WPF201-B

Module 4

Implementing WPF Custom & User Controls

1 Triggers

1.1 Property Triggers

Styles are not only used to setup initial properties of elements but also dynamically update them when the state of the element changes. Style triggers can be used to detect when a dependency property is changed during runtime and execute a set of setter actions.

Using style triggers: Triggers1\MainWindow.xaml

```
<Window.Resources>
    <Style x:Key="TextStyle1" TargetType="Control">
        <Setter Property="FontFamily" Value="Verdana" />
        <Setter Property="FontSize" Value="16" />
        <Setter Property="Margin" Value="4" />
        <Style.Triggers>
            <Trigger Property="IsFocused" Value="True">
                <Setter Property="FontStyle" Value="Italic" />
                <Setter Property="Background" Value="SteelBlue" />
                <Setter Property="Foreground" Value="White" />
            </Trigger>
            <Trigger Property="IsMouseOver" Value="True">
                <Setter Property="BorderThickness" Value="2" />
                <Setter Property="BorderBrush" Value="LightSteelBlue" />
            </Trigger>
        </Style.Triggers>
    </Style>
</Window.Resources>
<Grid>
    <StackPanel>
        <TextBox x:Name="Text1" Style="{StaticResource TextStyle1}" />
        <TextBox x:Name="Text2" Style="{StaticResource TextStyle1}" />
        <TextBox x:Name="Text3" Style="{StaticResource TextStyle1}" />
        <TextBox x:Name="Text4" Style="{StaticResource TextStyle1}"/>
     </StackPanel>
</Grid>
```

Sometimes you need more than one condition to be satisfied before an element can be styled. This can be done by using **MultiTrigger**. Add **Condition** to the **Conditions** collection property to specify the property values that would collectively activate the trigger. Add the following trigger to the **Triggers** of the previous style.

Example of a multiple conditions trigger

```
<MultiTrigger>
    <MultiTrigger.Conditions>
        <Condition Property="IsFocused" Value="True" />
        <Condition Property="IsMouseOver" Value="True" />
    </MultiTrigger.Conditions>
    <Setter Property="Foreground" Value="Cyan" />
     <Setter Property="FontSize" Value="32" />
</MultiTrigger>
```

1.2 Enter & Exit Actions

If you wish to perform animations rather than just setting the property values, you can also perform actions by adding the actions to **EnterActions** or to **ExitActions**. Enter actions are invoked when the element enters the state as defined by the trigger condition and exit actions are invoked when the element exits that state. To try this, first add a **LayoutTransform** setter to the style as shown below.

Setter to assign a RotateTransform to control

```
<Style x:Key="TextStyle1" TargetType="Control">
    <Setter Property="FontFamily" Value="Verdana" />
    <Setter Property="FontSize" Value="16" />
    <Setter Property="Margin" Value="4" />
    <Setter Property="LayoutTransform">
        <Setter.Value><RotateTransform Angle="0" /></Setter.Value>
    </Setter>
```

Then add the following storyboards as resources of the style. Note how you can target the **Angle** property of a RotateTransform assigned to LayoutTransform of the control. Once you have done this, add EnterActions and ExitActions to the **IsFocused** trigger to activate the storyboards. Run the application and see the results. Note that the **TargetType** of the style is **Control** so it is possible for us to apply the above style to all controls and not just only **TextBox**.

Storyboards added as style resources

```
<Style.Resources>
    <Storyboard x:Key="BeginRotate">
        <DoubleAnimation From="0" To="-15" Duration="00:00:01"
        Storyboard.TargetProperty="LayoutTransform.(RotateTransform.Angle)" />
    </Storyboard>
    <Storyboard x:Key="EndRotate">
        <DoubleAnimationFrom="-15" To="0" Duration="00:00:00.500"
        Storyboard.TargetProperty="LayoutTransform.(RotateTransform.Angle)" />
    </Storyboard>
</Style.Resources>
```

Activating actions from style trigger

```
<Trigger Property="IsFocused" Value="True">
    <Setter Property="FontStyle" Value="Italic" />
    <Setter Property="Background" Value="SteelBlue" />
    <Setter Property="Foreground" Value="White" />
    <Trigger.EnterActions>
        <BeginStoryboard Storyboard="{StaticResource BeginRotate}" />
    </Trigger.EnterActions>
    <Trigger.ExitActions>
        <BeginStoryboard Storyboard="{StaticResource EndRotate}" />
    </Trigger.ExitActions>
</Trigger>
```

