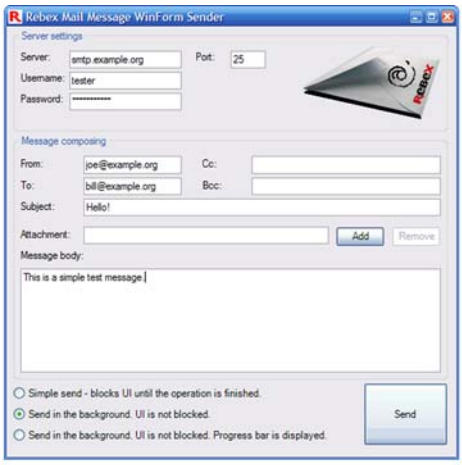
11: Control Templates - basics

11: Control Templates - basics

Motivation

- **Creating elegant GUIs is not easy … or cheap**
  - but is often a requirement in the commercial market



built-in controls are plain and not trivial to style … or create

.. so developers often turned to 3rd party controls to provide competitive UI

WPF offers a new path

- **Visual tree for controls is not programmatically defined**
  - designer can change “look” of controls without changing code

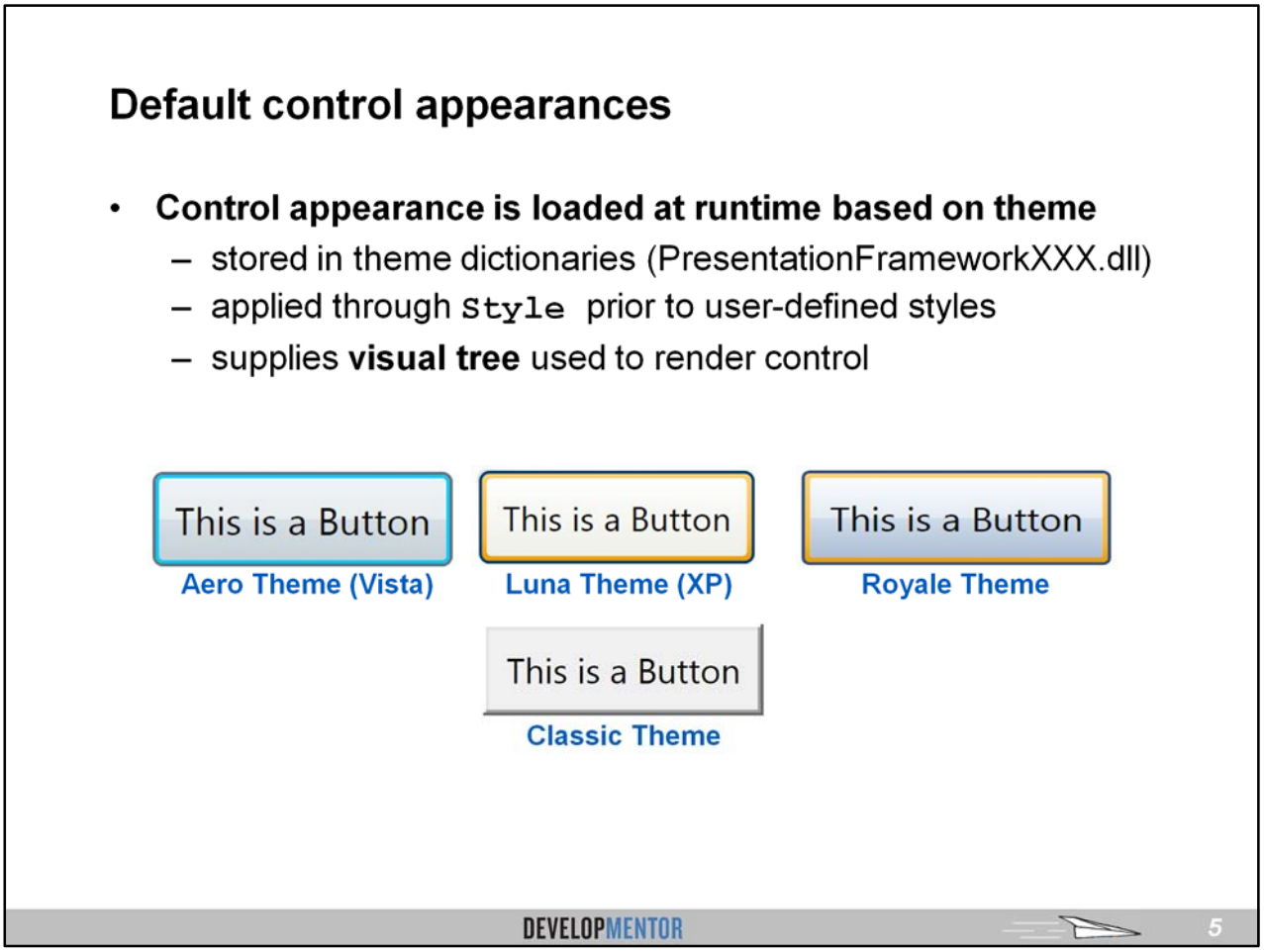
Recall: customizing content

- **One way to achieve this is to add custom content**
  - WPF makes this trivial because anything can be content

creating a button with image and text is trivial in XAML

```
<Button>
   <StackPanel>
      <Image Source="grass.jpg" Width="100" />
      <TextBlock HorizontalAlignment="Center" FontSize="14pt"
                 Foreground="Green">Grass</TextBlock>
   </StackPanel>
</Button>
```

…but what if the entire button needs to be customized?



