Entwicklung interaktiver Systeme

M. Sc. Tobias Puderer – WS 22/23

Übung 5: Styles, Trigger und Animationen (Event Trigger)

1. Ähnlich wie durch den Einsatz von CSS in der Webprogrammierung ist es möglich, Oberflächenelementen einen bestimmten Style zuzuweisen, der das Erscheinungsbild dieses Elements (oder wahlweise auch jedes Elements des gleichen Typs) verändert. Dies ist hilfreich um ein konsistentes Design der Anwendung zu schaffen, ohne dabei den Code mit Redundanzen aufzublähen.

   Zur Definition eines Styles wird ein Objekt des Typs `Style` verwendet, der an der `Resources`-Eigenschaft eines jeden Oberflächenelements gepflegt werden kann. Die `Resources`-Eigenschaft stellt die jeweilige Ressource (z.B. den Style) dem Element und allen seinen Kind-Elementen zur Verfügung, an dem sie gepflegt wurde.

   In Codelisting 1 sind Beispiele für die Verwendung von Styles zu sehen. Die Styles sind in der `Resources`-Eigenschaft des `Window`s gepflegt, daher gelten die Styles im Kontext des kompletten Fensters. Durch die Eigenschaft `TargetType` wird jeweils der Typ bestimmt, für welchen der Style gilt. Mit der Eigenschaft `x:Key` kann dem Style ein eindeutiger Name zugewiesen werden.

   Mittels des Typs `Setter` kann an einer Eigenschaft (`Property`) des Ziel-Elements ein neuer Wert (`Value`) gesetzt werden. Es können auch mehrere Setter definiert werden, um damit mehrere Eigenschaftswerte zur gleichen Zeit zu setzen.

   i) Alle `Button`s im Fenster besitzen die Breite 110.

   ii) Alle `Button`s im Fenster, welchen explizit der Style mit dem Namen „MyButtonStyle“ zugewiesen wurde, besitzen einen blauen Hintergrund.

   iii) Wie ii) aber die Angabe des Zieltyps entfällt. Daher ist die Anwendung nicht auf Button Elemente beschränkt. Alle Objekte mit einer Background-Eigenschaft, denen explizit der Style mit dem Namen „MyButtonStyle2“ zugewiesen wurde, besitzen einen roten Hintergrund. Beachten Sie, dass in diesem Fall die Eigenschaft voll qualifiziert geschehen muss, was im Beispiel durch die Angabe des Typs `Control` geschieht.

   Ein Style wird mit Hilfe der `Style`-Eigenschaft und unter Zuhilfenahme des Schlüsselworts `StaticResource` eingebunden indem auf den Wert der jeweiligen `x:Key` Eigenschaft verwiesen wird (Fall ii und iii in Codelisting 1), sofern er nicht ohnehin für jedes Element eines Typs definiert wurde (Fall i in Codelisting 1).

```
<Window.Resources>
  <!-- [i] Gilt für alle Buttons in diesem Fenster -->
  <Style TargetType="Button">
     <Setter Property="Width" Value="110"/>
  </Style>

  <!-- [ii] Gilt (NUR) für Buttons, die auf den Stil „MyButtonStyle“ verweisen -->
  <Style x:Key="MyButtonStyle" TargetType="Button">
     <Setter Property="Background" Value="Blue"/>
  </Style>

  <!-- [iii] Geht auch; kann dann für JEDEN Elementtyp gesetzt werden -->
  <Style x:Key="MyButtonStyle2">
     <Setter Property="Control.Background" Value="Red"/>
  </Style>

</Window.Resources>

 …
<!-- Auf Resourcen wird über die Markup Erweiterung StaticResource zugegriffen-->
<Button Style="{StaticResource MyButtonStyle}">Click me</Button>
```

Codelisting 1: Zuweisung eines Stils für Buttons

a) Versuchen Sie, das Beispiel in Codelisting 1 für alle drei Fälle zum Laufen zu bringen. Erweitern Sie dann den Fall ii) (Style „MyButtonStyle“) durch die zusätzliche Angabe einer Höhe von 150.

b) Es ist auch möglich, einen Style als Grundlage eines anderen Styles zu verwenden, d.h. ein Style „erbt“ Eigenschaften eines anderen Styles und kann weitere hinzufügen. Dies wird durch das Attribut `BasedOn` an dem erbenden `Style` Element festgelegt. Dieses verlangt die Angabe des eindeutigen Schlüssels (`x:Key`) eines anderen Styles, von dem geerbt werden soll.