1.3 Data Triggers

Even though data-binding is discussed in more details in later modules it is essential to know that any data bounded to elements can activate triggers. In this example we use a **ListBox** to display a list a objects. A ListBox is an **ItemsControl** where it has an **Items** property that you can add one or more objects to display. You can assign a collection of objects using **ItemsSource** property. If the object is not an UI element, the ListBox would convert the object to a string and display each item in a TextBlock. However you are allowed to assign a **DataTemplate** to the **ItemTemplate** property that the ListBox will use to generate the content for each item. In the following XAML content we added a ListBox element to display multiple items.

Using a ListBox to display multiple items: Triggers2\MainWindow.xaml

```
<Grid>
    <ListBox x:Name="lsbAccounts" />
</Grid>
```

You now need data to bind to the control. Add the **DataModel** source file provided by the instructor that contains **AccountList** class that provides an collection of **Account** instances. Each Account has properties ID, Name, Type, Location and Balance. Add a Loaded event handler to the window and write the following code to obtain data to be then assigned to the ListBox. When you execute the application you will just see a list of strings.

Binding items collection to ItemsControl using ItemsSource

```
protected void Window_Loaded(object sender, EventArgs e) {
    lsbAccounts.ItemsSource = new AccountList();
}
```

A DataTemplate provides a XAML fragment that is used to render the presentation for data elements. Properties in the template can be binded to properties of a data object by using **Binding** XAML markup extension. Templates can be shared since they are resources rather than UI element. The following is a **DataTemplate** resource that is placed as a window shared resource.

A DataTemplate resource

```
<DataTemplate x:Key="AccountItemTemplate">
    <Border Width="500" Margin="4" CornerRadius="8"
        BorderThickness="2" BorderBrush="SteelBlue">
        <Grid Margin="4">
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="120" />
                <ColumnDefinition />
            </Grid.ColumnDefinitions>
            <Label Grid.Row="0" Content="ID" />
            <Label Grid.Row="1" Content="Name" />
            <Label Grid.Row="2" Content="Type" />
            <Label Grid.Row="3" Content="Location" />
            <Label Grid.Row="4" Content="Balance" />
            <Label Grid.Column="1" Grid.Row="0" Content="{Binding ID}" />
            <Label Grid.Column="1" Grid.Row="1" Content="{Binding Name}" />
            <Label Grid.Column="1" Grid.Row="2" Content="{Binding Type}" />
            <Label Grid.Column="1" Grid.Row="3" Content="{Binding Location}" />
            <Label Grid.Column="1" Grid.Row="4" Content="{Binding Balance}" />
        </Grid>
    </Border>
</DataTemplate>
```

Assigning the template to ListBox

```
<ListBox x:Name="lsbAccounts"
    ItemTemplate="{StaticResource AccountItemTemplate}" />
```

Run the application and you should see the DataTemplate used for each item in the collection. By default the ListBox uses a vertical StackPanel as the container for the items. You can change panel using the **ItemsPanel.ItemsPanelTemplate** property. Following example we change the control to use a horizontal WrapPanel instead. This provides smoother scrolling and you can also decide to scroll horizontally instead.

Changing items panel

```
<ListBox x:Name="lsbAccounts"
    ItemTemplate="{StaticResource AccountItemTemplate}">
    <ListBox.ItemsPanel>
        <ItemsPanelTemplate>
            <WrapPanel />
        </ItemsPanelTemplate>
    </ListBox.ItemsPanel>
</ListBox>
```

Since you are using one data template for all the items, they all look the same when rendered. However it is possible to render each item differently based on the data in each object. You can do this by using data triggers. For example we wish to style the **Border** of each item to be styled different based on account **Type**. Create a style for the border and you can use **DataTrigger** to provide different setters base on the data you bound. Then assign the style to the border inside the template.

