bbv Software Services AG

# WPF – Styles und Templates

Michael Albertin

## *Wichtiger Hinweis*

*Die in diesem Booklet enthaltenen Angaben wurden nach bestem Wissen auf Grund aktueller Quellen zusammengestellt. bbv Software Services AG (bbv) gibt keine Garantie für die Vollständigkeit und Genauigkeit der gemachten Angaben. bbv lehnt jede Verantwortung für die im Booklet gemachten Aussagen ab. Für verbindliche Auskünfte zu praktischen Problemen oder bevor Entscheidungen getroffen werden, sollten unbedingt Fachleute von bbv konsultiert werden. bbv übernimmt keine Verantwortung für Verluste, die durch Anwendung oder Nichtanwendung des in dieser Veröffentlichung enthaltenen Materials entstehen.*

*Die im Booklet verwendeten Software- und Hardwarebezeichnungen, Firmen- und Produktenamen (insbesondere bbv, Scrum, XP, Crystal Clear) sind in den meisten Fällen eingetragene Marken und unterliegen als solche den gesetzlichen Bestimmungen.*

# Inhalt



# 1. Übersicht

Dieses Booklet beschäftigt sich mit den Grundlagen von Styles und Templates und zeigt anhand eines Beispielprojekts und vieler Codeausschnitte, wie man schnell und einfach die Vorteile von WPF nutzen kann.

Mit Hilfe von WPF besteht die Möglichkeit, die Oberfläche der Anwendung sehr flexibel und weitreichend an die spezifischen Fach- und Anwender-Anforderungen anzupassen. Die folgenden Kapitel gehen auf die wichtigsten Themen und Möglichkeiten zur Veränderung des UIs ein und zeigen mögliche Anwendungsbeispiele.

Zur Visualisierung einiger der gezeigten Möglichkeiten wird uns ein Fenster aus einer fiktiven Filmverwaltung begleiten.


Alle Beispiele und Code-Ausschnitte in diesem Booklet beruhen auf Visual Studio 2008 SP1 und dem .NET-Framework 3.5 SP1.

# 2. Ressourcen

## 2.1. Überblick

Eine Ressource ist ein vordefiniertes Element, identifiziert durch einen eindeutigen Schlüssel.

Abhängig vom Ort der Deklaration kann diese Ressource in der gesamten Applikation nur im aktuellen Fenster oder sogar auf ein spezifisches Control begrenzt angesprochen und verwendet werden.

Obwohl nahezu jedes instanzierbare Objekt als Ressource dienen kann, sind die folgenden Typen die am häufigsten verwendeten:

- Style
- DataTemplate, ControlTemplate
- IValueConverter, IMultiValueConverter
- Color
- SolidColorBrush, LinearGradientBrush

Das folgende Beispiel zeigt eine typische Deklaration von Ressourcen in einem `Window`:

```
<Window.Resources>
     <Converter:GenderImageConverter x:Key="GenderImageConverter" />
     <Converter:GenderStringConverter x:Key="GenderStringConverter"
/>
     <Color x:Key="BorderBrush_DarkColor">#FF032D27</Color>
     <Color x:Key="BorderBrush_LightColor">#FF319184</Color>
     <LinearGradientBrush x:Key="BorderBrush" EndPoint="1,0.5"
      StartPoint="0,0.5">
          <GradientStop Color="{StaticResource Border-
Brush_DarkColor}"
           Offset="0"/>
```

```
        <GradientStop Color="{StaticResource Border-
Brush_LightColor}"
          Offset="1"/>
    </LinearGradientBrush>
</Window.Resources>
```

Neben der Deklaration von Ressourcen ist im obigen Beispiel auch die Verwendung der Ressourcen über ihren Namen ersichtlich.

Zusätzlich zum eindeutigen Schlüssel (Eigenschaft `x:Key`) kann auch optional noch bestimmt werden, dass für jede Zuweisung eine neue Instanz der Ressource (Eigenschaft `x:Shared`) erstellt werden soll:

```
<Button x:Key="aButton" x:Shared="false" />
```

## 2.2 Statische und dynamische Ressourcen

Die Zuweisung von Ressourcen zu einer Eigenschaft erfolgt mit den Schlüsselworten `StaticResource` und `DynamicResource`.

Mit der nachfolgenden Anweisung wird der Eigenschaft `Color` die Ressource mit dem Namen `BorderBrush_DarkColor` zugewiesen:

```
Color="{StaticResource BorderBrush_DarkColor}"
```

Syntaktisch ähnlich zeigt sich diese Anweisung und erreicht ebenfalls, dass `BorderBrush_DarkColor` der `Color`-Eigenschaft zugewiesen wird:

```
Color="{DynamicResource BorderBrush_DarkColor}"
```

Der Unterschied liegt in der Auswertung der Zuweisung zur Laufzeit. Die statische Ressource wird beim Verarbeiten der `Resources`-Sektion einmalig ausgewertet und fortan als konstanter Wert betrachtet. Die referenzierte Ressource muss deshalb vor der ersten Anwendung deklariert werden.

Im Gegensatz zu dieser statischen Zuweisung erfolgt bei der dynamischen Zuweisung die Auswertung der Eigenschaft bei jedem Zugriff, kann dadurch Veränderungen an der Ressource erkennen und entsprechend darauf reagieren. Zudem kann die Deklaration der referenzierten Ressource auch erst im späteren Verlauf erfolgen.

Sollte der im Beispiel gezeigte `LinearGradientBrush` durch eine Animation verändert werden, muss diese Ressource der Zieleigenschaft dynamisch zugeordnet werden. Ein weiteres Beispiel für eine dynamische Bindung ist die Verwendung von System-Farben, da diese durch den Benutzer zur Laufzeit verändert werden können.

```
Fill="{DynamicResource {x:Static SystemColors.ControlBrushKey}}"
```

## 2.3 Application

Ressourcen, die in der gesamten Applikation zur Verfügung stehen sollen, werden im `Application`-Element deklariert:

```
<Application x:Class="App"
     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     xmlns:Converter="clr-namespace:Converter"
     StartupUri="View\MainView.xaml">
     <Application.Resources>
          <Converter:DateConverter x:Key="dateConverter" />
```

```
    </Application.Resources>
</Application>
```

### 2.4 Window und Page

Soll eine Ressource nur im aktuellen `Window` oder der aktuellen `Page` gültig sein, können diese dem `Window`- oder `Page`-Element angehängt werden.

### 2.5 Controls

Viele WPF-Klassen (Ableitungen von `FrameworkElement` und `FrameworkContentElement`) unterstützen ebenfalls die Deklaration von Ressourcen.

### 2.6 ResourceDictionary

Losgelöst von optischen Klassen können Ressourcen auch in separaten `ResourceDictionary`-Dateien abgelegt werden.

