any controls that should be modified by the dictionary.

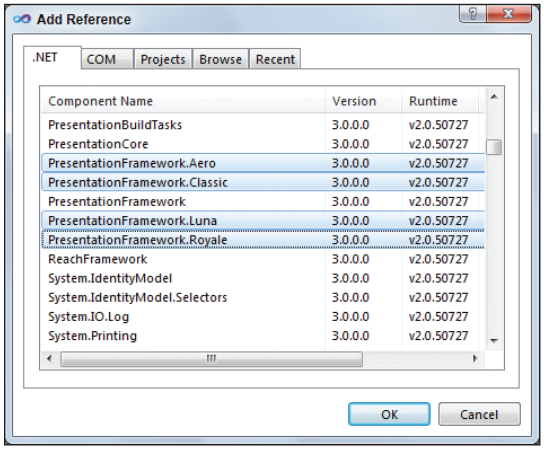

FIGURE 16-4

For example, the following code shows how the ShowThemes program uses the Luna Metallic theme. Each group of controls shown in Figures 16-1 and 16-3 is contained in a `StackPanel`. Each `StackPanel` has a `ResourceDictionary` that loads its theme.

Available for download on Wrox.com

```
<StackPanel>
    <StackPanel.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source=
"/PresentationFramework.Luna;component/themes/luna.metallic.xaml" />
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </StackPanel.Resources>
    <Label Style="{StaticResource lblStyle}" Content="luna.metallic"/>
    <Button Margin="10" Content="Click Me"/>
    <CheckBox Margin="10" Content="Check Me"/>
    <RadioButton Margin="10" Content="Press Me"/>
</StackPanel>
```

ShowThemes

Notice the unusual syntax for the `ResourceDictionary`'s `Source` property. The PresentationFramework.Luna piece tells WPF which library contains the theme, and the rest of the `Source` gives the name of the theme within the library.

For more information on Microsoft's standard themes, go to `msdn.microsoft.com/aa358533.aspx`. Links at the bottom of the web page lead to pages about the specific themes Aero, Classic, Luna, and Royale. From those pages, you can download XAML files that show how the themes are defined. You can then modify those files to build your own theme files.

THEME RESTRICTIONS

I have had bad luck getting Expression Blend to use specific themes. It seems to have trouble finding the DLLs for use by the `ResourceDictionary`'s `Source` property. I've also had bad luck getting the compiled executable to run.

Perhaps these issues will be fixed in a later release, but for now I use this technique only to see what the program will look like in different themes in programs built with Visual Studio. If you figure out how to get these working in Expression Blend, e-mail me at `RodStephens@vb-helper.com` and I'll post your solutions on the book's web page.

SKINS

Themes let a program automatically change to match the rest of the system's appearance. Selecting a specific theme lets a program change its appearance deliberately, but that's generally not necessary. The differences between the Luna Metallic and Aero Normal themes are so small that there's little reason to force the user to see one or the other when you could let the program use the system's default theme.

Skins are much more interesting. A *skin* is a packaged set of appearances and behaviors that can give an application (or part of an application) a distinctive appearance while still allowing it to provide its key features.

Skins are somewhat similar to themes in the sense that they define the appearance and behavior of an application, but they generally make much larger changes in the application's appearance than those shown in Figures 16-1 and 16-3. Rather than unifying all of the applications running on a system, the larger changes provided by skins can make an application stand out. A skin differentiates the application and makes it easier for the user to tell applications apart at a glance.

For example, Figures 16-5 and 16-6 show the ResourceDictionaries example program (which is described in Chapter 12) displaying two very different appearances. It's the same program in both figures and it contains the same controls — just rearranged slightly and with different colors, fonts, and so forth.

FIGURE 16-5

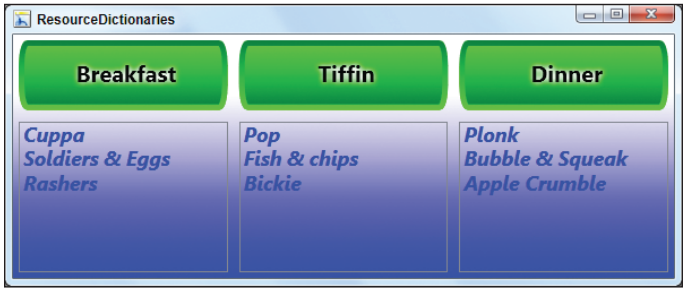

FIGURE 16-6

The following sections describe skins and explain several ways you can implement them in WPF.