Implementing DataTriggers

```
<Style x:Key="AccountItemStyle" TargetType="Border">
    <Style.Triggers>
        <DataTrigger Binding="{Binding Type}" Value="Savings">
            <Setter Property="Background" Value="Azure" />
        </DataTrigger>
        <DataTrigger Binding="{Binding Type}" Value="FixedDeposit">
            <Setter Property="Background" Value="Beige" />
        </DataTrigger>
        <DataTrigger Binding="{Binding Type}" Value="Checking">
            <Setter Property="Background" Value="Aquamarine" />
        </DataTrigger>
    </Style.Triggers>
<Style>
```

Assigning style to item border

```
<DataTemplate x:Key="AccountItemTemplate">
    <Border Width="500" Margin="4" CornerRadius="8"
        BorderThickness="2" BorderBrush="SteelBlue"
        Style="{StaticResource AccountItemStyle}">
```

If you want to style based on multiple data bindings use the **MultiDataTrigger** class instead. You can then assign multiple conditions to **Conditions** property. The setters will only be used when all the conditions are true. Following shows how to set the background to a gradient based on both account Type and Location. **EnterActions** and **ExitActions** are also available for data triggers so you can also automatically perform animations based on data bindings. Imagine displaying a list of products, a product would automatically blink when it is out of stock.

Implementing a MultiDataTrigger

```
<MultiDataTrigger>
    <MultiDataTrigger.Conditions>
        <Condition Binding="{Binding Type}" Value="FixedDeposit" />
        <Condition Binding="{Binding Location}" Value="Local" />
    </MultiDataTrigger.Conditions>
    <Setter Property="Background">
        <Setter.Value>
            <LinearGradientBrush
                StartPoint="0,0" EndPoint="0,1">
                <GradientStop Color="Beige" Offset="0" />
                <GradientStop Color="Azure" Offset="1" />
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
</MultiDataTrigger>
```

A **DataTemplate** can be assigned to a specific **DataType**. Then when data of that type is assigned to any element that uses a DataTemplate it would automatically look for a template that matches the data type. Remove the data template resource **Key** and replace it with **Type** binding to the **Account** type.

Defining a DataTemplate for a specific data type

```
<DataTemplate DataType="{x:Type local:Account}">
        :
```

Remove the resource binding from list box since it is no longer required to statically bind the DataTemplate. The correct DataTemplate will be used based on the type of data that has been assigned. This means that a list box can use multiple templates if you assign different types of objects to it. It can locate the correct template to display each type of object.

2 Templated Controls

You can extend WPF by implementing your own UI elements and resources. All WPF elements derive directly or indirectly from **DependencyObject**. This allows your WPF elements to have dependency and attached properties, routed commands and events.

Class library project

```
Project Name:    Symbion
Project Type:    Visual C# | Windows | Custom Control Library
Location     :   C:\WPFDEV\SRC
Solution     :   Module4
```

You can register an URI for your namespaces by using **XmlnsDefinitionAttribute**. Register the URI by providing two strings; the first is the URI to use be used in XAML and the second is the actual .NET namespace your classes are defined in. You can register multiple .NET namespaces using a single URL.

Registering a XML namespace URI: Properties\AssemblyInfo.cs

```
[assembly:XmlnsDefinition(
    "http://www.symbolicon.net/symbion",
    "Symbion")]
```

Registering a prefix using the URI to use the library

```
<Window x:Class="Custom1.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sym="http://www.symbolicon.net/symbion"
    Title="MainWindow" Height="350" Width="525">
            :
</Window>
```

2.1 Templates & Parts

You can create custom controls that uses a template to determine the visual look and behavior of the control. This would allow the control to be themed. You must provide the default template in a resource dictionary named **generic.xaml** in a folder named **Themes**. The **TargetType** of the **Style** resource that contains the template must be the same as the custom control type which become the key to fetch the style. To register this style as default style for your control, you must have a static constructor that uses the **DefaultStyleKeyProperty** class to register its default style key.

Registering default style key: Symbion\MetroButton.cs

```
using System.Windows;
using System.Windows.Controls;

namespace Symbion {
    public class MetroButton : Control {
        static MetroButton() {
            DefaultStyleKeyProperty.OverrideMetadata(typeof(MetroButton),
                new FrameworkPropertyMetadata(typeof(MetroButton)));
        }
    }
}
```

Default style and template for control: Symbion\Themes\Generic.xaml

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:Symbion">
    <Style TargetType="{x:Type local:MetroButton}">
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="{x:Type local:MetroButton}">
            <Border Background="{TemplateBinding Background}"
                BorderBrush="{TemplateBinding BorderBrush}"
                BorderThickness="{TemplateBinding BorderThickness}">
                    </Border>
            </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
</ResourceDictionary>
```

Testing the custom control: Custom1\MainWindow.xaml