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
   <Color x:Key="BorderBrush_DarkColor">#FF032D27</Color>
   <Color x:Key="BorderBrush_LightColor">#FF319184</Color>
</ResourceDictionary>
```

Diese so deklarierten Ressourcen werden anschliessend an der gewünschten Stelle, z. B. auf Ebene der Applikation, eingebunden:

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="AdditionalResources.xaml"
/>
```

```
            <ResourceDictionary>
                <Converter:DateConverter x:Key="dateConverter" />
            </ResourceDictionary>
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

## 2.7 Themes

Ein Theme kann als Sammlung von vielen Ressourcen betrachtet werden. Durch das Laden eines anderen Themes kann der Applikation eine möglicherweise komplett neue Oberfläche zugewiesen werden (Skins):

```
Application.Current.Resources =
    (ResourceDictionary)Application.LoadComponent(
        new Uri("Themes/MyTheme.xaml", UriKind.Relative));
```

## 2.8 MSBuild Resource-Dateien

Eine andere Art von Ressourcen sind im Projekt vorhandene Dateien wie z. B. Bilder, die mit dem MSBuild-Vorgang `Resource` kompiliert werden.

Im Unterschied zu den in Xaml definierten Ressourcen wird hier über eine spezielle URI auf die Ressource verwiesen:

```
ImageSource="pack://application:,,,/Images/delete.png"
```

Ressourcen aus referenzierten Assemblies können ebenfalls verwendet werden:

```
ImageSource="pack://application:,,,/ReferencedAssembly;component/add.
png"
```

### 2.9 MSBuild Page-Dateien

Elemente wie Window, Page, FlowDocument, UserControl und ResourceDictionary werden normalerweise mit dem MSBuild-Vorgang Page kompiliert und können ebenfalls über die Resource-URI angesprochen werden.

### 2.10 Lose Dateien

Soll die Ressource nicht in die Applikation eingebettet, sondern lose mitgeliefert werden, kann über die folgende URI darauf zugegriffen werden. Die URI beschreibt einen Pfad relativ zur ausführenden Assembly.

```
ImageSource="pack://siteoforigin:,,,/Images/delete.png"
```

# 3. Style

## 3.1 Überblick

Als Style versteht man eine Sammlung von Eigenschaften und Verhaltensweisen, die zur Laufzeit einem Control oder Element explizit oder implizit zugewiesen wird. Diese Anpassung kann sehr weit gehen und das betroffene Objekt bis auf die Grundelemente modifizieren.

Im Gegensatz zu WinForms ist für eine solch tiefgehende Modifikation keine Ableitung des Controls mehr notwendig.

```
<Style TargetType="{x:Type TextBox}">
     <Setter Property="Margin" Value="0" />
     <Setter Property="Height" Value="Auto" />
     <Setter Property="VerticalContentAlignment" Value="Top" />
     <Setter Property="HorizontalAlignment" Value="Stretch" />
     <Setter Property="VerticalAlignment" Value="Center" />
     <Setter Property="Template" Value="{StaticResource
StyledTextBox}" />
</Style>
```

Typischerweise zeichnet sich ein Style durch die Benennung des Ziel-Typs mit Hilfe der Eigenschaft `TargetType` aus. Mit diesem Typ-Verweis wird ein Bezugskontext gebildet, und alle nicht explizit benannten Eigenschaften beziehen sich auf diesen Typ.

Im obigen Beispiel bedeutet dies, dass `Margin`, `Height` und alle anderen Zuweisungen auf die `TextBox`-Properties angewandt werden. In den direkt nachfolgenden Kapiteln und im Kapitel 4 Template wird darauf nochmals eingegangen.

### 3.2 Vererbung

Wird ein Style auf verschiedenen Ebenen deklariert, wird durch den Auswertungsbaum die innerste Zuweisung gewinnen. Soll nun jedoch eine Erweiterung anstelle einer Ersetzung eines Styles erfolgen, kann dies mit einer Vererbung und der Eigenschaft `BasedOn` erreicht werden:

```
<Style x:Key="BaseButtonStyle" TargetType="{x:Type Button}">
     <Setter Property="Foreground" Value="RoyalBlue" />
</Style>
<Style x:Key="DerivedButtonStyle" BasedOn="{StaticResource
BaseButtonStyle}">
     <Setter Property="Button.Background" Value="RoyalBlue" />
</Style>
```

Im obigen Beispiel wurde im Basis-Style der Ziel-Datentyp `Button` explizit angegeben. Die Eigenschaft `Foreground` ist damit eindeutig dem `Button`-Typ zugeordnet. Der vererbte Style besitzt in diesem Beispiel keine Typ-Zuweisung. Deshalb muss der Typ bei der Eigenschaftsbenennung explizit angegeben werden.

### 3.3 Styles als Ressourcen

Styles können als Ressourcen angelegt und damit im entsprechenden Gültigkeitsbereich verwendet werden.

Wie vorgängig bereits erwähnt, haben Ressourcen einen eindeutigen Namen und können darüber angesprochen werden. Wird ein Style als Ressource angelegt, gilt das hier natürlich ebenfalls. Das kapiteleinleitende Beispiel zeigt jedoch, dass es auch ohne ein `x:Key` geht.

Wird bei einem Style nur der `TargetTyp` ohne `x:Key` angegeben, wird dieser zum Standard und im Style-Gültigkeitsbereich automatisch allen Instanzen des Ziel-Typs zugewiesen, sofern diese keine eigenen Style-Zuweisungen besitzen.

```
<Button x:Name="btnDefaultStyle" Content="Standard-Style" />
<Button x:Name="btnExplicitStyle" Content="Eigener Style"
Style="myBtnStyle" />
```

### 3.4 Direkte Deklaration

Ein Style kann auch direkt der `Style`-Eigenschaft eines Elements zugewiesen werden:

```
<Button x:Name="btnExplicitStyle" Content="Mit Style" >
     <Button.Style>
          <Style>
               <Setter Property="Button.Foreground" Value="RoyalBlue"
/>
          </Style>
     </Button.Style>
</Button>
```

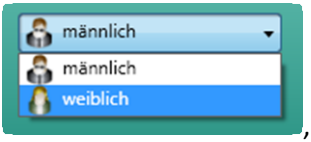
 

# 4. Template

## 4.1 Überblick

Im Gegensatz zu WinForms ist es mit WPF nun sehr einfach, Controls nach eigenen Wünschen anzupassen und sie in Art und Präsentation zu verändern. Templates in Kombination mit Styles, Ressourcen und Datenbindung sind bei richtiger Anwendung ein mächtiges Instrument.

Ganz egal, ob es sich wie im Demo-Projekt um eine ComboBox

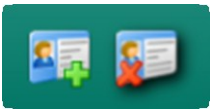



,

um eine ListBox


,

um Buttons



oder Eingabe-Felder


handelt, die Möglichkeiten sind nahezu grenzenlos.

## 4.2 ControlTemplate

Mit dem ControlTemplate bestimmt man das Aussehen eines Controls. Wie bereits von den Styles bekannt, gibt es auch hier eine Typ-Bindung über die Eigenschaft TargetType.

Wird das Template auf ein Control angewendet, ersetzt WPF das Standard-UI mit der im Template spezifizierten Definition.

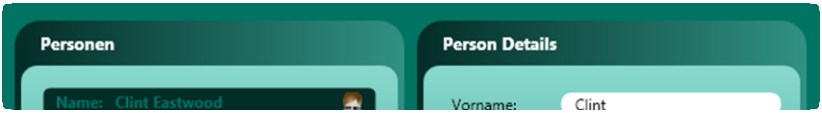
Das nachfolgende Beispiel zeigt das ControlTemplate für die GroupBox-Controls aus dem Demo-Projekt:

```
<ControlTemplate TargetType="{x:Type GroupBox}">
     <Border Padding="5px">
          <Grid>
               <Grid.RowDefinitions>
                    <RowDefinition Height="30px" />
                    <RowDefinition Height="*" />
               </Grid.RowDefinitions>
               <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="20px" />
               <ColumnDefinition Width="*" />
                    <ColumnDefinition Width="20px" />
               </Grid.ColumnDefinitions>
               <Rectangle StrokeThickness="0" Stroke="Transparent"
RadiusX="20"
                RadiusY="20" Fill="{StaticResource BorderBrush}"
Grid.Row="0"
                Grid.Column="0" Grid.RowSpan="2" Grid.ColumnSpan="3"
/>
               <Rectangle StrokeThickness="5" Stroke="{StaticResource
                BorderBrush}" RadiusX="15" RadiusY="15"
Fill="{StaticResource
                FillBrush}" Grid.Row="1" Grid.Column="0"
Grid.ColumnSpan="3" />
               <TextBlock Padding="0 5px 0 0" FontWeight="Bold"
                FontSize="10pt" Text="{Binding RelativeSource=
                {RelativeSource AncestorType={x:Type GroupBox}},
Path=Header}"
                Grid.Row="0" Grid.Column="1"
VerticalAlignment="Center"
```

```
                Foreground="White" />
               <Border Grid.Row="1" Grid.Column="1" Padding="0 20px 0
20px">
                    <ContentPresenter />
               </Border>
          </Grid>
     </Border>
</ControlTemplate>
```

In der Demo-Applikation präsentieren sich die GroupBox-Controls nun so:




Das obige Beispiel zeigt jedoch nicht nur die einfache Verwendung von Xaml-Code zur Definition der UI-Komponente, es enthält auch Datenbindungen und das neue Element ContentPresenter. Mit der Datenbindung beschäftigen wir uns im Kapitel 5 Databinding. Der ContentPresenter kann als eine Art Platzhalter betrachtet werden, in dem beim Rendering der Teil des Controls platziert wird, der bei der jeweiligen Control-Instanz über die Content-Eigenschaft bereitgestellt wurde.

Einem anderen Ansatz folgt der Platzhalter AdornedElementPlaceholder. Hier wird das gesamte ursprüngliche Control beim Rendering eingefügt. Das Template wirkt in diesem Fall wie ein Rahmen um das Control herum. Dieses Verhalten eignet sich sehr gut, um z. B. Validierungen neben einer TextBox anzuzeigen. Ein passendes Beispiel befindet sich im Kapitel 5.9 Validierung.

### 4.3 DataTemplate

Anders als das ControlTemplate richtet sich das DataTemplate an sich wiederholende Objekte in datengebundenen Controls wie z. B. der ListBox und dient der Präsentation der gebundenen Daten.

```
<DataTemplate DataType="{x:Type Model:Actor}">
     <StackPanel Margin="3" HorizontalAlignment="Stretch">
     <StackPanel.Style>
          <Style>
               …
               </Style>
          </StackPanel.Style>
          …
     </StackPanel>
</DataTemplate>
```

Die Anwendung des DataTemplates kann als direkte Zuweisung u. a. aus einer Ressource erfolgen:

```
<ListBox ItemTemplate="{StaticResource MyDataTemplate}" …
```

Soll die Wahl des DataTemplates basierend auf dem Datentyp der jeweils konkret gebundenen Daten erfolgen, kann dies mit der Typen-Bindung über `DataType` erreicht werden:

```
<DataTemplate DataType="{x:Type Model:Actor}"> …
```

Noch flexibler und feiner kann das Template mit einem ItemTemplateSelector bestimmt werden:

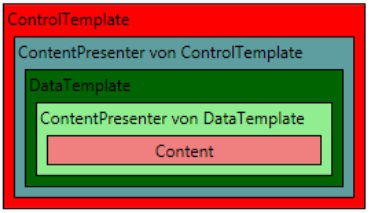
```
<ListBox ItemTemplateSelector="{StaticResource
myDataTemplateSelector}" …
```

Dazu muss eine Ableitung von `DataTemplateSelector` erstellt und in der Methode `SelectTemplate` nach eigener Logik das gewünschte DataTemplate retourniert werden.

Als weitere Anwendungsmöglichkeit kann das DataTemplate auch verwendet werden, um die Darstellung der `Content`-Eigenschaft eines `ContentControls` festzulegen:

```
<Button>
     <Button.ContentTemplate>
     <DataTemplate>
          <StackPanel Orientation="Horizontal">
               <Grid>
                    …
                    </Grid>
                    <ContentPresenter Content="{Binding}"/>
               </StackPanel>
          </DataTemplate>
     </Button.ContentTemplate>
</Button>
```

Im Gegensatz zum `ControlTemplate`, welches das Aussehen des Controls selbst bestimmt, erlaubt das `DataTemplate` in diesem Fall, den Inhalt des Controls zu verändern. Beide Templates können natürlich auch gemeinsam verwendet werden:



### 4.4 ItemsPanelTemplate

Datengebundene Auflistungen, in WPF als `ItemsControl` realisiert, können mit dem `ItemsPanelTemplate` verändert werden. Hierbei beschreibt das Template den Container für die datengebundenen Objekte.

```
<ListBox>
     <ListBox.ItemsPanel>
          <ItemsPanelTemplate>
               <StackPanel Orientation="Horizontal"
VerticalAlignment="Center"
                HorizontalAlignment="Center"/>
          </ItemsPanelTemplate>
     </ListBox.ItemsPanel>
</ListBox>
```

Alternativ kann ein `ItemsPanel` auch direkt in einem `Control-Template` mit der Eigenschaft `IsItemsHost` definiert werden:

```
<ListBox>
     <ListBox.Template>
          <ControlTemplate TargetType="ListBox">
               <ScrollViewer HorizontalScrollBarVisibility="Auto">
                    <StackPanel Orientation="Horizontal"
IsItemsHost="True"/>
               </ScrollViewer>
          </ControlTemplate>
     </ListBox.Template>
</ListBox>
```

Möchte man jedoch das `ItemsPanelTemplate` getrennt vom `ControlTemplate` definieren, bietet sich der Platzhalter `ItemsPresenter` an:

```
<ListBox>
     <ListBox.Template>
          <ControlTemplate TargetType="ListBox">
               <ScrollViewer HorizontalScrollBarVisibility="Auto">
                    <ItemsPresenter/>
               </ScrollViewer>
          </ControlTemplate>
     </ListBox.Template>
</ListBox>
```

Ein komplettes Beispiel könnte so aussehen:


Die starken und unharmonischen Farben sind bewusst gewählt, um die verschiedenen Elemente und deren Eigenschaften besser unterscheiden zu können.