Microsoft has the system theme dictionaries documented here: http://msdn.microsoft.com/en-us/library/aa358533.aspx

Manually selecting a theme

- **Explicitly loading a theme will force it to be used**
  - can load statically in resources, or dynamically in code

```
<Application xmlns="..." xmlns:x="...">
   <Application.Resources>
      <ResourceDictionary Source=
        "/PresentationFramework.Aero,
         Version=3.0.0.0, Culture=Neutral,
         PublicKeyToken=31bf3856ad364e35;
         Component/themes/aero.normalcolor.xaml" />
   </Application.Resources>
</Application>
```

adds all `Styles` in specified `ResourceDictionary` to display Aero theme

Creating a custom visual tree for controls

- **Controls define their Visual Tree through Control Template**
  - can be changed through `Control.Template` property
  - often set through `Style`


```
<Button Width="200" Height="100" FontSize="12pt"
        Style="{StaticResource GlowingButton}">
   <TextBlock>
      <Italic>This is</Italic><Bold>a
      <Run Foreground="Red">Button</Run></Bold>
   </TextBlock>
</Button>
```


Defining a control template

- **Control templates are defined in XAML**
  - redefines control appearance without impacting behavior


```
<Button Width="200" Height="50" Background="Blue">
   <Button.Template>
      <ControlTemplate>
         <Border BorderBrush="Red" BorderThickness="3"
                 CornerRadius="10" Background="Orange"
                 Height="50" Width="200">
           <TextBlock HorizontalAlignment="Center">
             <Italic>This is</Italic><Bold>a
             <Run Foreground="Red">Button</Run></Bold>
           </TextBlock>
         </Border>
      </ControlTemplate>
   </Button.Template>
</Button>
```


Problems with our template

- **Visual appearance has changed but ..**
  - properties are hard coded, real button properties ignored
  - lost interactivity (rollover effects, "press" effects, etc.)

```
<ControlTemplate>
   <Border BorderBrush="Red" BorderThickness="3"
           CornerRadius="10" Background="Orange"
           Height="50" Width="200">
      <TextBlock HorizontalAlignment="Center">
         <Italic>This is</Italic><Bold>a
         <Run Foreground="Red">Button</Run></Bold>
      </TextBlock>
   </Border>
</ControlTemplate>
```

this should really be defined by the button and not part of the visual style

TemplateBinding

- **Elements of the control template can *data bind* to the parent**
  - allows template to use properties defined on control itself


```
<ControlTemplate>
   <Border CornerRadius="10"
           BorderBrush="{TemplateBinding BorderBrush}"
           BorderThickness="{TemplateBinding BorderThickness}"
           Background="{TemplateBinding Background}"
           Height="{TemplateBinding Height}"
           Width="{TemplateBinding Width}">
      ...
   </Border>
</ControlTemplate>
```

`{TemplateBinding}` provides optimized[1] form of:

```
{Binding RelativeSource={RelativeSource TemplatedParent}}
```

DEVELOPMENTOR 10

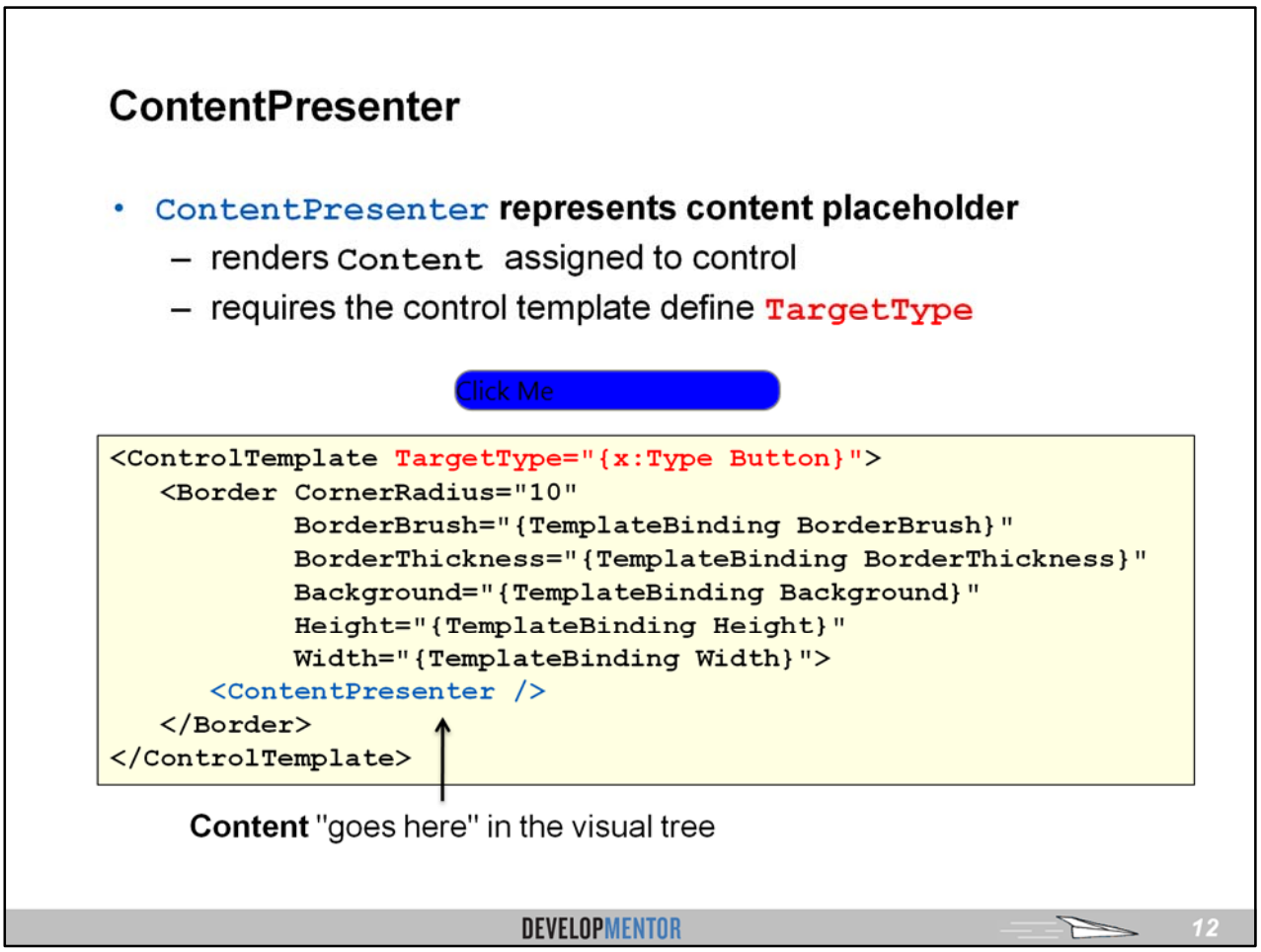
[1] Be aware that {TemplateBinding} does not support coercing values – it requires that the source and target are the proper types. If you need a TypeConverter to run, then you will need to use the full {Binding} syntax shown on the slide.