HARD WORK WARNING

Be warned that skinning takes a lot of work! Depending on the technique you use, it may not be very complicated work, but it can be very time-consuming.

WPF provides so many tools for creating attractive user interfaces that it's easy to spend hours fiddling with control properties and arrangements, trying to build the world's most beautiful interface. Now multiply that effort to provide multiple skins, and you could end up spending days on a window instead of "only" hours.

Skin Purposes

Usually skins are mostly decorative, changing the application's colors, button shapes, form designs, background images, and so forth. The skins shown in Figures 16-5 and 16-6 look very different but only superficially. They still use the same controls in roughly the same positions.

Although skins are often decorative and used to increase a program's "coolness factor," multiple skins can have legitimate business purposes.

For example, in the United States, roughly 8 percent of men and 0.4 percent of women have some form of color vision deficiency and thus have trouble distinguishing among certain colors. If your application provides multiple skins, users can change the colors or shapes used by the program so they have less trouble getting the information they need.

In addition, as the general user population ages, applications must be ready to help older users. Larger fonts, menus, buttons, and other components can make understanding and using an application easier for users. Providing multiple skins with different element sizes also allows users with larger screens to take advantage of the space they have available.

One use for skins that is usually overlooked is to make different interfaces so you can use the same application for different purposes. For example, suppose you're writing an order entry system. Different kinds of users would need to see different pieces of an order at different times.

When an order is initially created, the order entry clerk needs to know all about the customer and order, and possibly payment information (depending on your arrangements with the customers). Later, the shipping clerk who packages up the customer's order only needs to know about the items ordered and the customer's shipping address, not payment information or previous order history. The program might automatically send the customer an invoice, but if the customer calls with a question, a billing clerk may need to know about the customer's payment method and possibly past orders.

You can use different skins to satisfy the needs of these different users. The order entry clerk's skin would let the user locate customer data and enter information about a new order. The shipping clerk's skin would display information about the current order and the customer's shipping address while hiding payment information and previous order history.

The OrderTracking example program displays four different interfaces for different kinds of users. Figure 16-7 shows the program's four skins for managers, billing clerks, order entry clerks, and shipping clerks.