```
<ListBox>
     <ListBox.Template>
          <ControlTemplate>
               <Grid>
                    <Grid.ColumnDefinitions>
                         <ColumnDefinition Width="20px" />
                         <ColumnDefinition Width="*" />
                         <ColumnDefinition Width="20px" />
                    </Grid.ColumnDefinitions>
                    <Grid.RowDefinitions>
                         <RowDefinition Height="20px" />
                         <RowDefinition Height="*" />
                         <RowDefinition Height="20px" />
                    </Grid.RowDefinitions>
                    <Rectangle RadiusX="10px" RadiusY="10px"
Fill="Red"
                     Grid.ColumnSpan="3" Grid.RowSpan="3" />
                    <ItemsPresenter Grid.Row="1" Grid.Column="1" />
               </Grid>
          </ControlTemplate>
     </ListBox.Template>
     <ListBox.ItemsPanel>
          <ItemsPanelTemplate>
               <WrapPanel Background="Blue" />
          </ItemsPanelTemplate>
     </ListBox.ItemsPanel>
     <ListBox.ItemTemplate>
          <DataTemplate>
               <Border CornerRadius="5" BorderThickness="5px"
BorderBrush="Yellow"
                Background="Beige" Padding="5px">
                    <TextBlock Text="{Binding Content}"
Background="Green"
                     Foreground="DarkSalmon" />
               </Border>
          </DataTemplate>
     </ListBox.ItemTemplate>
     <Label Content="ein Text" />
     <Label Content="Text" />
     <Label Content="noch ein Text" />
     <Label Content="etwas mehr Text" />
</ListBox>
```

# 5. Databinding

## 5.1 Überblick

Mit Hilfe von Datenbindungen werden Eigenschaften oder Werte referenziert und an andere Eigenschaften zugewiesen, sodass diese den referenzierten Wert übernehmen, anzeigen oder damit etwas berechnen bzw. bestimmen. Die Bindung muss jedoch nicht einseitig erfolgen und wird auch zum Erfassen und Ändern von Werten eingesetzt, indem die Eigenschaft an ein Eingabe-Control gebunden wird und der Anwender auf diese Weise den Wert manipulieren kann.

## 5.2 ElementName-Binding

Die einfachste Art der Datenbindung ist das Verknüpfen von Eigenschaften.

```
<StackPanel Orientation="Horizontal">
     <TextBox x:Name="myTextBox" Text="Ein Text"
      Width="{Binding ElementName=myScrollbar, Path=Value}" />
     <TextBlock Text="{Binding ElementName=myTextBox, Path=Text}" />
     <TextBlock Text=" Feld-Breite: " />
     <TextBlock Text="{Binding ElementName=myScrollbar, Path=Value}"
/>
</StackPanel>
<ScrollBar x:Name="myScrollbar" Orientation="Horizontal" Minimum="20"
 Maximum="250" Value="60" />
```

Der Feld-Text wird im nebenstehenden `TextBlock` angezeigt und aktualisiert sich bei jeder Änderung automatisch. Die Feld-Breite bestimmt sich aus dem Wert der `ScrollBar` und passt sich laufend an. Der genaue Wert wird im letzten `TextBlock` dargestellt.

Der Wert muss jedoch nicht immer direkt angezeigt werden, sondern kann z. B. in eine Beurteilung von Zuständen einfliessen, was schlussendlich eine (sichtbare) Auswirkung hat. Das nachfolgende Beispiel erstellt eine Abhängigkeit zwischen einem `Button` und einer `CheckBox` und ermittelt die Editierbarkeit des `Buttons` aufgrund des `Check`-Zustands der `CheckBox`.

```
<StackPanel Orientation="Vertical">
     <CheckBox x:Name="cbButtonState" Content="Knopf aktiviert" />
     <Button Content="Knopf"
          IsEnabled="{Binding ElementName=cbButtonState,
Path=IsChecked}" />
</StackPanel>
```

Nach dem gleichen Verfahren kann eine solche Abhängigkeit zwischen Elementen auch über Triggers erstellt werden:

```
<StackPanel Orientation="Vertical">
     <CheckBox x:Name="cbButtonState" Content="Knopf aktiviert" />
     <Button Content="Knopf">
          <Button.Style>
               <Style>
                    <Style.Triggers>
                         <DataTrigger Binding="{Binding
ElementName=cbButtonState,
                          Path=IsChecked}" Value="False">
                              <Setter Property="Button.IsEnabled"
Value="False" />
                         </DataTrigger>
                    </Style.Triggers>
               </Style>
          </Button.Style>
     </Button>
</StackPanel>
```

## 5.3 Source-Binding

Bisher wurde als Datenquelle jeweils ein benanntes Element über die Eigenschaft `ElementName` referenziert. Soll jedoch ein Objekt (Instanz einer Klasse, ein DataSet oder XML-Daten) als Quelle dienen, wird dies mit der Eigenschaft `Source` erreicht.

```
<TabItem.Resources>
     <System:String x:Key="strRes">Ein Text</System:String>
</TabItem.Resources>
…
<TextBlock Text="{Binding Source={StaticResource strRes}}" />
```

Natürlich sind auch komplexere Objekte erlaubt, wie das nachfolgende Beispiel zeigt. Als Datenquelle für eine Liste wird das Resultat einer Methode verwendet:

```
…
<TabItem.Resources>
<ObjectDataProvider ObjectType="{x:Type IO:DirectoryInfo}"
 MethodName="GetFiles" x:Key="FileSource">
     <ObjectDataProvider.ConstructorParameters>
          <System:String>c:\</System:String>
     </ObjectDataProvider.ConstructorParameters>
</ObjectDataProvider>
</TabItem.Resources>
…
<TextBlock Text="Pfad: " />
<TextBox Text="{Binding Source={StaticResource FileSource},
 Path=ConstructorParameters[0], BindsDirectlyToSource=True}" />
<ListBox ItemsSource="{Binding Source={StaticResource FileSource}}"
/>
```

XML-Daten können mit dem `XmlDataProvider` als `BindingSource` dienen:

```
…
<TabItem.Resources>
     <XmlDataProvider x:Key="myData" XPath="DataEntries">
          <x:XData>
               <DataEntries xmlns="">
                    <DataEntry Name="Eintrag 1">
                         <SubEntry Name="SubEintrag 1"/>
                         <SubEntry Name="SubEintrag 2"/>
                    </DataEntry>
               </DataEntries>
          </x:XData>
     </XmlDataProvider>
</TabItem.Resources>
…
```

### 5.4 DataContext-Binding

In WPF werden Eigenschaften an die untergeordneten Elemente vererbt, bzw. sie teilen sich die gleichen Eigenschaften. Der `DataContext` nutzt dieses Verhalten und erlaubt eine Datenbindung auf einem Content-Element wie z. B. einem `StackPanel` und stellt damit den enthaltenen Elementen diese Daten direkt zur Verfügung.

Im nachfolgenden Beispiel wird ein `Brush` als Ressource angelegt und als `DataContext` im `StackPanel` festgelegt. Das `Label` kann nun direkt die `Background`-Eigenschaft binden. Als `Content` wird die `Color`-Eigenschaft vom aktuellen `DataContext`, also die `Brush`-Ressource gebunden.