   Definieren Sie fünf `TextBox` Elemente. Jede davon soll einen roten Hintergrund sowie eine Breite von 200 Pixel und eine Höhe von 50 Pixel besitzen. Zwei davon sollen zusätzlich die Schriftfarbe „weiß“ und Schriftgröße 22 erhalten. Realisieren Sie dies über Styles unter Zuhilfenahme von `BasedOn`. Erstellen Sie hierzu zunächst den Style mit dem allgemeinen Fall der ersten drei `TextBox`-Elemente und ergänzen Sie für die beiden anderen einen zweiten Style mit den zusätzlichen Anforderungen (Schriftfarbe und –größe). Recherchieren Sie ggf. die Verwendung von `BasedOn`.

c) Angenommen, wir haben eine Applikation mit mehreren Fenstern und ein selbst definierter Stil für einen Elementtypen ihrer Wahl (z.B. für `Textbox`) soll fensterübergreifend für die komplette Applikation gelten. Überlegen Sie, in

welchen Dateien eines WPF-Projekts XAML Code definiert werden kann oder sehen sie noch einmal in Übungsblatt 2 nach.

An welcher Stelle ist es Ihrer Meinung nach sinnvoll die Definition des Styles vorzunehmen? (TIPP: Es genügt bereits in Visual Studio ein neues WPF `Window` zu erstellen und dort ein Element einzufügen, für dessen Typ ein übergreifender Stil definiert wurde. Wenn Sie alles richtig gemacht haben, werden Sie dies bereits im Visual Studio Designer sehen ☺)

d) WPF unterstützt die Definition von Vorlagen, sog. *Templates*, welche - noch stärker als Styles - in der Lage sind, die Erscheinungsform eines Oberflächenobjekts zu verändern. Hierbei wird der komplette visuelle Baum durch eine Kombination anderer Elemente ersetzt. Die folgende Definition verschafft dem Standardbutton abgerundete Ecken. Templates werden ebenfalls innerhalb der `Resources`-Eigenschaft definiert.

```
<Window.Resources>
   <ControlTemplate x:Key="ButtonTemplate">
     <Grid Width="100" Height="100">
       <Border BorderThickness="1" Grid.Row="0" Grid.ColumnSpan="2"
          CornerRadius="30,30,30,30" BorderBrush="Black" Background="LightGreen">
       <!–Template Binding erlaubt die Referenzierung von Eigenschaften des
       Zieltyps innerhalb der Templatedefinition. Hier: Buttonbeschriftung -->
          <TextBlock Text="{TemplateBinding Button.Content}"
       HorizontalAlignment="Center" VerticalAlignment="Center" FontSize="40"/>
       </Border>
      </Grid>
    </ControlTemplate>
</Window.Resources>
 …
<!-- Auf Resourcen wird über die Markup Erweiterung StaticResource zugegriffen-->
<Button Template="{StaticResource ButtonTemplate}">OK</Button>
```

Codelisting 2: Button mit abgerundeten Ecken mit Hilfe von Templates

Erstellen Sie ein Template, das den Button in die Form (und gerne auch Farbe) einer Zitrone bringt. Was müssen Sie an dem Beispiel in Codelisting 2 verändern um dies zu erreichen?

2. Im vorherigen Übungsblatt wurde der Hintergrund und die Schriftfarbe einer **TextBox** mit einer Kombination von XAML und Code Behind automatisch geändert, wenn mit der Maus über die **TextBox** gefahren bzw. der Bereich verlassen wurde (Stichwort *MouseEnter* und *MouseLeave*).

   Eine Alternative für ein solches Vorgehen bieten sog. *Trigger*. Diese lassen sich direkt in XAML definieren, ohne dass eine Handler-Methode im Code-Behind hinterlegt bzw. C# Code geschrieben werden muss. Viele Elemente bieten hierfür die Eigenschaft *IsMouseOver*, die automatisch auf diese Art von Ereignis reagiert. Die Definition erfolgt direkt an einem **Style**-Element. Anders als in der vorherigen Implementierung muss sich um die Rückfärbung in diesem Fall nicht gekümmert werden. Das Beispiel in Codelisting 3 rotiert den Button beim Überfahren mit der Maus um 45°.

```
<Window.Resources>
  <Style TargetType="Button">
    <Style.Triggers>
       <!-- Rotiert den Button um 45°, wenn die Maus ihn berührt-->
      <Trigger Property="Button.IsMouseOver" Value="true">
       <!-- Setter Property definiert die zu ändernde Eigenschaft (z.B.
       Hintergrundfarbe)-->
       <!-- Setter Value definiert den Wert, welchen die Eigenschaft annehmen
       soll-->
         <Setter Property="RenderTransform">
           <Setter.Value>
               <RotateTransform Angle="45"/>
           </Setter.Value>
        </Setter>
      </Trigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```

Codelisting 3: Rotation des Elements um 45° beim Überfahren mit der Maus

a) Setzen Sie Übung 4.1 d) erneut mit Hilfe von Triggern um, d.h. realisieren Sie einen Wechsel der Hintergrund- und Textfarbe beim Überfahren einer **TextBox** mit der Maus.

b) Ein Trigger ist nicht auf Boolean-Eigenschaften beschränkt, sondern kann auch dann ausgelöst werden, wenn eine beliebige Eigenschaft einen bestimmten Wert enthält z.B. eine Zahl oder ein Text. Erstellen Sie mit diesem Wissen eine TextBox, deren Hintergrund den Farbverlauf *LightBlue* zu *Blue* erhält, wenn der eingegebene Text der Zeichenkette „EIS“ entspricht.

c) Ermitteln Sie, welchen Zweck ein `MultiTrigger` erfüllt und erstellen Sie ein kleines, selbstgewähltes Beispiel.

3. Animation

WPF bietet die Möglichkeit, Animationen einzubinden, die durch Benutzeraktionen (z.B. Mausklick) auslösbar sind. Um eine Animation zu definieren, wird ein Storyboard benötigt, in welchem die auszuführenden Schritte aufgeführt werden.

Erneut wird mit Triggern gearbeitet, die z.B. an einem Stil oder direkt am Oberflächenelement bekannt gemacht werden können. Im Beispiel handelt es sich dieses Mal allerdings um **EventTrigger**, welche nicht mit Eigenschaften, sondern mit einem RoutedEvent (z.B. Button.Click) verknüpft werden können. Wird ein Stil verwendet, können auch Property Trigger verwendet werden, die Sie bereits in Teilaufgabe 2 a) kennengelernt haben.