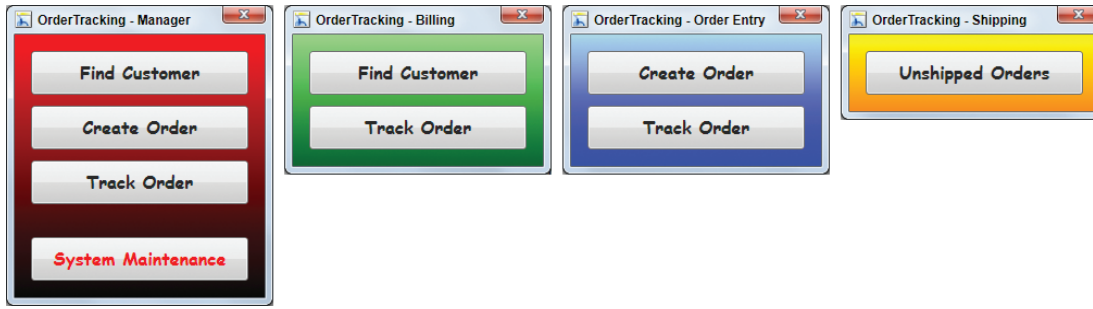

FIGURE 16-7

The following XAML code shows how the OrderTracking program works:


```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="Window1"
    x:Name="Window"
    SizeToContent="WidthAndHeight"
    Width="300" Height="360"
    ResizeMode="NoResize">

    <!--  Load skin resources  -->
    <Window.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="resBasics.xaml"/>
                <!--
                <ResourceDictionary Source="resBillingClerk.xaml"/>
                <ResourceDictionary Source="resOrderEntry.xaml"/>
                <ResourceDictionary Source="resShippingClerk.xaml"/>
                 -->
                <ResourceDictionary Source="resManager.xaml"/>
```

```
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Window.Resources>

    <!--  Set window properties from resources  -->
    <Window.Background>
        <StaticResource ResourceKey="brWindow"/>
    </Window.Background>
    <Window.FontFamily>
        <StaticResource ResourceKey="ffWindow"/>
    </Window.FontFamily>
    <Window.FontSize>
        <StaticResource ResourceKey="fsWindow"/>
    </Window.FontSize>
    <Window.FontWeight>
        <StaticResource ResourceKey="fwWindow"/>
    </Window.FontWeight>
    <Window.Title>
        <StaticResource ResourceKey="txtTitle"/>
    </Window.Title>

    <StackPanel Margin="10">
        <Button Content="Unshipped Orders" Click="btnUnshippedOrders_Click"
         Visibility="{StaticResource visUnshippedOrder}"/>
        <Button Content="Find Customer" Click="btnFindCustomer_Click"
         Visibility="{StaticResource visFindCustomer}"/>
        <Button Content="New Order" Click="btnNewOrder_Click"
         Visibility="{StaticResource visCreateOrder}"/>
        <Button Content="Track Order" Click="btnTrackOrder_Click"
         Visibility="{StaticResource visTrackOrder}"/>

        <Label Height="30"
         Visibility="{StaticResource visSystemMaintenance}"/>
        <Button Content="System Maintenance" Click="btnSystemMaintenance_Click"
         Foreground="Red" Height="40"
         Visibility="{StaticResource visSystemMaintenance}"/>
    </StackPanel>
</Window>
```

OrderTracking

The program begins by defining `Window` attributes. Setting the `SizeToContent` attribute to `WidthAndHeight` makes the window automatically resize itself to fit its content so the window is an appropriate size no matter which skin it is using. The code also sets the `ResizeMode` attribute to `NoResize` so the window stays that size.

Next, the code loads its resource dictionaries. The first one, resBasics.xaml, contains values that are the same for every skin. It defines the window's font properties and contains an unnamed `Button` style that sets the `Button` sizes and margins.

After that, the code includes the resource dictionary for the skin it should display. The previous code includes the resource file for managers, resManager.xaml, and the other resource files are commented out.

The code then uses resource properties to set the window's background brush, font, and title. Giving the skins different backgrounds and titles makes it easier to tell the skins apart at a glance.

Next, the code defines a `StackPanel` containing a series of `Buttons`. The `Buttons`' `Visibility` properties are set using resources defined in the skin resource dictionaries. The basic dictionary resBasics.xaml sets `Visibility = Collapsed` for all of the buttons. The other dictionaries override those settings to display the appropriate buttons. For example, in the Order Entry dictionary, resOrderEntry.xaml, the values `visCreateOrder` and `visTrackOrder` are set to `Visible` so the "Create Order" and "Track Order" buttons are shown in the order entry skin.

The following code shows the resBasics.xaml resource dictionary that defines common values for all of the skins. It defines the `Window`'s font characteristics and the `Button` style. It also hides all of the `Buttons`.


```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib">

    <!--  Main window  -->
    <FontFamily x:Key="ffWindow">Comic Sans MS</FontFamily>
    <FontWeight x:Key="fwWindow">Bold</FontWeight>
    <sys:Double x:Key="fsWindow">18</sys:Double>
    <sys:String x:Key="txtTitle">OrderTracking</sys:String>

    <!--  Buttons style  -->
    <Style TargetType="Button">
        <Setter Property="Width" Value="200"/>
        <Setter Property="Height" Value="50"/>
        <Setter Property="Margin" Value="10"/>
    </Style>

    <!--  Button visibilities  -->
    <Visibility x:Key="visUnshippedOrders">Collapsed</Visibility>
    <Visibility x:Key="visCreateOrder">Collapsed</Visibility>
    <Visibility x:Key="visFindCustomer">Collapsed</Visibility>
    <Visibility x:Key="visTrackOrder">Collapsed</Visibility>
    <Visibility x:Key="visSystemMaintenance">Collapsed</Visibility>
</ResourceDictionary>
```