Defining Content

- **Another problem is the content …**
  - what if we wanted an image and text?

```
<ControlTemplate>
   <Border BorderBrush="Red" BorderThickness="3"
           CornerRadius="10" Background="Orange"
           Height="50" Width="200">
      <TextBlock HorizontalAlignment="Center">
         <Italic>This is</Italic><Bold>a
         <Run Foreground="Red">Button</Run></Bold>
      </TextBlock>
</Border>
</ControlTemplate>
```

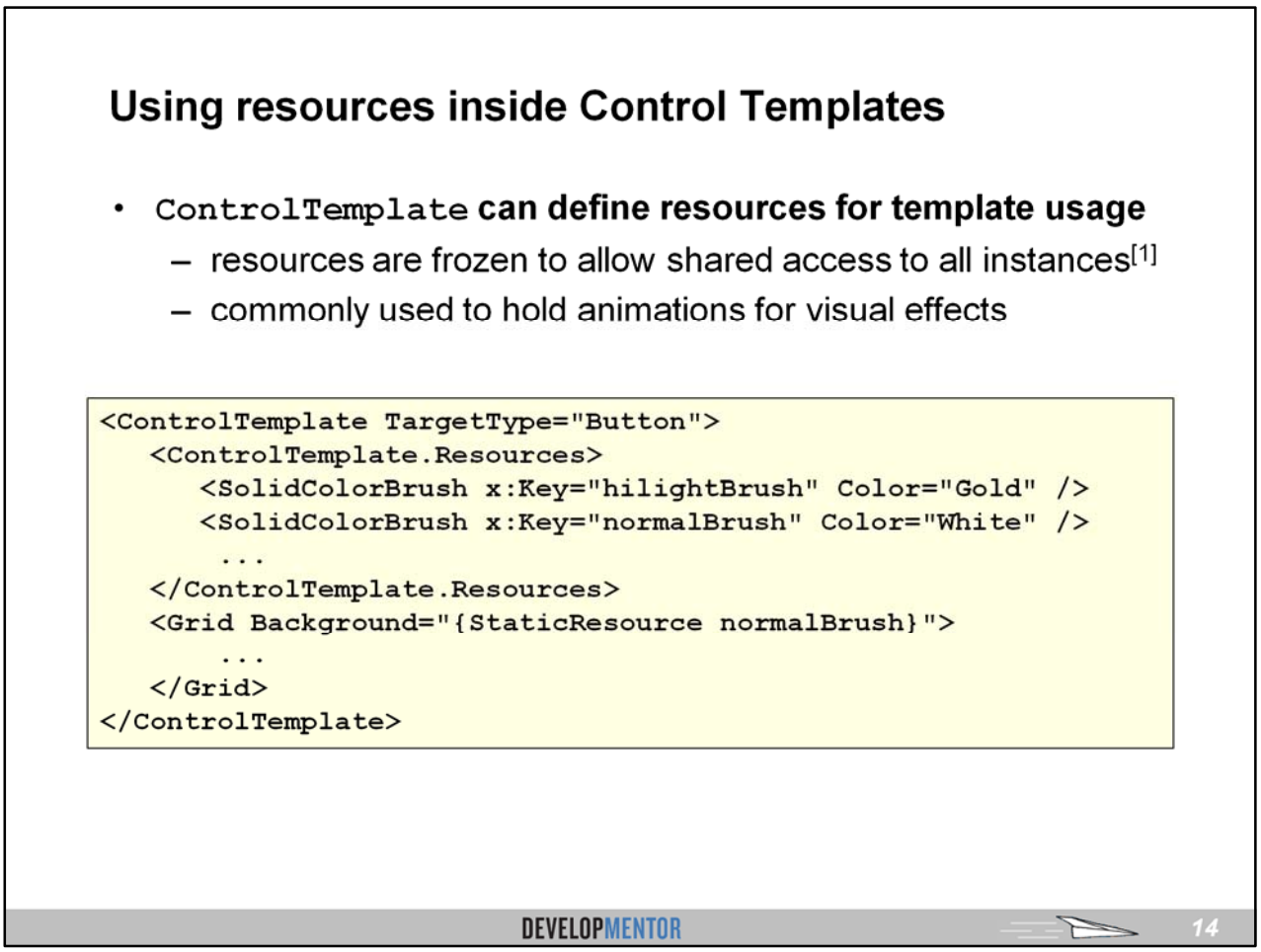
this should also be defined by the button and not part of the visual style



Setting properties on ContentPresenter

- **Some properties are not applied automatically**
  - alignment and positioning need adjustment


```
<ControlTemplate TargetType="{x:Type Button}">
   <Border ...>
      <ContentPresenter
         Margin="{TemplateBinding Padding}"
         HorizontalAlignment="{TemplateBinding
                     HorizontalContentAlignment}"
         VerticalAlignment="{TemplateBinding
                     VerticalContentAlignment}" />
      </Border>
</ControlTemplate>
```

[1] Many underlying elements derive from Freezable – this is a base class that provides the ability to "freeze" all properties on the type. This allows an optimization with resources such as animations and brushes. The downside is that once an object is frozen, it's property values cannot be changed. This manifests itself in animations with an error like:

"Failed object initialization (ISupportInitialize.EndInit). Cannot freeze this Storyboard timeline tree for use across threads."

In the above example, this error would happen if you attempted to animate the brush color. The workaround is define the elements you want to animate as resources – for example the above definition would be changed to:

<Color x:Key="hilightColor" Color="Gold" />

<SolidColorBrush x:Key="hilightBrush" Color="{StaticResource hilightColor}" />

Then the color of the brush could be animated.