```
<StackPanel>
     <StackPanel.Resources>
          <SolidColorBrush Color="Orange" x:Key="MyBrush"/>
     </StackPanel.Resources>
     <StackPanel DataContext="{StaticResource MyBrush}">
          <Label Content="{Binding Color}" Background="{Binding}" />
     </StackPanel>
</StackPanel>
```

Die `DataContext`-Eigenschaft kann in der Element-Hierarchie auch neu definiert werden, um z. B. abhängige Listen zu ermöglichen.

Das nachfolgende Beispiel verwendet eine XML-Datenstruktur bestehend aus DataEntry und den ChildNodes SubEntry, abgelegt als Ressource myData. Diese Ressource wird als `DataContext` bestimmt und agiert nun als Datenquelle für die nachfolgenden Bindungen.

Der ersten `ListBox` mit dem Namen parentList werden nun die DataEntry-Objekte über die `ItemsSource` zugewiesen und über ein `DataTemplate` deren Namen angezeigt. Es ist gut erkennbar, dass die Datenbindungen immer nur relativ zu ihren direkt übergeordneten Bindungen erfolgen.

Die zweite `ListBox`, kein Kind-Element von parentList, definiert nun einen neuen `DataContext`, indem eine Datenbindung auf die `Items`-Collection von parentList erstellt wird. Die anzuzeigenden SubEntry-Objekte, ChildNodes vom selektierten DataEntry, werden wieder als relativ bezogene Datenbindung der `ItemsSource`-Eigenschaft zugewiesen.

Damit nun die abhängige Liste sich auf das richtige, das selektierte Hauptobjekt beziehen kann, muss in der parentList die Eigenschaft `IsSynchronizedWithCurrentItem` aktiviert werden.

Natürlich könnte nun eine weitere Liste angehängt und potenziell vorhandene ChildNodes von SubEntry angezeigt werden.

```
<StackPanel Orientation="Vertical" DataContext="{StaticResource
myData}">
     <ListBox x:Name="parentList" ItemsSource="{Binding
XPath=DataEntry}"
      IsSynchronizedWithCurrentItem="True">
          <ListBox.ItemTemplate>
               <DataTemplate>
                    <TextBlock Text="{Binding XPath=@Name}" />
               </DataTemplate>
          </ListBox.ItemTemplate>
     </ListBox>
     <ListBox DataContext="{Binding ElementName=parentList,
Path=Items}"
      ItemsSource="{Binding XPath=SubEntry}" >
          <ListBox.ItemTemplate>
               <DataTemplate>
                    <StackPanel Orientation="Horizontal">
                         <TextBlock Text="{Binding XPath=@Name}" />
                         <TextBlock Text=" (" />
                         <TextBlock Text="{Binding XPath=../@Name}" />
                         <TextBlock Text=")" />
                    </StackPanel>
               </DataTemplate>
          </ListBox.ItemTemplate>
     </ListBox>
</StackPanel>
```

Soll neben der abhängigen Liste auch noch ein Bereich mit Detaildaten des selektierten Objekts angezeigt werden, bietet sich auch hier wieder der `DataContext` an.

```
<StackPanel DataContext="{Binding ElementName=parentList,
Path=SelectedItem}"
 Orientation="Horizontal">
     <TextBlock Text="Name: " />
     <TextBox Text="{Binding XPath=@Name,
UpdateSourceTrigger=PropertyChanged}"
      />
</StackPanel>
```

Anstelle der direkten Bindung, falls z. B. zusätzlicher Code ausgeführt werden soll, kann im Code-Behind-File z. B. das `SelectionChanged`-Ereignis der ersten `ListBox` abonniert und bei der Behandlung das `SelectedItem` als `DataContext` dem `StackPanel` zugewiesen werden. In diesem Fall muss dem `StackPanel` ein Name gegeben werden, damit dieser im Code zur Verfügung steht.

## 5.5 RelativeSource

Nachdem wir nun die Vererbung über `DataContext`, das Referenzieren eines Elements mit `ElementName` und den Bezug von Ressourcen mittels `Source` betrachtet haben, verbleibt noch die vierte und letzte Möglichkeit, eine Datenquelle anzugeben. Die `RelativeSource` erlaubt das Bestimmen der Datenquelle als relativen Bezug zum aktuellen Element.

Eigene Eigenschaften können mit dem `Self`-Mode referenziert werden:

```
<TextBox Background="{Binding RelativeSource={RelativeSource Self},
 Path=Text}" Text="Red" VerticalAlignment="Center" />
```

Eigenschaften aus einem Element im eigenen `VisualTree` lassen sich mit dem `FindAncestor`-Mode referenzieren. Mit dem `AncestorTyp` wird der im `VisualTree` zu suchende Typ angegeben und über `AncestorLevel` kann bestimmt werden, welches Objekt vom entsprechenden Typ verwendet werden soll:

```
<Border BorderBrush="Blue" BorderThickness="2">
     <Border Grid.Row="3" BorderBrush="Red" BorderThickness="4">
          <UniformGrid Columns="2" Rows="1">
               <StackPanel Orientation="Horizontal">
                    <TextBlock Text="Rote Rahmenbreite: " />
                    <TextBlock Text="{Binding
RelativeSource={RelativeSource
                     FindAncestor, AncestorType={x:Type Border}},
                     Path=BorderThickness.Top}" />
               </StackPanel>
               <StackPanel Orientation="Horizontal">
                    <TextBlock Text="Blaue Rahmenbreite: " />
                    <TextBlock Text="{Binding
RelativeSource={RelativeSource
                     FindAncestor, AncestorType={x:Type Border},
AncestorLevel=2},
                     Path=BorderThickness.Top}" />
               </StackPanel>
          </UniformGrid>
     </Border>
</Border>
```

Im `TemplatedParent`-Mode ist es möglich, die Eigenschaften des Elements zu referenzieren, welches das Template anwendet:

```
<Button BorderBrush="Violet" Background="Yellow">
     <Button.Template>
          <ControlTemplate>
               <Ellipse Stroke="{Binding RelativeSource=
                {RelativeSource TemplatedParent}, Path=BorderBrush}"
                Fill="{Binding RelativeSource=
                {RelativeSource TemplatedParent}, Path=Background}" />
          </ControlTemplate>
     </Button.Template>
```

```
</Button>
```

Im Gegensatz zu den vorherigen Möglichkeiten, die einen relativen Bezug zu einem Element hergestellt haben, erlaubt der `PreviousData`-Mode in einem datengebundenen Element den Bezug auf das unmittelbar vorher gebundene Datenobjekt:

```
<DataTemplate>
     <StackPanel Orientation="Horizontal">
          <TextBlock Text="Name: " />
          <TextBlock Text="{Binding XPath=@Name}" />
          <TextBlock Text=" Aktueller Wert: " />
          <TextBlock Text="{Binding XPath=@Wert}" />
          <TextBlock Text=" Vorherigert Wert: " />
          <TextBlock Text="{Binding
           RelativeSource={RelativeSource PreviousData},
XPath=@Wert}" />
     </StackPanel>
</DataTemplate>
```

### 5.6 TemplateBinding

Das `TemplateBinding` hat den gleichen Effekt wie ein `RelativeSource`-Binding im `TemplatedParent`-Mode, jedoch in einer kürzeren Schreibweise:

```
<Button BorderBrush="Violet" Background="Yellow" Height="20px">
     <Button.Template>
          <ControlTemplate>
               <Ellipse Stroke="{TemplateBinding BorderBrush}"
                Fill="{TemplateBinding Background}" />
          </ControlTemplate>
     </Button.Template>
</Button>
```

## 5.7 UpdateSourceTrigger

Mit der `UpdateSourceTrigger`-Eigenschaft von `Binding` kann der Zeitpunkt der Datenbindung gesteuert werden.

Die Eigenschaft kann bei Bedarf definiert werden, anderenfalls gilt der Standardwert `Default` (im Normalfall `PropertyChanged`, für `TextBox`-Controls `LostFocus`).

Der Wert `LostFocus` löst die Datenbindung beim Verlassen des entsprechenden Controls aus. Im Gegensatz dazu aktualisiert der Wert `PropertyChanged` das Binding bei jeder einzelnen Änderung. Und mit dem Wert `Explicit` wird das Aktualisieren des DataBindings komplett dem Entwickler überlassen.

```
BindingExpression beText =
     txtBox.GetBindingExpression(TextBox.TextProperty);
beText.UpdateSource();
```

## 5.8 Mode

Es kann jedoch nicht nur der Zeitpunkt der Datenbindung bestimmt werden, sondern auch die Richtung und das Flussverhalten.

Die Eigenschaft Mode von Binding ermöglicht mit den folgenden Werten unterschiedliche Verhaltensweisen:

Default bzw. TwoWay aktualisiert sowohl die Datenquelle als auch das Ziel bei einer Wertänderung auf der gegenüberliegenden Seite. OneWay beschränkt sich auf die Übertragung von der Quelle zum Ziel und deaktiviert sich, sobald das Ziel anderweitig verändert wird.

OneTime führt die Datenbindung von der Quelle zum Ziel exakt ein Mal aus. Und OneWayToSource kehrt die Richtung um und aktualisiert die Quelle.

## 5.9 Validierung

Die Validierung von datengebundenen Benutzereingaben kann in WPF entweder als frei implementierte `ValidationRule` oder fachbezogen in einem Business-Object und dem `IDataErrorInfo`-Interface erfolgen.

Die eigenständigen `ValidationRule`-Klassen implementieren die `Validate`-Methode und geben über eine `ValidationResult`-Instanz ihr Resultat an das Databinding zurück.

Aktiviert wird die Validierung über die `ValidationRules`-Auflistung des Databindings:

```
<TextBox Style="{StaticResource ErrorStyle}" … >
     <TextBox.Text>
          <Binding … >
               <Binding.ValidationRules>
                    <local:RangeValidationRule LowerValue="50"
UpperValue="100" />
               </Binding.ValidationRules>
          </Binding>
     </TextBox.Text>
</TextBox>
```

Alternativ kann die Validierung auch Fachobjekt bezogen in einem Business-Object erfolgen. In diesem Fall muss die `DependencyObject`-Klasse das `IDataErrorInfo`-Interface implementieren. Die `Error`-Eigenschaft muss hier eine objektweit gültige Fehlermeldung

(oder einen leeren Text) zurückgeben. Die Validierung selbst erfolgt in der `this`-Indexeigenschaft und liefert für die angefragte Eigenschaft eine Text-Fehlermeldung:

```
public class SimpleBo : DependencyObject, IDataErrorInfo
{
     public static readonly DependencyProperty IntValueProperty =
          DependencyProperty.Register("IntValue", typeof(int),
          typeof(SimpleBo), null);

     public int IntValue
     {
          get { return (int)GetValue(IntValueProperty); }
          set { SetValue(IntValueProperty, value); }
     }

     public string this[string property]
     {
          get
          {
               string msg = "";
               switch (property)
               {
                    case "IntValue":
                         if (IntValue <= 0)
                              msg = "Die Zahl muss grösser 0 sein.";
                         if (IntValue > 100)
                              msg = "Die Zahl muss kleiner 100 sein.";
                         break;
                    default:
                         throw new ArgumentException
                              ("Unbekannte Eigenschaft: " + property);
               }
               return msg;
          }
     }

     public string Error
     {
          get { return ""; }
     }
}
```

Aktiviert wird die Validierung, indem die beiden folgenden Eigenschaften auf `true` gesetzt werden:

```
<TextBox Text="{Binding …
     ValidatesOnDataErrors=true, ValidatesOnExceptions=true} />
```

Im Fehlerfall zeichnet WPF einen simplen roten Rahmen um das betroffene Objekt. Über die `Validation.ErrorTemplate`-Eigenschaft kann jedoch ein eigenes Design verwendet werden. Das folgende Beispiel zeichnet einen roten Punkt an der rechten Seite des betroffenen Controls und zeigt die Fehlermeldung als `Tooltip` im Control an:

```
<ControlTemplate x:Key="ErrorTemplate">
     <DockPanel>
          <Ellipse DockPanel.Dock="Right" Margin="2,0"
               ToolTip="Eingabefehler" Width="10" Height="10" >
               <Ellipse.Fill>
                    <LinearGradientBrush>
                         <GradientStop Color="#11FF1111" Offset="0" />
                         <GradientStop Color="#FFFF0000" Offset="1" />
                    </LinearGradientBrush>
               </Ellipse.Fill>
        </Ellipse>
          <AdornedElementPlaceholder />
     </DockPanel>
</ControlTemplate>
<Style x:Key="ErrorStyle">
     <Setter Property="FrameworkElement.Margin" Value="4,4,10,4" />
     <Setter Property="Validation.ErrorTemplate"
          Value="{StaticResource ErrorTemplate}" />
     <Style.Triggers>
          <Trigger Property="Validation.HasError" Value="True">
               <Setter Property="FrameworkElement.ToolTip">
                    <Setter.Value>
                         <Binding
Path="(Validation.Errors)[0].ErrorContent"
                              RelativeSource="{x:Static
RelativeSource.Self}" />
```

```
                    </Setter.Value>
                </Setter>
            </Trigger>
        </Style.Triggers>
</Style>
…
<TextBox Style="{StaticResource ErrorStyle}"
     Text="{Binding … ValidatesOnDataErrors=true,
ValidatesOnExceptions=true}"/>
…
</TextBox>
```

## 5.10 Converter

In vielen Fällen liegen die Daten für die Datenbindung nicht im gewünschten Datentyp oder Format vor, und es ist eine Umwandlung notwendig. Für diesen Zweck kann ein ValueConverter, eine speziell implementierte Klasse, als Unterstützung angegeben werden:

```
<local:ValueConverter x:Key="valueConverter" />
…
<TextBox Text="{Binding ElementName=s, Path=Value,
     Converter={StaticResource valueConverter},
ConverterParameter=0.00}">
</TextBox>
```

Dazu muss vorgängig in den Ressourcen eine Instanz der ValueConverter-Klasse angelegt und anschliessend im gewünschten Binding referenziert werden. Über die optionale Eigenschaft `ConverterParameter` können zusätzliche Informationen wie z. B. ein Zahlenformat übergeben werden.

Die ValueConverter-Klasse implementiert das `IValueConverter`-Interface und ermöglicht eine ein- oder zweiseitige Typumwandlung:

```
 [ValueConversion(typeof(double), typeof(string))]
public class ValueConverter : IValueConverter
{
     public object Convert(object value, Type targetType, object
parameter,
          CultureInfo culture)
     {
          return ((double)value).ToString(parameter.ToString(),
               culture.NumberFormat);
     }

     public object ConvertBack(object value, Type targetType, object
parameter,
          CultureInfo culture)
     {
          double t = 0.0;
          double.TryParse(value.ToString(), NumberStyles.Any,
               culture.NumberFormat, out t);
          return t;
     }
}
```

Wie das Beispiel zeigt, werden über das Attribut `ValueConversion` Ausgangs- und Zieldatentyp bestimmt und anschliessend die Methoden `Convert` und `ConvertBack` (optional) ausprogrammiert.

Über den optionalen Übergabewert `parameter` kann frei verfügt und dieser z. B. für das gewünschte Zahlenformat verwendet werden. Ebenfalls steht es dem Programmierer frei, die mitgelieferte `CultureInfo` anzuwenden.

Reicht eine einzelne Angabe zur Umwandlung nicht aus, können über `MultiBinding` mehrere Werte an einen MultiValueConverter übergeben werden:

```
<local:MultiValueConverter x:Key="multiValueConverter" />
…
<StackPanel Orientation="Horizontal">
     <Slider x:Name="r" Value="100" Minimum="0" Maximum="255" />
     <Slider x:Name="g" Value="150" Minimum="0" Maximum="255" />
     <Slider x:Name="b" Value="200" Minimum="0" Maximum="255" />
</StackPanel>
…
<Border>
     <Border.Background>
          <MultiBinding Converter="{StaticResource
multiValueConverter}">
               <Binding ElementName="r" Path="Value" />
               <Binding ElementName="g" Path="Value" />
               <Binding ElementName="b" Path="Value" />
          </MultiBinding>
     </Border.Background>
</Border>
```

Die zugehörige MultiValueConverter-Klasse implementiert in diesem Fall das `IMultiValueConverter`-Interface:

```
public class MultiValueConverter : IMultiValueConverter
{
     public object Convert(object[] values, Type targetType,
          object parameter, CultureInfo culture)
     {
          var r = (double) values[0];
          var g = (double) values[1];
          var b = (double) values[2];
          var c = Color.FromRgb((byte)r, (byte)g, (byte)b);
          return new SolidColorBrush(c);
     }

     public object[] ConvertBack(object value, Type[] targetTypes,
          object parameter, CultureInfo culture)
     {
          var c = ((SolidColorBrush)value).Color;
          return new object[] {(double)c.R, (double)c.G,
(double)c.B};
     }
}
```

Die Anwendungsmöglichkeiten der ValueConverter sind nahezu unbegrenzt und reichen von simplen Datentypen-Umwandlungen bis hin zu komplexen Berechnungen unter Verwendung von externen Quellen (z. B. Währungsumrechnung mit tagesaktuellem Wechselkurs).

Es muss jedoch nicht immer eine Umwandlung sein. Mit Hilfe der ValueConverter können auch optische Validierungen (z. B. Text rot einfärben, falls ein negativer Wert erfasst wurde) oder Hinweise bzw. Optimierungen erreicht werden.

Anhand einer `ListBox` und mit verschiedenen `ValueConverter`-Klassen zeigt dieses Anwendungsbeispiel, wie ein Steuerelement datenabhängig optisch ansprechend angepasst werden kann.


Die `ListBox` wird mit einer XML-Ressource über die Eigenschaft `DataContext` befüllt und mit einem `ListBoxItem`-Template versehen.

```
<Window.Resources>
    <XmlDataProvider x:Key="data" XPath="Persons">
        <x:XData>
            <Persons xmlns="">
                <Person Id="1" Name="Dagobert Duck"
Salary="1000000"
                    Gender="m" Boss="selbstständig" />
                <Person Id="20" Name="Daisy Duck" Salary="2000"
```

```
                    Gender="f" Boss="unbekannt" />
                <Person Id="99" Name="Donald Duck" Salary="-500"
                    Gender="m" Boss="Dagobert Duck" />
            </Persons>
        </x:XData>
    </XmlDataProvider>
    <ListBoxWithConverter:SalaryConverter x:Key="vcSalary" />
    <ListBoxWithConverter:NumericFormatConverter
x:Key="vcNumericFormat" />
    <ListBoxWithConverter:GenderConverter x:Key="vcGender" />
    <ListBoxWithConverter:ImageConverter x:Key="vcImage" />
</Window.Resources>

<ListBox DataContext="{DynamicResource data}"
    ItemsSource="{Binding XPath=Person}"
Grid.IsSharedSizeScope="True"
    HorizontalContentAlignment="Stretch">
    <ListBox.Resources>
        <Style TargetType="{x:Type ListBoxItem}">
        <Setter Property="Template">
        <Setter.Value>
        <ControlTemplate>
            <Border x:Name="BackgroundRect" Background="DarkKhaki"
                CornerRadius="5" Margin="2px" Padding="5px">
            <Grid x:Name="ContentHost"
HorizontalAlignment="Stretch"
                    TextBlock.Foreground="Black">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="10px" />
                <ColumnDefinition Width="Auto"
SharedSizeGroup="col1" />
                <ColumnDefinition Width="10px" />
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="10px" />
                <ColumnDefinition Width="*"/>
                <ColumnDefinition Width="10px" />
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="10px" />
                <ColumnDefinition Width="Auto"
SharedSizeGroup="col2"/>
                <ColumnDefinition Width="10px" />
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>
            <Border Grid.Row="0" Grid.Column="1" Grid.RowSpan="2"
```

```
                    CornerRadius="20" Padding="5px" Width="40"
Height="40"
                    Background="{Binding XPath=@Gender,
                    Converter={StaticResource vcGender}}"
                    HorizontalAlignment="Right"
VerticalAlignment="Center">
                    <TextBlock Text="{Binding XPath=@Id}"
FontSize="20"
                         FontWeight="Bold"
HorizontalAlignment="Center"
                         VerticalAlignment="Center" />
               </Border>
               <Border Grid.Row="0" Grid.Column="3" Grid.RowSpan="2"
                    CornerRadius="5" Padding="5px" Width="40"
Height="40"
                    Background="White" HorizontalAlignment="Center"
                    VerticalAlignment="Center">
                    <Image Source="{Binding XPath=@Name,
                         Converter={StaticResource vcImage}}" />
               </Border>
               <TextBlock Grid.Row="0" Grid.Column="5"
Grid.RowSpan="2"
                    Text="{Binding XPath=@Name}" FontSize="20"
FontWeight="Bold"
                    VerticalAlignment="Center" />
               <TextBlock Grid.Row="0" Grid.Column="7"
Text="Angestellt bei: " />
               <TextBlock Grid.Row="0" Grid.Column="9"
                    Text="{Binding XPath=@Boss}"
HorizontalAlignment="Right" />
               <TextBlock Grid.Row="1" Grid.Column="7"
Text="Einkommen: " />
               <TextBlock Grid.Row="1" Grid.Column="9"
                    Text="{Binding XPath=@Salary,
                    Converter={StaticResource vcNumericFormat},
                    ConverterParameter='#,##0.00'}"
                    Foreground="{Binding XPath=@Salary,
                    Converter={StaticResource vcSalary}}"
                    HorizontalAlignment="Right" />
               </Grid>
               </Border>
               <ControlTemplate.Triggers>
               <Trigger Property="Selector.IsSelected" Value="True">
                    <Setter TargetName="BackgroundRect"
                         Property="Background" Value="DarkGoldenrod"
/>
                    <Setter TargetName="ContentHost"
```