OrderTracking

The following code shows the resOrderEntry.xaml skin resource dictionary. It defines the `Window`'s background brush and title, and the `Button` visibilities for the order entry skin.


```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib">

    <!--  Main window  -->
    <LinearGradientBrush x:Key="brWindow" StartPoint="0,0" EndPoint="0,1">
        <GradientStop Color="LightBlue" Offset="0"/>
        <GradientStop Color="Blue" Offset="1"/>
    </LinearGradientBrush>
```

```
    <sys:String x:Key="txtTitle">OrderTracking - Order Entry</sys:String>

    <!--  Button visibilities  -->
    <Visibility x:Key="visCreateOrder">Visible</Visibility>
    <Visibility x:Key="visTrackOrder">Visible</Visibility>
</ResourceDictionary>
```

OrderTracking

To use the OrderTracking program, you would compile the program and save the executable program. Then you would change the included resource dictionary to load a different skin, recompile the program, and save the new executable. You would repeat the process until you had created an appropriate executable for each type of user.

Rather than creating separate versions of the program for each type of user, you could load the appropriate skin at run time. The following sections describe three ways you can build skinnable applications in WPF, all of which let the program change its skin at run time.

Resource Skins

The program ResourceDictionaries shown in Figures 16-5 and 16-6 uses two different sets of resources to change its appearance at design time.

The following code shows the `Window`'s resource dictionary. The inner `ResourceDictionary` elements load two different resource dictionaries. Because dictionaries loaded later override those loaded earlier, you can change the application's appearance by changing the order of these two elements. (As shown here, the RedRed.xaml dictionary is loaded second, so the program uses its red interface, shown in Figure 16-5.)


```
<Window.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="ResBlue.xaml"/>
            <ResourceDictionary Source="ResRed.xaml"/>
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Window.Resources>
```

ResourceDictionary

While the program ResourceDictionaries can display two different appearances, you need to modify the program at design time to pick the skin you want. This may be useful for building different interfaces for different kinds of users, but a truly skinnable program should allow the user to change skins at run time.

To turn this into a truly skinnable application, all you need to do is give the program the ability to change skins at run time.

The Skins example program is very similar to the program ResourceDictionaries except that it can change skins at run time. To make that possible, most of its resources are dynamic rather than static. The program also contains two new user interface elements: an `Image` and a `Label`.

RESTRICTED SKINS

If you use different skins for different kinds of users (e.g., the order entry clerk and shipping clerk described in the previous section), then you'll need to restrict the skins that each user can load. For example, you probably wouldn't want the shipping clerk to be able to load the billing clerk's skin and view the customer's credit card information.

When it displays its red interface, this program adds a small `Image` in its upper-right corner. This `Image` has a context menu that displays the choices Red and Blue (shown on the left in Figure 16-8), which let you pick between the red and blue skins.

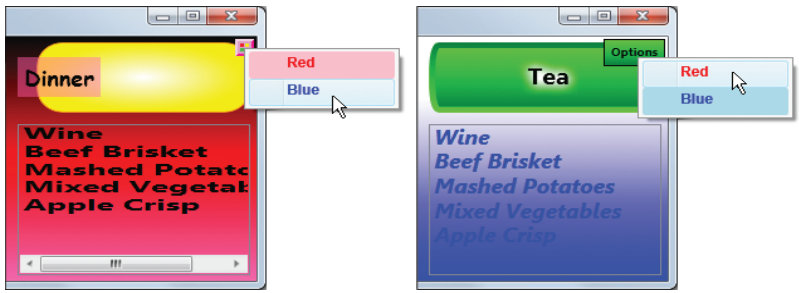

FIGURE 16-8

The program's blue interface displays a label in its upper-right corner (on the right in Figure 16-8) that displays the same context menu.