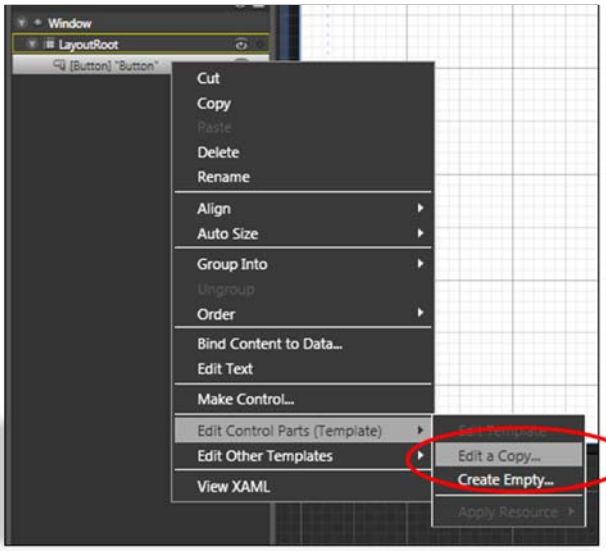
Sharing templates

- **Control templates are normally included as part of the style**
  - can be named, or provided as part of the default style
  - `Control.Template` assigned like any other property

```
<Style x:Key="customButtonStyle" TargetType="Button">
   <Setter Property="Template">
      <Setter.Value>
         <ControlTemplate TargetType="{x:Type Button}">
            <Border ...>
               <ContentPresenter ... />
            </Border>
            <ControlTemplate.Triggers />
         </ControlTemplate>
      </Setter.Value>
   </Setter>
   ...
</Style>
```


Templating controls with Blend

- **Blend is a great way to create control templates**
  - provides a drag+drop UI for building complete templates



right-click on the control in the Objects and Timeline and select Edit Control Parts (Template) and then Edit a Copy

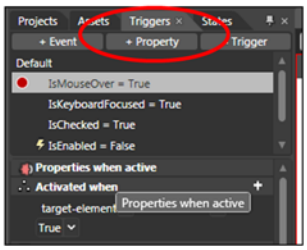
Interacting with templated controls

- **Once a control is templated it loses all visual behaviors**
  - must be supplied by template designer
- **Common behaviors that need definition include**
  - rollover effect (`IsMouseOver`)
  - "push" effect (`IsPressed`)
  - disabled state (`IsEnabled`)
  - focus (`IsKeyboardFocused`)

Pressed state shrinks control and adds "emboss" bitmap effect

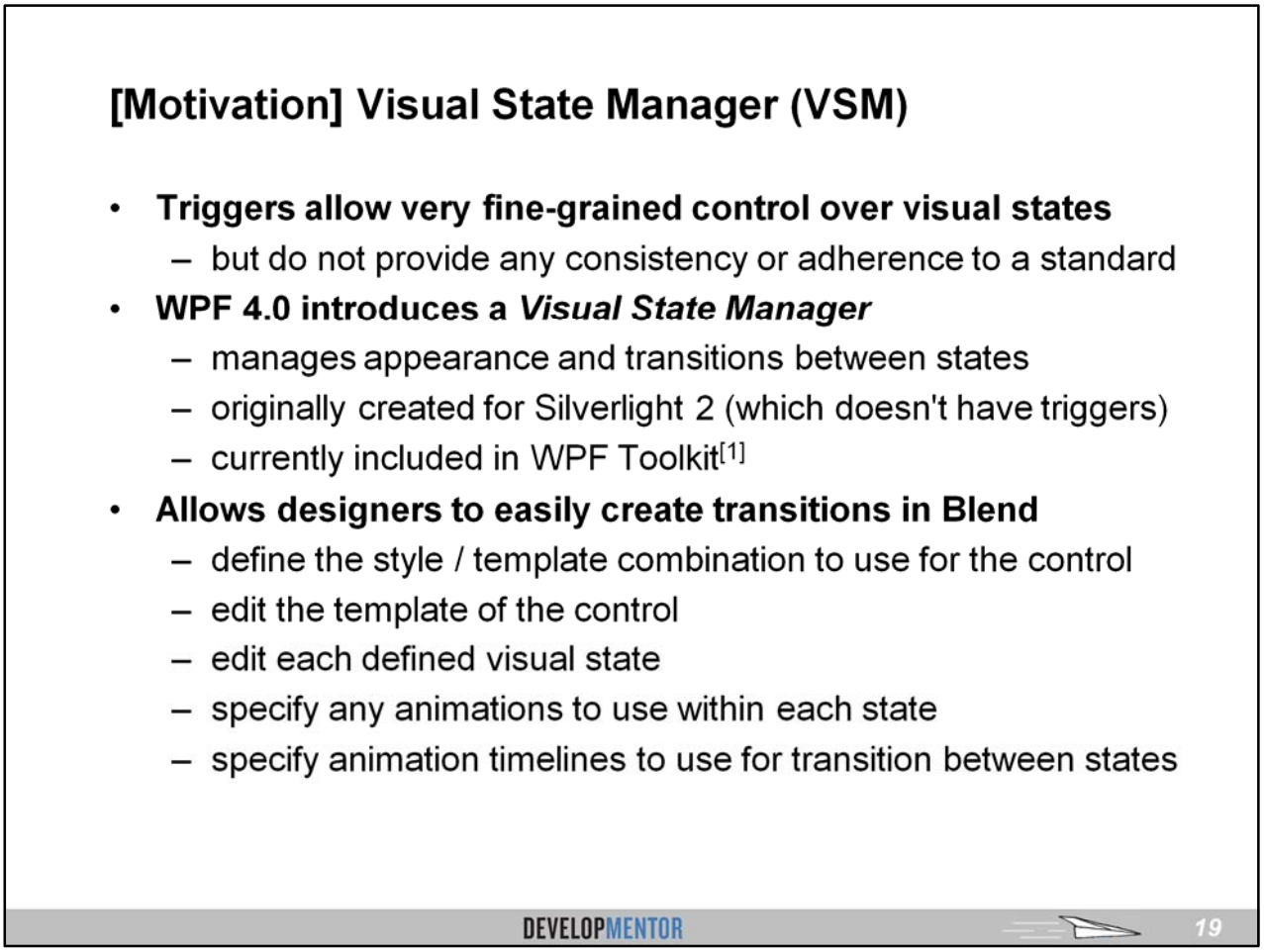
Adding triggers to templated controls

- **Triggers can be defined as part of control template**
  - adds dynamic visual behavior to control being templated
  - includes property test and one or more property setters