```
<Grid>
    <StackPanel
        Orientation="Horizontal"
        HorizontalAlignment="Center">
        <sym:MetroButton x:Name="btn1" />
        <sym:MetroButton x:Name="btn2" />
        <sym:MetroButton x:Name="btn3" />
    </StackPanel>
</Grid>
```

The **Style** is not only for applying a template but for configuring default properties of the control. You can then create default content for the template. If this control is not only for presentation but the user is also allowed to interact with it, you must assign names to those parts so that you can target them in a trigger or access them from your control code. For our control at the moment, we have only one part named as **PART_BORDER**.

Setting default values for control properties: Symbion\Themes\Generic.xaml

```
<Style TargetType="{x:Type local:MetroButton}">
    <Setter Property="SnapsToDevicePixels" Value="True" />
    <Setter Property="UseLayoutRounding" Value="True" />
    <Setter Property="Background" Value="White" />
    <Setter Property="BorderThickness" Value="2" />
    <Setter Property="BorderBrush" Value="Gray" />
    <Setter Property="Cursor" Value="Hand" />
    <Setter Property="Width" Value="40" />
    <Setter Property="Height" Value="40" />
    <Setter Property="Margin" Value="4" />
                    :
```

Declaring elements and named parts in the template

```
<ControlTemplate TargetType="{x:Type local:MetroButton}">
    <Grid x:Name="_Root">
        <Ellipse x:Name="PART_BORDER"
            Stroke="{TemplateBinding BorderBrush}"
            StrokeThickness="{TemplateBinding BorderThickness}"
            Fill="{TemplateBinding Background}" />
    </Grid>
</ControlTemplate>
```

To fetch parts in a template you can override the **OnApplyTemplate** method. In the method you can access the **Template** and use **FindName** to locate required parts. You are free to determine what part type is acceptable. If the part type is **UIElement** or **FrameworkElement**, the new template can choose to use any type of element for the part. If part type is **Ellipse** or **Shape**, then you are restricting the type of part.

Usually the template designer will have to ensure that the part with the correct name and the required type must be in their custom template. However, even if they don't, we will create a dummy part to ensure that our control will still work properly but will not be rendered visually. Here we access our part to attach an event handler to it.

Fetching named parts from the template: MetroButton.cs

```
private UIElement _border;

public override void OnApplyTemplate() {
    _border = Template.FindName("PART_BORDER", this) as UIElement;
    if (_border == null) _border = new Ellipse();
    _border.MouseLeftButtonUp += Border_MouseLeftButtonUp;
}
private void Border_MouseLeftButtonUp(object sender, MouseButtonEventArgs e) {
    // activated when the left button released on the border
    e.Handled = true; // stop routing
}
```

2.2 Routed Events

Click is not a standard event for all the controls. We wish our control to behave like a button so we need to register our own **RoutedEvent** and raise the event when a user interacts with a template part as shown below. Now regardless of whether the default or the custom template is used, the required part will generate the **Click** event when the left mouse button is released over it.

Declaring a routed event

```
public static readonly RoutedEvent ClickEvent;
```

To attach and remove handlers to routed events, you need to call **AddHandler** and **RemoveHandler** methods. You can implement a CLR event to simplify management of event handlers. You can then assign handlers using the same .NET syntax.

Declaring a .NET event to simply access to WPF routed event

```
public event RoutedEventHandler Click {
    add { AddHandler(ClickEvent, value); }
    remove { RemoveHandler(ClickEvent, value); }
}
```

Routed events have to be registered with the WPF **EventManager** which would help to route the events to the event targets that will handle the event. You will need to choose the correct **RoutingStrategy** to ensure that the event reaches the correct target.

Registering routed event with EventManager

```
static MetroButton() {
        ClickEvent = EventManager.RegisterRoutedEvent(
            "Click", RoutingStrategy.Bubble,
            typeof(RoutedEventHandler), typeof(MetroButton));
                    :
}
```

There are three available routing strategies. **Direct** means no routing. **Bubble** mean route upwards from child to parent. **Tunnel** means routing in the opposite direction which is downwards from parent to child. The default strategy is bubble. Anywhere in your code, construct a **RoutedEventArgs** object to store event information and use **RaiseEvent** to trigger the routed event.

Available routing strategies

```
Bubble  Routing from child upwards to parent, eventually to root
Tunnel  Routing from parent to child, eventually to leaf element
Direct  Must be handled directly, does not bubble nor tunnel
```

Raising event when user interacts with template part