The following code shows how the program displays its Options textbox on the blue interface:

Available for download on Wrox.com

```
<Label MouseDown="Options_MouseDown"
 Grid.Row="0" Grid.Column="2" Margin="2"
 Content="Options" FontSize="10"
 HorizontalAlignment="Right" VerticalAlignment="Top"
 Foreground="Black" BorderBrush="Black"
 BorderThickness="1"
 Visibility="{DynamicResource visBlue}"
>
    <Label.Background>
        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <GradientStop Color="Lime" Offset="0"/>
            <GradientStop Color="Green" Offset="1"/>
        </LinearGradientBrush>
    </Label.Background>
    <Label.ContextMenu>
        <ContextMenu Name="ctxOptions">
            <MenuItem Header="Red" Background="Pink"
             Foreground="Red"
             Click="ctxSkin_Click" Tag="ResRed.xaml"/>
            <MenuItem Header="Blue" Background="LightBlue"
```

```
                Foreground="Blue"
                Click="ctxSkin_Click" Tag="ResBlue.xaml"/>
        </ContextMenu>
    </Label.ContextMenu>
</Label>
```


This code contains four real points of interest:

1. First, the `Label`'s `MouseDown` event triggers the `Options_MouseDown` event handler. This routine, which is shown in the following code, displays the context menu by setting the `ContextMenu`'s `IsOpen` property to `True`:


```
// Display the options context menu.
private void Options_MouseDown(object sender, RoutedEventArgs e)
{
    ctxOptions.IsOpen = true;
}
```


2. Second, the `Label`'s `Visibility` property is set to the value of the `visBlue` static resource. This resource has the value `True` in the Blue resource dictionary and `False` in the Red resource dictionary. This means that the `Label` is visible only in the blue interface. The red interface's skin-changing `Image` uses a similar `visRed` resource that is only `True` in the red interface.
3. Third, the context menu's items have a `Tag` property that names the XAML resource file that they load. For example, the Blue menu item has its `Tag` set to `ResBlue.xaml`. The program uses the `Tag` property to figure out which file to load when the user picks a menu item.
4. Finally, both of the context menu's items fire the `ctxSkin_Click` event handler shown in the following code to load the appropriate skin:


```
// Use the selected skin.
private void ctxSkin_Click(object sender, RoutedEventArgs e)
{
    // Get the context menu item that was clicked.
    MenuItem menu_item = (MenuItem)sender;

    // Create a new resource dictionary, using the
    // menu item's Tag property as the dictionary URI.
    ResourceDictionary dict = new ResourceDictionary();
    dict.Source = new Uri((String)menu_item.Tag, UriKind.Relative);

    // Remove all but the first dictionary.
    while (App.Current.Resources.MergedDictionaries.Count > 1)
    {
        App.Current.Resources.MergedDictionaries.RemoveAt(1);
    }

    // Install the new dictionary.
    App.Current.Resources.MergedDictionaries.Add(dict);
}
```

This code gets the menu item that triggered the event and looks at the item's `Tag` property to see which resource file to load. It creates a `ResourceDictionary` object loaded from the file, removes old resource dictionaries from the application's MergedDictionaries collection, and adds the new dictionary.

> **REMOVED RESOURCES REDUX**
>
> The program doesn't remove the first resource dictionary so that WPF doesn't get confused about missing resources and issue a flock of warnings. For more information, see the note "Removed Resources" in the "Dynamic Resources" section of Chapter 12.

When the program loads the new resource dictionary, WPF detects the changed values and updates all of the window's dynamic resources.

This technique is what most developers think of as *skinning* in WPF applications: The program loads multiple resource files at run time to provide different skins.

Animated Skins

The skins described in the previous section use separate resource dictionaries to provide different appearances. The program's XAML file sets its control properties to resource values so that when you change the resource values, the interface changes accordingly.

Another way to change property values is to use property animation. Chapter 14 covers property animation in greater detail, but this section explains briefly how to use animation to provide skinning.

XAML files allow you to define triggers that launch storyboards that represent property animations. For example, when the user presses the mouse down over a rectangle, the XAML code can run a storyboard that varies the `Rectangle`'s `Width` property smoothly from 100 to 200 over a 1-second period.

The AnimatedSkins example program uses this technique to provide skinning. Figure 16-9 shows the program displaying its green skin. Figure 16-10 shows its blue skin.