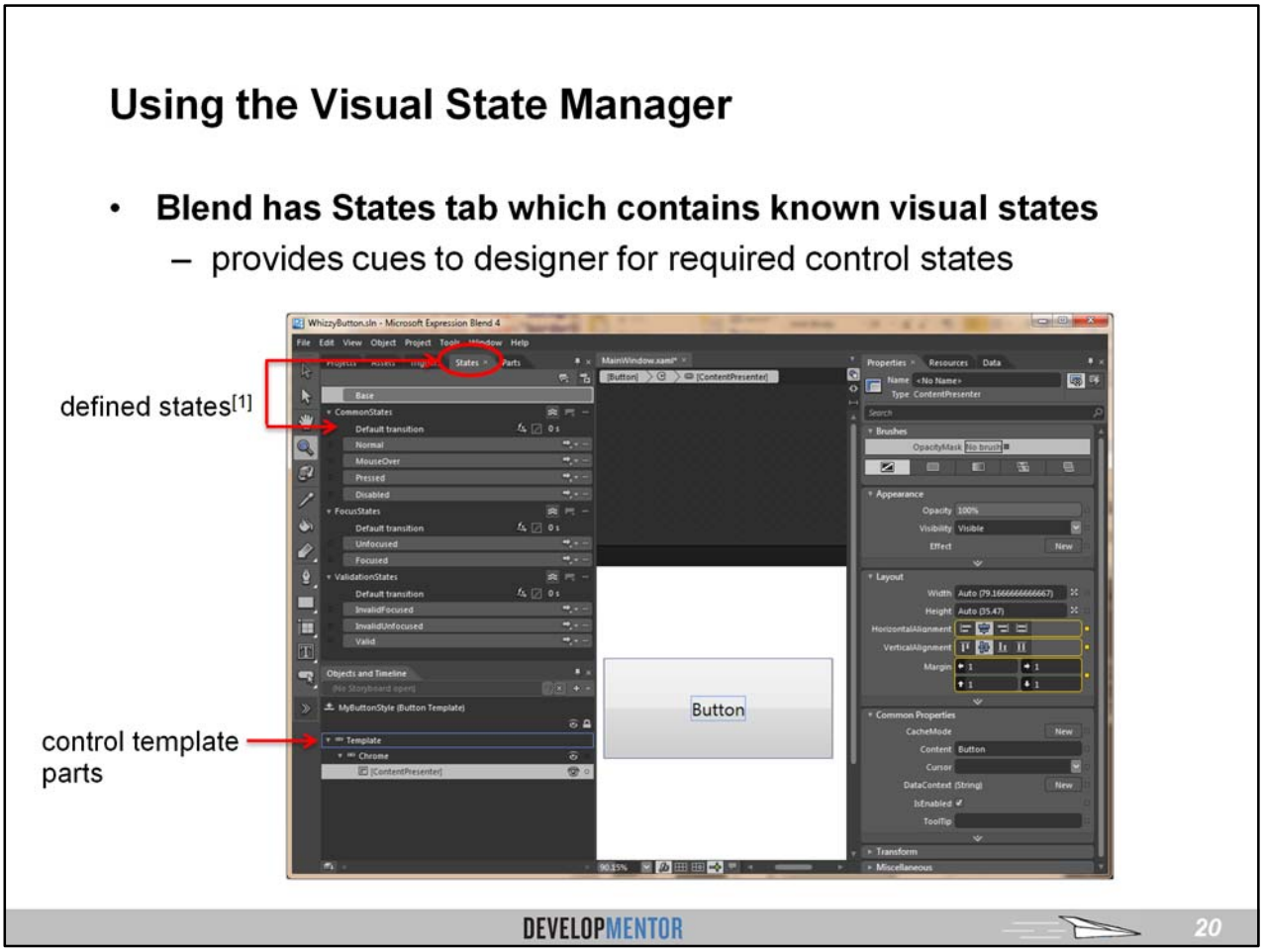
add property trigger and then set appropriate properties during recording…

```
<ControlTemplate TargetType="Button">
   <ControlTemplate.Triggers>
      <Trigger Property="IsPressed" Value="True">
         <Setter Property="RenderTransform" Value="..." />
      </Trigger>
   </ControlTemplate.Triggers>
</ControlTemplate>
```