```
private void Border_MouseLeftButtonUp(object sender, MouseButtonEventArgs e) {
    RaiseEvent(new RoutedEventArgs(ClickEvent, this));
//  e.Handled = true;
}
```

2.3 Dependency Properties

One of the main reasons why we need to implement custom controls is because we cannot locate an existing control that has all the properties that we need for use in the control template or triggers. If there is an existing control that all we need to do is to customize the control template instead. The **Button** control has no property that we can assign geometry. For our control, we will declare and register a dependency property named **Icon** for assigning geometry.

Declaring, exposing and register dependency properties

```
public static readonly DependencyProperty IconProperty;
public Geometry Icon {
    get { return (Geometry)GetValue(IconProperty); }
    set { SetValue(IconProperty, value); }
}
static MetroButton() {
    IconProperty = DependencyProperty.Register(
        "Icon", typeof(Geometry), typeof(MetroButton));
                :
```

Setting the default content property

```
[ContentProperty("Icon")]
public class MetroButton : Control { ... }
```

We will now update our template to use a **Path** element to present that geometry. We don't need to write code as we can use **TemplateBinding** to bind the geometry to render to the **Data** property of the Path.

Presenting geometry with Path in control template

```
<ControlTemplate TargetType="{x:Type local:MetroButton}">
    <Grid x:Name="_Root">
        <Ellipse x:Name="PART_BORDER"
            Stroke="{TemplateBinding BorderBrush}"
            StrokeThickness="{TemplateBinding BorderThickness}"
            Fill="{TemplateBinding Background}" />
        <Path x:Name="PART_ICON" Margin="8" Stretch="Fill"
            Stroke="{TemplateBinding BorderBrush}"
            StrokeThickness="{TemplateBinding BorderThickness}"
            Data="{TemplateBinding Icon}"
            IsHitTestVisible="False" />
    </Grid>
</ControlTemplate>
```

Creating geometry resource: Custom3\MainWindow.xaml

```
<Window.Resources>
    <PathGeometry x:Key="geo1">
        M0.25,0.5L0.75,0.5M0.5,0.25L0.25,0.5L0.5,0.75
    </PathGeometry>
</Window.Resources>
```

Binding geometry to Icon dependency property

```
<sym:MetroButton x:Name="btn1" Click="btn1_Click"
    Icon="{StaticResource geo1}" />
```

Rather than leaving the **Icon** blank initially we can have default geometry to be used when no geometry has been assigned to the property. Declare a **PathGeometry** as a resource in **Generic.xaml** and bind it to the **Icon** property in the default style for the control. You can now test this out in the application.

Default geometry for Icon property: Symbion\Themes\Generic.xaml

```
<PathGeometry x:Key="CrossIcon">
    M0.25,0.25L0.75,0.75
    M0.75,0.25L0.25,0.75
</PathGeometry>

<Style TargetType="{x:Type local:MetroButton}">
    <Setter Property="Icon" Value="{StaticResource CrossIcon}" />
                        :
</Style>
```

There is a problem with the **StrokeThickness** property of the **Ellipse** and the **Path** in the template which at the moment is bound to **BorderThickness**. It does not work as BorderThickness and StrokeThickness are different types. BorderThickness type is **Thickness** while StrokeThickness is a **double** value. To resolve this, add a separate dependency property for StrokeThickness. You can try this on your own.

2.4 Visual State

At the moment, when user interacts with the control, there is no change in the visual state of the control. Only the mouse cursor will change to indicate the control can be pressed when it is over the control. We can add triggers to the control template to detect any changes in control state properties to update the visual state. However, we have to ensure that our control has all of the required state properties. For our control, we need to have **IsMouseOver**, **IsPressed** and **IsEnabled** state properties. All controls have IsMouseOver and IsEnabled but not IsPressed. Thus we need to add this state property to our control. It is a dependency property but the **set** method should be **private**.

Declaring, exposing and registering state properties: MetroButton.cs

```
public static readonly DependencyProperty IsPressedProperty;
public bool IsPressed {
    get { return (bool)GetValue(IsPressedProperty); }
    private set { SetValue(IsPressedProperty, value); }
}
static void MetroButton() {
                :
    IsPressedProperty = DependencyProperty.Register(
        "IsPressed", typeof(bool), typeof(MetroButton));
}
```

We now need to attach an additional event handler to the border part to detect if the mouse left button is pressed over it and then update the state property appropriately. The state will also be updated when the button is released.

Registering event handlers to update state