When you click the appropriate control, a trigger launches a storyboard that:

- Resizes the main window and changes its `Background` brush.
- Hides and displays the small blue or green ellipses in the upper-right corner that you click to switch skins.
- Moves `Labels`.

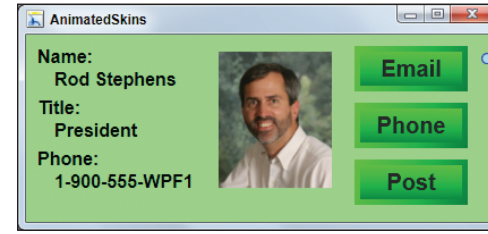

FIGURE 16-9

FIGURE 16-10

- Resizes, moves, and changes the corner radii of the rectangles that act as buttons.
- Changes the `Fill` and `Stroke` brushes for the rectangles pretending to be buttons.
- Changes the text displayed in the `Labels`.
- Moves and resizes the `Image`.
- Changes the background and foreground colors.

Figure 16-11 shows the program a bit less than halfway done switching from the green to the blue skin. In this figure, you can see that the colors are moving from green to blue, the labels are moving, and the rectangle buttons have new positions, sizes, captions, and rounded corners.

In addition to displaying very different appearances, animated skins let the user watch as one interface morphs into another. The effect is extremely cool.

FIGURE 16-11

THE PRICE OF COOLNESS

You might argue that coolness isn't really the focus in many applications, and you would be completely correct, but programmers who write skins aren't usually focused on getting by with the least possible work. It's hard to argue that most skinning serves anything other than an aesthetic purpose, so as long as you're spending extra effort providing skins, it's not completely fair to say that the extra coolness of animated skins isn't worth the effort. By the same token, you could argue that you shouldn't even be using WPF and should stick with Windows Forms programming, which is generally easier.

That being said, however, be warned that animating skins is a *lot* of work. Tweaking the animations to give everything exactly the right position, size, and appearance takes time. The AnimatedSkins example program uses only two storyboards (one for each skin) but more than 100 property animations to get everything right. And the differences between these two skins aren't as great as some I've seen, so you could spend a huge amount of time getting everything just right.

One interesting side effect of this technique is that one animation doesn't need to finish before a new animation can start. For example, suppose you click on the blue circle in Figure 16-9 to switch to the blue skin. After the controls start moving to their new positions, you can click on the green circle shown in Figure 16-10. At that point, the controls immediately start moving back to their positions for the green skin without going all the way to their blue skin positions.

Dynamically Loaded Skins

One of the drawbacks of the previous two skinning techniques is that they only modify existing objects. They can display an `Ellipse`, `Button`, or `Label` with different properties, but they are still the same `Ellipse`, `Button`, or `Label`. For example, you cannot provide one skin that launches tasks with `Buttons`, another that uses `Menus`, and a third that uses `Labels`.

One common solution to this problem is to include every set of controls in every skin and then hide the ones that you don't need. For example, you would include `Buttons`, `Menus`, and `Labels` in every skin. Then the button-oriented skin would hide the `Menus` and `Labels`, the menu-oriented skin would hide the `Buttons` and `Labels`, and the label-oriented skin would hide the `Buttons` and `Menus`.

Another solution to this problem might be to use separate XAML files that sit on top of the same code-behind. Unfortunately, WPF doesn't handle this situation very well.

WPF provides methods for loading XAML files with or without event handlers attached. The short version of the story is, if you want to load XAML code with event handlers, then you can only have one XAML file associated with each code-behind class. If you load XAML code without event handlers, then you need to wire up the event handlers yourself.

THE LONGER STORY

If you want to load XAML files with event handlers, then you need to associate the XAML with a class defined in your code-behind. Unfortunately, WPF adds its own automatically generated bonus routines to perform some extra chores such as connecting the XAML events with the event handlers provided by your class. If you try to associate two XAML files with the same class, WPF creates multiple copies of those routines with the same signatures and that confuses Visual Studio.

You might try to make multiple classes for the XAML files by having them inherit from a common base class that provides all of the necessary functionality. Sadly, the automatically generated code makes your class inherit from a WPF control type. For example, if your XAML file contains a `Grid` as its root element, then the code makes your class inherit from the `Grid` class. That means that you cannot also make it inherit from your desired base class.