[1] Download the WPF toolkit from http://www.codeplex.com/wpf

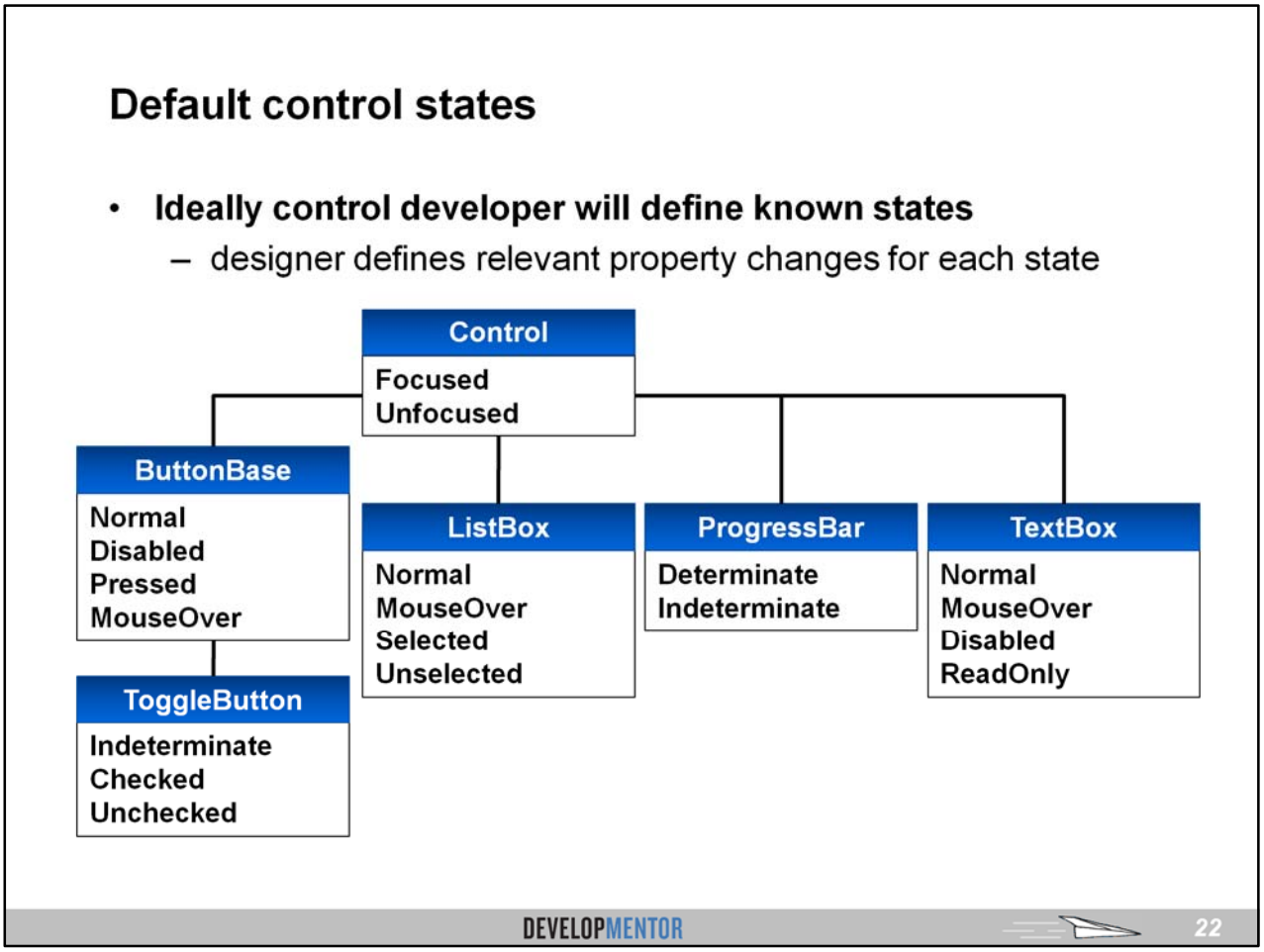
11: Control Templates - basics



[1] Note that at this point, while VSM is in CTP status, the default states is empty. This is because no VSM states are defined on the button template – WPF itself doesn't use VSM for state transitions. This is in contrast to Silverlight where you will see default states.

Visual State Manager [groups]

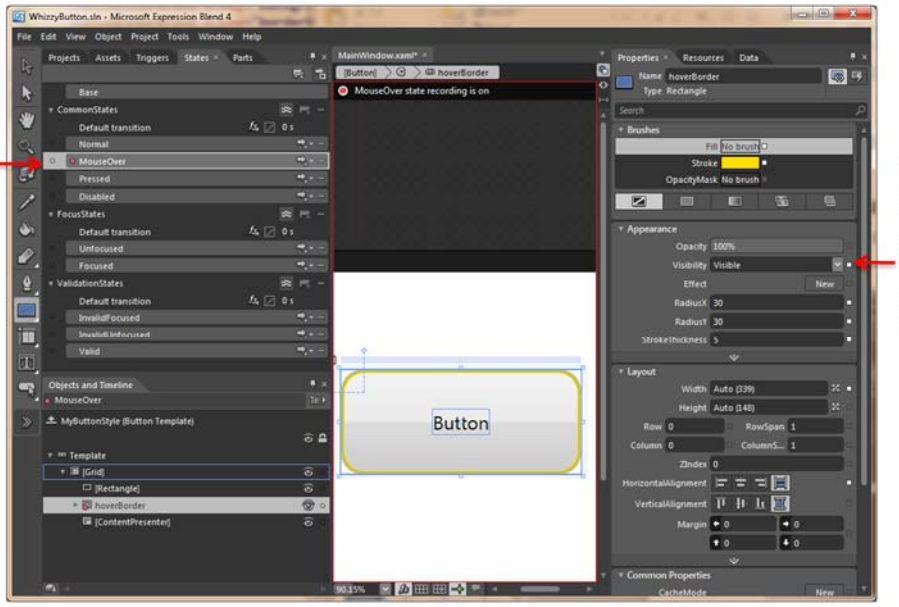
- **Visual states are placed inside a `VisualStateGroup`**
  - states within a group are mutually exclusive
    - e.g. Normal and Disabled
- **A template can contain multiple groups**
  - states between groups can be logically combined
    - e.g. Normal + Focused
- **Control author needs to carefully plan the states**
  - to make it easier for a designer to craft the visuals


Default control states
• Ideally control developer will define known states
– designer defines relevant property changes for each state
Control
Focused
Unfocused
ButtonBase
Normal
Disabled
Pressed
MouseOver
ToggleButton
Indeterminate
Checked
Unchecked
ListBox
Normal
MouseOver
Selected
Unselected
ProgressBar
Determinate
Indeterminate
TextBox
Normal
MouseOver
Disabled
ReadOnly
DEVELOPMENTOR
22

Defining visual state changes

- **Edit the visual appearance for each state**
  - Blend will create the appropriate *animations*

click to change the appearance for the **MouseOver** state



change the hover border's **Visibility** so that it appears

DEVELOPMENTOR 23

Animation details

- **WPF Animations change the value of a property over time**
  - works with dependency properties
  - typically activated through triggers or VSM
- **Animations are timer resolution independent**
  - uses linear interpolation to obtain values
  - better hardware produces smoother animations not faster ones
- **Types of things you might animate**
  - position, scale , colors, opacity, rotation, etc.
- **WPF supports different types of animations**
  - discrete, key-frame and path-based