```
public override void OnApplyTemplate() {
    _border = Template.FindName("PART_BORDER", this) as UIElement;
    if (_border == null) _border = new Ellipse();
    _border.MouseLeftButtonUp += Border_MouseLeftButtonUp;
    _border.MouseLeftButtonDown += Border_MouseLeftButtonDown;
}
private void Border_MouseLeftButtonDown(object sender, MouseButtonEventArgs e){
    IsPressed = true; //e.Handled = true;
}
private void Border_MouseLeftButtonUp(object sender, MouseButtonEventArgs e) {
    RaiseEvent(new RoutedEventArgs(ClickEvent, this));
    IsPressed = false; //   e.Handled = true;
}
```

Now that we have all the state properties that we need, update the template to set additional properties and add additional elements that we access and update using triggers. Assign triggers to the **Triggers** property of **ControlTemplate**. Use **Setters** to change properties or do animations using **EnterActions** and **ExitActions**.

Additional elements and properties: Themes\Generic.xaml

```
<Grid x:Name="_Root">
    <Ellipse x:Name="PART_BORDER" Opacity="0.5"
        Stroke="{TemplateBinding BorderBrush}"
        StrokeThickness="{TemplateBinding StrokeThickness}"
        Fill="{TemplateBinding Background}" />
    <Path x:Name="PART_ICON" Margin="8" Stretch="Fill" Opacity="0.5"
        Stroke="{TemplateBinding BorderBrush}"
        StrokeThickness="{TemplateBinding StrokeThickness}"
        Data="{TemplateBinding Icon}" IsHitTestVisible="False" />
    <Rectangle x:Name="PART_OVERLAY"
        Fill="White" Opacity="0" IsHitTestVisible="False"/>
</Grid>
```

Triggers to update visual state

```
<ControlTemplate.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
        <Setter TargetName="PART_BORDER" Property="Opacity" Value="0.75" />
        <Setter TargetName="PART_ICON" Property="Opacity" Value="0.75" />
    </Trigger>
    <Trigger Property="IsPressed" Value="True">
        <Setter TargetName="PART_BORDER" Property="Opacity" Value="1" />
        <Setter TargetName="PART_ICON" Property="Opacity" Value="1" />
    </Trigger>
    <Trigger Property="IsEnabled" Value="False">
        <Setter TargetName="PART_OVERLAY" Property="Opacity" Value="0.75" />
    </Trigger>
</ControlTemplate.Triggers>
```

Our **MetroButton** is now a fully functional button that can be used directly. The **Icon** property can be assigned the geometry to be present in the button. It can be fully styled where you can change the size, alignment, thickness and brush to paint each button. You can also change the entire visual look and behavior by replacing the default style with a custom style that overrides the template.

3 User Controls

3.1 Implementing User Controls

Building a complex user-interface in one window can daunting especially if it involves a lot of animations and there is a need to show different content or to present the same content but in different ways. However this can be done quite easily by using user controls. User controls allow you to componentize user-interfaces. This simplifies construction of the UI. User controls can be easily reused and replaced dynamically at runtime. Animations can be encapsulated within as well as local resources. Just like any control you can add dependency properties, routed commands and events as well. Just like a window, you can use the Visual Designer to visually construct a user control or use XAML for the presentation. Following is a simple control to display an image with a title and with a border.

Example user control: Symbion\PhotoImage.xaml

```
<UserControl x:Class="Symbion.PhotoImage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid HorizontalAlignment="Center" VerticalAlignment="Center"
        RenderTransformOrigin="0.5,0.5" Background="White">
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>
        <Border x:Name="photoBorder" Grid.RowSpan="2"
            BorderBrush="Black" BorderThickness="4" />
        <Image x:Name="photoImage" Margin="16,16,16,0"
               Source="Images\success_story.jpg" />
        <TextBlock x:Name="photoText" Grid.Row="1"
            Text="success_story.jpg"
            TextAlignment="Center" Margin="8" />
        <Grid.RenderTransform>
            <ScaleTransform x:Name="scaleTransform"
                ScaleX="0.5" ScaleY="0.5" />
        </Grid.RenderTransform>
    </Grid>
</UserControl>
```

To create a more interactive control add storyboards and triggers. The following are storyboards to animate the scale transform assigned to the root layout of the control. Triggers are assigned to mouse events to start the correct storyboard.

Adding storyboards for control