```
                        Property="TextBlock.Foreground" Value="White"
/>
                </Trigger>
                </ControlTemplate.Triggers>
            </ControlTemplate>
            </Setter.Value>
            </Setter>
            </Style>
        </ListBox.Resources>
        <ListBox.ItemsPanel>
            <ItemsPanelTemplate>
                <UniformGrid Columns="1" HorizontalAlignment="Stretch"
                    VerticalAlignment="Top" />
            </ItemsPanelTemplate>
        </ListBox.ItemsPanel>
</ListBox>
```

Mit Hilfe der Eigenschaft `Grid.IsSharedSizeScope` kann WPF angewiesen werden, ausgewählte Spalten über mehrere Tabellen hinweg gleichförmig auszurichten. Die betreffenden Spalten werden mit der Eigenschaft `SharedSizeGroup` eindeutig benannt und unabhängig vom Inhalt der Tabelle in allen Instanzen mit identischer Breite dargestellt.

Der `GenderConverter` wird mit der Geschlecht-Eigenschaft gebunden und liefert einen `SolidColorBrush`, welcher an die Hintergrundfarbe des `Border`-Objekts zugewiesen wird:

```
public class GenderConverter : IValueConverter
{
     public object Convert(object value, Type targetType,
          object parameter, CultureInfo culture)
     {
          if (value.ToString() == "m")
               return new SolidColorBrush(Colors.PowderBlue);
          else
               return new SolidColorBrush(Colors.Plum);
     }
     …
}
```

Der `ImageConverter` ermittelt aufgrund der Namen-Eigenschaft einen `Image`-Dateinamen:

```
class ImageConverter : IValueConverter
{
     public object Convert(object value, Type targetType,
          object parameter, CultureInfo culture)
     {
          string name = value.ToString();
          return string.Format("{0}.jpg", name.Replace(' ', '_'));
     }
     …
}
```

Mit dem `NumericFormatConverter` wird das Einkommen gemäss dem Übergabewert `parameter` numerisch formatiert:

```
class NumericFormatConverter : IValueConverter
{
     public object Convert(object value, Type targetType,
          object parameter, CultureInfo culture)
     {
          int v = 0;
          int.TryParse(value.ToString(), out v);
          return v.ToString(parameter.ToString());
     }
     …
}
```

Und der `SalaryConverter` bestimmt abschliessend die Textfarbe des Einkommens:

```
public class SalaryConverter : IValueConverter
{
     public object Convert(object value, Type targetType,
          object parameter, CultureInfo culture)
     {
          int salary = 0;
          int.TryParse(value.ToString(), out salary);

          if (salary < 0)
               return new SolidColorBrush(Colors.Red);
          else
               return new SolidColorBrush(Colors.White);
     }
     …
}
```

# 6. Trigger

Mit Hilfe von Triggern können Ressourcen, Styles und Templates verfeinert und so dynamische Veränderungen in Aussehen und Verhalten erlaubt werden, beeinflusst durch das Anwenderverhalten und/oder andere Controls bzw. gebundene Daten.

In den vorangegangenen Beispielen wurden bereits einfache Trigger verwendet, um z. B. auf Benutzerinteraktionen zu reagieren:

```
<Trigger Property="Selector.IsSelected" Value="True">
     <Setter TargetName="BackgroundRect"
          Property="Background" Value="DarkGoldenrod" />
     <Setter TargetName="ContentHost"
          Property="TextBlock.Foreground" Value="White" />
</Trigger>
```

Dieser Trigger, innerhalb eines `ControlTemplates` einer `ListBox` verwendet, überwacht mit Hilfe der `Selector.IsSelected`-Eigenschaft das jeweilige `ListboxItem` und ändert im selektierten Zustand die Hintergrund- und Textfarbe von ausgewählten Objekten.

Aus Platzgründen wird das sehr umfangreiche Thema der Trigger in diesem Booklet jedoch nicht weiter behandelt.

# 7. Glossar

| Begriff | Erklärung |
|---|---|
| AdornedElementPlaceholder | In einem ControlTemplate ein Platzhalter, worin das ursprüngliche Control eingefügt wird. Das Template entspricht damit einem Rahmen um das eigentliche Control herum. |
| BindsDirectlyToSource | Bestimmt, ob das DataBinding relativ zu den gebundenen Daten (false, default) erfolgt oder ob direkt eine Eigenschaft der DataSource gebunden werden soll. |
| ContentPresenter | Innerhalb eines Controls ein Platzhalter für den instanzspezifischen Inhalt. |
| IsSynchronizedWithCurrentItem | Bestimmt, ob ein Selector die Eigenschaft SelectedItem mit der Eigenschaft CurrentItem der ItemsCollection aus der Eigenschaft Items synchronisieren soll. |
| ItemsPresenter | Innerhalb von datengebundenen Controls ein Platzhalter für die gebundenen Datenobjekte. |
| IValueConverter, IMultiValueConverter | Interface für Datenumwandlungsklassen. Wird über die Converter-Eigenschaft in der Binding-Anweisung verknüpft. |
| IValueProvider | Ein xyzAutomationPeer, gecastet auf IValueProvider, erlaubt das direkte Auslesen des zugehörigen WPF-Control-Wertes. |

| | |
|---|---|
| x:FieldModifier | Mit diesem Schlüsselwort kann die Sichtbarkeit von benannten Objekten, falls notwendig, selbst definiert werden. |
| x:Key | Benennt ein Ressourcen-Objekt und kann sowohl ein String als auch ein Objekt sein. |
| x:Name | Benennt ein Objekt eindeutig und erstellt eine aus dem Code heraus ansprechbare Variable. |
| x:Null | Erlaubt die Zuweisung von null an eine Eigenschaft. |
| x:Shared | Bestimmt, ob eine Ressource gemeinsam für alle Bindungen verwendet werden kann (true, default) oder ob die Ressource jedes Mal neu instanziiert werden muss. |
| x:Static | Das Schlüsselwort erlaubt den Zugriff auf statische Variablen und Eigenschaften von Klassen. |

**Unsere Booklets und vieles mehr finden Sie unter**

**www.bbv.ch**

**bbv Software Services AG**

Stark in den Bereichen

- Beratung und Coaching
- Methoden und Technologien
- Software-Entwicklung
- Software-Qualitätssicherung

bbv
Software Services AG