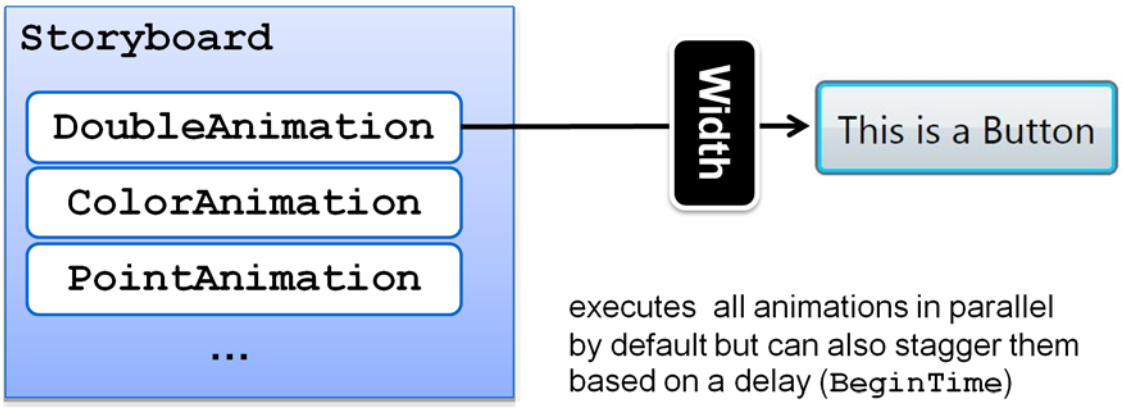
Creating animations directly in XAML

- **Event Triggers are often used to start animations in XAML**
  - start and stop storyboards which contain animations

```
<Button Width="50" Content="Click Me">
  <Button.Triggers>
     <EventTrigger RoutedEvent="Button.Loaded">
        <EventTrigger.Actions>
          <BeginStoryboard>
              <Storyboard>
                <DoubleAnimation From="100" To="400"
                    Duration="0:0:0.5" RepeatBehavior="2x"
                    Storyboard.TargetProperty="Width" />
              </Storyboard>
          </BeginStoryboard>
        </EventTrigger.Actions>
     </EventTrigger>
  </Button.Triggers>
</Button>
```


Storyboards

- **Storyboards coordinate animations**
  - generally stored as resources
  - provide capability to start, stop, pause and continue animations
  - provide attached properties to identify target and property



executes all animations in parallel by default but can also stagger them based on a delay (`BeginTime`)

Identifying the target object and property

- `Storyboard.TargetName` **identifies the XAML element**
  - any object can be specified
- `Storyboard.TargetProperty` **identifies path to the property**
  - must end on a `DependencyProperty`

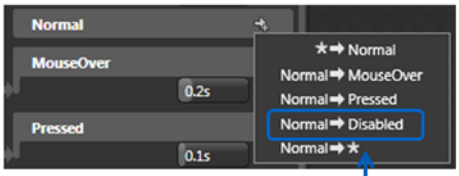
```
<Storyboard Storyboard.TargetProperty="Width">
   <DoubleAnimation From="100" To="300" />
   <ColorAnimation From="Red" To="Blue"
      Storyboard.TargetName="gradientStop"
      Storyboard.TargetProperty="Offset" />
</Storyboard>
```

target property can be inherited by all animations in storyboard or can be assigned directly to each animation

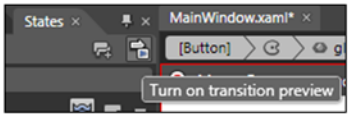
Visual State Manager [transitions]

- **Transitions can be applied between states**
  - to/from any state
  - to/from a specific state
- **Provides an override for the animation duration specific to the state transition**

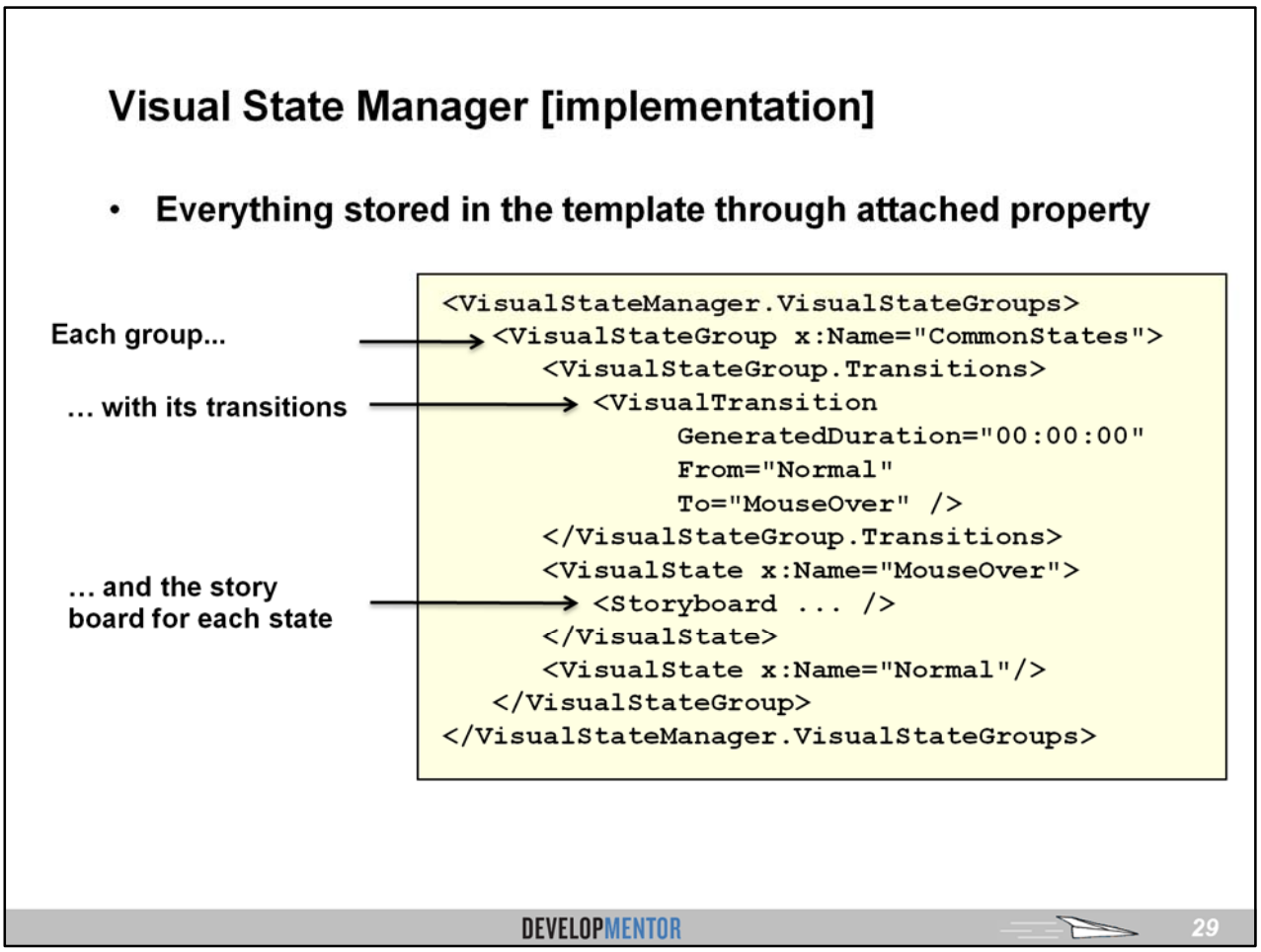
take 1/2 seconds to animate to **Pressed** from any state



select this to define the transition time for moving from **Normal** to **Disabled**