```
<UserControl.Resources>
    <Storyboard x:Key="ZoomIn">
        <DoubleAnimation
            Storyboard.TargetName="scaleTransform"
            Storyboard.TargetProperty="ScaleX"
            Duration="00:00:00.250"
            To="1" />
        <DoubleAnimation
            Storyboard.TargetName="scaleTransform"
            Storyboard.TargetProperty="ScaleY"
            Duration="00:00:00.250"
            To="1" />
    </Storyboard>
    <Storyboard x:Key="ZoomOut">
        <DoubleAnimation
            Storyboard.TargetName="scaleTransform"
            Storyboard.TargetProperty="ScaleX"
            Duration="00:00:00.150"
            To="0.5" />
        <DoubleAnimation
            Storyboard.TargetName="scaleTransform"
            Storyboard.TargetProperty="ScaleY"
            Duration="00:00:00.150"
            To="0.5" />
    </Storyboard>
</UserControl.Resources>
```

Triggers to activate storyboards

```
<Grid.Triggers>
    <EventTrigger RoutedEvent="MouseEnter">
        <BeginStoryboard Storyboard="{StaticResource ZoomIn}" />
    </EventTrigger>
    <EventTrigger RoutedEvent="MouseLeave">
        <BeginStoryboard Storyboard="{StaticResource ZoomOut}" />
    </EventTrigger>
</Grid.Triggers>
```

Create an application project to test out the above control. Create multiple instances of the control in a panel. It does not matter if all the controls display the same image and title at the moment as we can make this customizable. A user control is just like any UI element. You can use it in **DataTemplate**, **ControlTemplate** and anywhere UI elements can be used.

Testing the user control: Custom2\MainWindow.xaml

```
<UniformGrid Rows="2" Columns="3">
    <sym:PhotoImage />
    <sym:PhotoImage />
    <sym:PhotoImage />
    <sym:PhotoImage />
    <sym:PhotoImage />
    <sym:PhotoImage />
</UniformGrid>
```

3.2 Customizing User Controls

You just need to expose properties to allow the user control to be customized. If the properties are just wrappers for child element properties in the control there will not be any need to define your own dependency properties.

Exposing properties for customization: Symbion\PhotoImage.xaml.cs

```
public partial class PhotoImage : UserControl {
    public PhotoImage() {
        InitializeComponent();
    }
    public ImageSource PhotoSource {
        get { return photoImage.Source; }
        set { photoImage.Source = value; }
    }
    public string PhotoTitle {
        get { return photoText.Text; }
        set { photoText.Text = value; }
    }
}
```

Customizing user control properties

```
<UniformGrid Rows="2" Columns="3">
  <sym:PhotoImage PhotoSource="Images\carddeck.jpg" PhotoTitle="Card Deck" />
  <sym:PhotoImage PhotoSource="Images\korner.jpg" PhotoTitle="Korner" />
  <sym:PhotoImage PhotoSource="Images\moma.jpg" PhotoTitle="Moma" />
  <sym:PhotoImage PhotoSource="Images\summertree.jpg"
        PhotoTitle="Summer Tree"/>
  <sym:PhotoImage PhotoSource="Images\vintage.jpg" PhotoTitle="Vintage" />
  <sym:PhotoImage PhotoSource="Images\success_story.jpg"
        PhotoTitle="Success Story"/>
</UniformGrid>
```

4 Primitive Parts

4.1 Thumb

Primitive parts are base elements that are mostly used as parts in control templates to construct more complex controls. For example, a **Slider** is build from a **Track** that has two **RepeatButton** and a **Thumb**. A normal button only generates a **Click** event while it is pressed while a RepeatButton continue to generate events. A **Thumb** is used to represent a part that can be dragged. Since the thumb itself is a templated control you can fully customize how each looks using a separate template.

ControlTemplate for thumb: Parts1\MainWindow.xaml

```
<Window.Resources>
    <ControlTemplate x:Key="template1">
        <Ellipse Fill="Red" Width="32" Height="32" />
    </ControlTemplate>
</Window.Resources>
```

Use **DragDelta** event on a Thumb to detect it being dragged and update its position in the event handler. You can easily reposition a Thumb in a Canvas using the Top and Left attached properties. Alternatively you can use Margin or TranslateTransform to position the Thumb anywhere. You can also track the start and end of the drag using **DragStarted** and **DragCompleted** events.

Assigning template to thumbs and detect dragging