The only direct solution I've found is to make completely separate classes for each XAML file, but that kind of defeats the goal of trying to use common code-behind.

Wiring up events to event handlers isn't hard, although it does reduce the separation between user interface design and writing the code-behind. Now the interface designer and the programmer must agree on the event handlers that the code will use and on the names of the controls that use them.

INTERFACE IRONY

The difficulty of attaching multiple XAML files to the same code-behind seems somewhat ironic given how much emphasis WPF places on separation of user interface and code-behind. You can separate an interface from its code but only as long as you keep them logically associated with each other.

The SkinInterfaces example program displays new skins at run time by loading XAML files and wiring up their event handlers. Figures 16-12 and 16-13 show the program displaying its two skins.

FIGURE 16-12

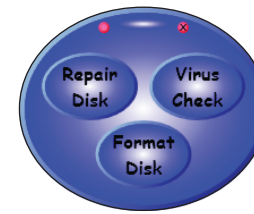

FIGURE 16-13

These skins not only provide radically different appearances, but they also use different types of controls that generate different kinds of events. The following table lists the types of controls and events that each skin uses:

	RED SKIN		BLUE SKIN	
PURPOSE	CONTROL	EVENT	CONTROL	EVENT
Switch skin	`Polygon`	`MouseDown`	`Ellipse`	`MouseDown`
Move form	`Rectangle`	`MouseDown`	`Ellipse`	`MouseDown`
Exit	`Grid` (containing a `Rectangle` and a `TextBlock`)	`MouseDown`	`Grid` (containing an `Ellipse` and a `TextBlock`)	`MouseDown`
Repair disk	`Button`	`Click`	`Grid` (containing an `Ellipse` and a `TextBlock`)	`MouseDown`
Virus check	`Button`	`Click`	`Grid` (containing an `Ellipse` and a `TextBlock`)	`MouseDown`
Format disk	`Button`	`Click`	`Grid` (containing an `Ellipse` and a `TextBlock`)	`MouseDown`

When the program loads a XAML file, it looks through the new controls and attaches event handlers to those that need them.

To provide some separation between the XAML files and the code-behind, the program uses a separate group of routines to do the real work. Event handlers catch the control events and call the work routines to do all the interesting stuff.

The following code shows how the blue skin defines its red switch skin circle on the left at the form's top:


```
<Grid
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Tag="Blue"
    >

        Lots of code omitted ...

    <Ellipse Name="ellSkin" Width="10" Height="10"
     Canvas.Left="70" Canvas.Top="13"
     Cursor="Cross" ToolTip="Change Skin"
     Fill="HotPink" Stroke="{StaticResource brRedStroke}"
     StrokeThickness="2" Tag="Red.xaml"
    />

        Lots of code omitted ...

</Grid>
```

SkinInterfaces

The code fragment starts with a `Grid` control as its root element. (You can use other controls for the file's root, but `Grid` is convenient, partly because the Visual Studio WPF Window Designer can understand how to display the file if its root is a `Grid`.) The root element's `Tag` property is set to the name of the skin it represents, in this case, *Blue*.

The code shown here omits all of the other controls except the "switch skin" circle.

The most important pieces of the circle's definition are its name, *ellSkin*, and its `Tag`, `Red.xaml`. The `Tag` property tells the code-behind which XAML skin file to load when the circle is clicked.

Ignoring for the moment how this control is wired up to its event handler, the following code shows the event handler that the circle executes. Since this event handler is shared by this circle and the red skin's "change skin" polygon (the blue triangle in the upper-left corner), it's called `pgnSkin_MouseDown`.


```
private void pgnSkin_MouseDown(object sender, MouseButtonEventArgs e)
{
    FrameworkElement element = (FrameworkElement)sender;
    LoadSkin(element.Tag.ToString());
}
```

SkinInterfaces

This code gets the element that triggered the event (either the blue skin's `Ellipse` or the red skin's `Polygon`), reads that element's `Tag` property to see which XAML file to load, and passes the filename to the function `LoadSkin`.