Blend provides preview support so designer can see transition in action



Visual State Manager [code perspective]

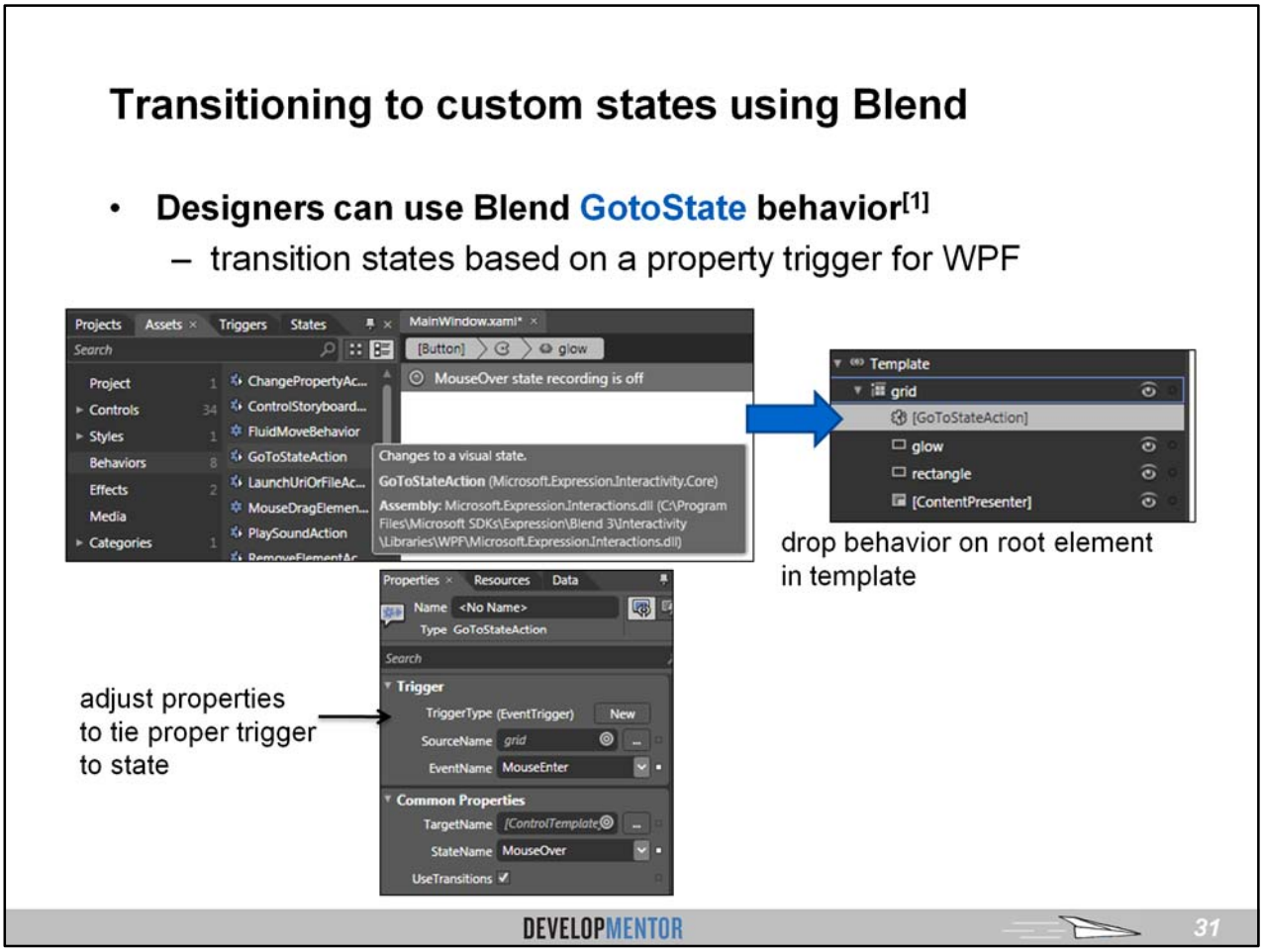
- **Built-in controls can automatically transition states**
  - as long as state names are recognized
- **Custom states require developer interaction**
  - typically in response to an event or property change[1]
- **Use `VisualStateManager.GoToState` to force transitions**
  - specify the object, the state and whether to use transition animations

```
void OnClickExpander(object sender, RoutedEventArgs e)
{
   if (graph.IsExpanded)
      VisualStateManager.GoToState(this, "ExpandedGraph", true);
   else
      VisualStateManager.GoToState(this, "CollapseGraph", true);
}
```

DEVELOPMENTOR 30

[1] There is also a blend behavior – GotoState which can be used to transition between visual states.

11: Control Templates - basics



[1] This behavior is included in the Expression Blend behaviors samples.

Visual State Manager final notes

- **VSM can be used with *all* types of control**
  - which includes user controls
- **You can therefore use VSM in your application to:**
  - show and hide menu options
  - show and hide panels (e.g. similar to the VS IDE tool windows)
  - implement interesting wizard animations
    - e.g. wizard "pages" that slide or resize
- **Overall, it makes it much easier for designers to design compelling user interfaces**

Summary

- **Control Templates decouple the visual structure from the control functionality**
  - allowing easier "custom" controls defined in XAML
- **Visual Designer role can now build visual of controls**
  - replaces existing visual tree
  - not necessary to involve procedural code in most cases
- **Use Visual State Manager to define various states and transitions**
  - can be defined completely in Blend by designer role