```
<Canvas>
    <Thumb x:Name="thumb1" Canvas.Left="0" Canvas.Top="0"
        Template="{StaticResource template1}"
        DragDelta="thumb1_DragDelta"/>
    <Thumb x:Name="thumb2" Canvas.Left="0" Canvas.Top="0"
        Template="{StaticResource template1}"
        DragDelta="thumb1_DragDelta"/>
</Grid>
```

We reposition the thumb in the Canvas using Left and Top properties. If you are not using a Canvas the **Margin** property or a **TranslateTransform** can be also used to reposition elements in any panel. If you want to support horizontal dragging, then add **HorizontalChange** value to horizontal position. To support vertical dragging, update add **VerticalChange** value to vertical position. Using Thumb, you can easily create UI that can be dragged around by placing it in a control template to assign to the thumb.

Update position of thumbs

```
private void Thumb1_DragDelta(object sender, DragDeltaEventArgs e) {
    var thumb = (Thumb)sender;
    Canvas.SetLeft(thumb, Canvas.GetLeft(thumb) + e.HorizontalChange);
    Canvas.SetTop(thumb, Canvas.GetTop(thumb) + e.VerticalChange);
}
```

4.2 Popup

Popup is a part commonly used for tooltip, menus and drop-down controls. It display its content in front of controls regardless of layout, even outside a window. By default a Popup is placed below an element. Use **Placement** property to change the location of the popup. Regardless of where the popup is added you can use **PlacementTarget** property to make it appear relative to any element in the UI. **HorizontalOffset** and **VerticalOffset** properties can adjust the popup relative position to Placement.

Declaring and opening a popup: Parts2\MainWindow.xaml

```
<Grid>
    <Popup Name="myPopup" IsOpen="True">
        <Border CornerRadius="8,4,0,16" Background="LightSteelBlue"
            BorderThickness="2" BorderBrush="SteelBlue"    >
            <TextBlock Text="This is a popup!" Foreground="White" Margin="8" />
        </Border>
    </Popup>
</Grid>
```

Placing the popup next to a button

```
<Button x:Name="btn1" Content="Button1"
        Width="60" Height="30" Click="btn1_Click" />
<Popup Name="myPopup" IsOpen="True"
    PlacementTarget="{Binding ElementName=btn1}"
    Placement="Right" HorizontalOffset="-10" VerticalOffset="10">
                :
```

By default a popup is closed and not visible. You can write code to open or close it using the **IsOpen** property. However once it is opened it stays open by default unless you set the **StaysOpen** property to false. When the focus is not on the placement target, the popup will automatically be closed. A popup has built-in animations that you can select using **PopupAnimation** property. However the **AllowsTransparency** property should be set to not force the popup to be always opaque.

Open popup from code

```
private void btn1_Click(object sender, RoutedEventArgs e) {
    if (!myPopup.IsOpen) myPopup.IsOpen = true;
}
```

An initially closed popup

```
<Popup Name="myPopup" IsOpen="False" StaysOpen="False"
   PlacementTarget="{Binding ElementName=btn1}"
    Placement="MousePoint">
            :
```

Using the Fade popup animation

```
<Popup Name="myPopup" IsOpen="False" StaysOpen="False"
    PlacementTarget="{Binding ElementName=btn1}"
    Placement="MousePoint"
    AllowsTransparency="True"
    PopupAnimation="Fade">
            :
```

There is no automatic shadow applied to popups but you can add a shadow effect but since GPU effects is rendered outside of layout, you would need to add spacing within the popup to accommodate for the shadow.

Adding a shadow in the popup

```
<Popup Name="myPopup" IsOpen="False" StaysOpen="False"
    PlacementTarget="{Binding ElementName=btn1}"
    Placement="MousePoint"
    AllowsTransparency="True"
    PopupAnimation="Fade">
    <Border CornerRadius="8,4,0,16" Background="LightSteelBlue"
        BorderThickness="2" BorderBrush="SteelBlue" Margin="4">
        <TextBlock Text="This is a popup!" Foreground="White" Margin="8" />
        <Border.Effect><DropShadowEffect /></Border.Effect>
    </Border>
</Popup>
```