The function `LoadSkin` uses the following code to load a XAML skin file. To save space, the code only shows a few of the statements that connect controls to their event handlers.


```
// Load the skin file and wire up event handlers.
private void LoadSkin(string skin_file)
{
    // Load the controls.
    FrameworkElement element =
        (FrameworkElement)Application.LoadComponent(
            new Uri(skin_file, UriKind.Relative));
    this.Content = element;

    // Wire up the event handlers.
    Button btn;
    Polygon pgn;
    Rectangle rect;
    Grid grd;
    Ellipse ell;

    switch (element.Tag.ToString())
    {
        case "Red":
            btn = (Button)element.FindName("btnRepairDisk");
            btn.Click += new RoutedEventHandler(btnRepairDisk_Click);

                Code for other controls omitted
            break;

        case "Blue":
                Lots of code omitted

            // Uses the same event handler as rectMove.
            ell = (Ellipse)element.FindName("ellMove");
            ell.MouseDown +=
                new System.Windows.Input.MouseButtonEventHandler(
                    rectMove_MouseDown);

            grd = (Grid)element.FindName("grdExit");
            grd.MouseDown +=
                new System.Windows.Input.MouseButtonEventHandler(
                    grdExit_MouseDown);
            break;
    }
}
```

SkinInterfaces

The code starts by using the WPF `LoadComponent` method to load the desired XAML skin file. It sets the `Window`'s main content element to the root loaded from the file so the new controls are displayed.

Next, the code checks the newly loaded root element's `Tag` property to see whether it is now displaying the red or the blue skin. Depending on which skin is loaded, the code looks for specific controls in the skin and connects their event handlers.

For example, if the red skin is visible, the code uses `FindName` to locate the `btnRepairDisk Button` and adds the `btnRepairDisk_Click` event handler to its `Click` event.

The previous code omits most of the code connecting controls to event handlers. It does, however, show how the code finds the `ellSkin` control (the "change skin" circle) and adds the `pgnSkin_MouseDown` event handler to its `MouseDown` event.

The code that wires up the controls that are not shown here is similar.

The program's final piece is in the `Window`'s constructor, which is shown in the following code. After the `Window` is initialized, the code calls `LoadSkin` to start with the red skin.


```
public Window1()
{
    this.InitializeComponent();

    // Insert code required on object creation below this point.

    // Start with the red skin.
    LoadSkin("Red.xaml");
}
```

SkinInterfaces

That completes the program's circle of life. When the program starts, it calls `LoadSkin` to load the red skin. `LoadSkin` loads the controls and wires up their event handlers. In particular, it attaches an event handler to the `pgnSkin` "change skin" control's `MouseDown` event. When you click on the polygon, the `pgnSkin_MouseDown` event handler executes and calls `LoadSkin` to start the whole process over again.

Despite the extra code-behind that locates specific controls and attaches events to event handlers, this technique is reasonably straightforward. Wiring up the controls can be long, but it's easy to understand.

The skin files can set control properties directly instead of requiring that you use a huge number of dynamic resources, so the code is a lot simpler than it is when you use different resource dictionaries.

This method doesn't provide property animation, which makes it less cool, but it's much easier to implement.

Finally, this technique allows different skins to use different controls for similar purposes. It lets you make skins that launch actions from `Buttons`, `MenuItems`, `MouseDown` events, and pretty much any other event you might want to catch.

SUMMARY

This chapter explains themes and skins. Themes let every application on the user's computer provide a similar look and feel. Normally, you don't need to do anything to take advantage of themes. If you don't override the default appearance of controls, then they automatically match the system's currently selected theme and update themselves as needed when the theme changes.

Skins let you change an application's appearance and behavior, essentially letting you define a "mini-theme" for the application. The examples in this chapter show how to use different skins for different purposes, load skins at design time or run time, build animated skins, and load skins that may use completely different controls.

Chapters 12 through 16 cover topics that control the application's behavior and appearance. They explain how to use resources, styles, templates, triggers, and themes to give an application a distinctive and consistent look-and-feel. These techniques are useful for building any WPF application.

The chapters that follow turn to more specific topics that are not necessarily essential for every application. These chapters explain important techniques that you will find useful in many applications. For example, Chapter 17 explains one of the more basic needs of many applications: printing.