```
<Button.Triggers>
  <EventTrigger RoutedEvent="Button.Click">
    <EventTrigger.Actions>
      <BeginStoryboard Name="GrowShrink" >
        <Storyboard>
       <!--Verbreitere den Button ausgehend von der Startgröße 100 (From) zu 200
       (To) innerhalb 1 Sekunde (Duration) und mache die Änderung anschließend
       rückgängig (Autoreverse)-->
          <DoubleAnimation Storyboard.TargetProperty="Width"
                           From="100"
                           To="200"
                           Duration="0:0:1"
                           AutoReverse="True" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger.Actions>
  </EventTrigger>
</Button.Triggers>
```

Ein Storyboard erwartet eine Animation. Diese bezieht sich i.d.R. auf eine bestimmte Eigenschaft eines Oberflächenelements z.B. die Breite. Die Art der Animation ist vom Typ der Eigenschaft abhängig. Im Beispiel kann dem Wert (`width`) ein Wert vom Datentyp **Double** zugewiesen werden. Ebenso existieren Animationen für **Integer**, **Byte**, …

```
<!–Actions für das Stoppen/Pausieren/Fortsetzen der Animation. Diese können – je
nach Situation - innerhalb eines Triggers gestartet werden -->
<StopStoryboard BeginStoryboardName="GrowShrink" />
<PauseStoryboard BeginStoryboardName="GrowShrink" />
<ResumeStoryboard BeginStoryboardName="GrowShrink" />
```

Ein `Storyboard` wird innerhalb eines `BeginStoryboard` Elements gekapselt oder von diesem angesprochen (z.B. wenn die Animation als Ressource hinterlegt ist), d.h. beim Stoppen (`StopStoryboard`), Pausieren (`PauseStoryboard`) und Fortsetzen (`ResumeStoryboard`) der Animation wird der Name des `BeginStoryboard` Elements verwendet (was zugegebenermaßen wenig intuitiv ist). Innerhalb eines Storyboards können sich auch mehrere Animationen befinden, die sich auf unterschiedliche Eigenschaften auswirken.

a) Erstellen Sie nun einen Button, der bei Klick innerhalb 1/10 Sekunde um 20 Pixel kleiner wird (je in der Länge und Breite) und anschließend die Ausgangsgröße wieder annimmt.

   Hinweis: Recherchieren Sie in dem Zusammenhang die Eigenschaft „By“ der `DoubleAnimation`.

b) Trigger können z.B. auch am umgebenden Container (z.B. `Grid` oder `StackPanel`) definiert werden. Die Quelle des Triggers bzw. des RoutedEvents muss in diesem Fall allerdings explizit referenziert werden (Vgl. Eigenschaft SourceName im XAML Code in der ZIP Datei).

   Da der Trigger nun nicht direkt zu animierenden Oberflächenelement hängt, sondern am umgebenden Container, muss dem Storyboard mitgeteilt werden, welches Oberflächenelement als Ziel der Animation sein wird.

```
<Button Name="MeinButton" Content="Klick mich"/>
  …

<BeginStoryboard Name="ButtonAnimationStart">
   <Storyboard TargetName="MeinButton">
    …
   </Storyboard>
</BeginStoryboard>
```

   Kopieren Sie die die zwei Bilddateien (PNG) und die Quelldateien aus OLAT in ihr Projektverzeichnis. Das Ziel ist es, die schlafende Katze bei einem Klick auf den „Start“ Button (endlos) atmen zu lassen. Dies können Sie simulieren, indem Sie den Torso gleichmäßig verkleinern und vergrößern. Ein Klick auf den „Stop“ Button soll die Animation stoppen.