1 Einführung WPF

WPF: Windows Presentation Foundation

1.1 Layout/Grössen

Layout in C# oder XAML geschrieben. XAML ist leichter und kürzer. Als Grösseneinheit wird DIP (Device Independent Pixels) verwendet.

1.2 Hello WPF

Dateien:

- App.xaml: Markup der Startup-Klasse
- App.xaml.cs: Coed-Behind der Startup-Klasse
- MainWindow.xaml: Markup des Hauptfensters
- MainWindow.xaml.cs: Code-Behind des Hauptfensters
- AssemblyInfo.cs: Projektspezifische Meta-Daten

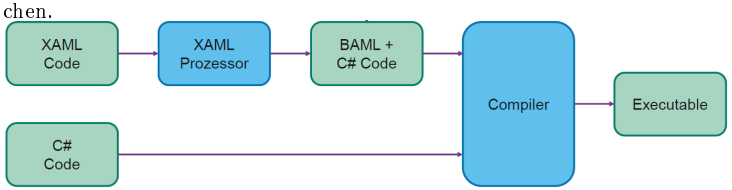
Deployment:

- Framework-Dependent Executable (FDE): .NET Core muss manuell installiert werden. Erzeugt sehr kleines Binary.
- Self-Contained Deployment (SCD): .NET Core in Binary integriert. → Sehr grosses Binary (150MB für hello world)

2 GUI-Programmierung

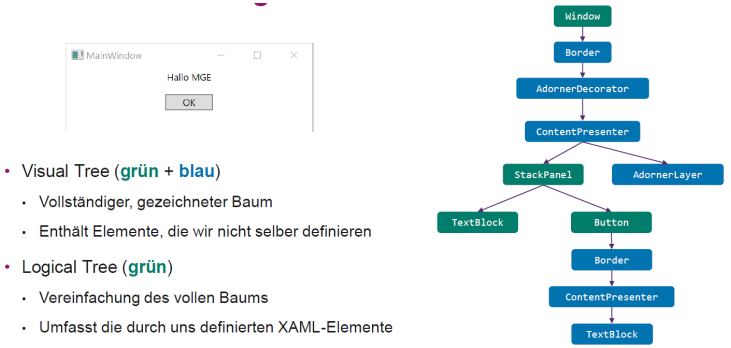
2.1 XAML Allgemein

Beschreibungssprache von Microsoft zur Gestaltung graphischer Oberflächen.



Für Design kann auch C# verwendet werden, XAML ist jedoch leichter, kürzer, lesbarer und hat einen Designer. Microsoft Blend für das Designen.

2.1.1 Visual Tree und Logical Tree



- Visual Tree (grün + blau)
  - Vollständiger, gezeichneter Baum
  - Enthält Elemente, die wir nicht selber definieren
- Logical Tree (grün)
  - Vereinfachung des vollen Baums
  - Umfasst die durch uns definierten XAML-Elemente

2.1.2 Namespaces

Mit xmlns werden XML-Namespaces definiert.
→ Ohne Doppelpunkt: Standard-Namespace (Elemente können ohne Präfix verwendet werden)
→ Mit Doppelpunkt: Nenannter Namespace (Elemente können nur mit Präfix verwendet werden)

Übliche Namespaces in WPF:

- Der Standard-Namespace wird auf die WPF Control Library gesetzt
- x für XAML-spezifische Elemente
- d für Elemente des visuellen Designers
- mc für Elemente der «Markup Kompatibilität»
- local für Elemente aus unserem eigenen Assembly

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/
    xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend
      /2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-
      compatibility/2006"
  xmlns:local="clr-namespace:Vorlesung_09"
  mc:Ignorable="d"
  ... />
```

2.1.3 Named Elements

Elemente können benannt werden. → Ermöglicht Zugriff auf Code-Behind. Attribut führt zu Property in generierter Klasse

```
// XAML:
<TextBlock Name="WpfAttribute" Text="WPF" />
<TextBlock x:Name="XamlAttribute" Text="XAML" />
// Code Behind:
this.WpfAttribute.Text = "...";
this.XamlAttribute.Text = "...";
```

2.1.4 Syntaxen

```
// Attribute Syntax:
<Button Background="Blue"
      Foreground="Red"
      Content="Mein Button" />
// Property Element Syntax:
<Button>
   <Button.Background>
      <SolidColorBrush Color="Blue"/>
   </Button.Background>
   <Button.Foreground>
      <SolidColorBrush Color="Red"/>
   </Button.Foreground>
   <Button.Content>
      Mein Button
   </Button.Content>
</Button>
```

2.1.5 Type Converters

```
// XAML:
<local:LocationControl Center="10, 20" />
// Control:
public class LocationControl : TextBlock {
   public Location Center {
      set => this.Text = $"{value.Lat} / {value.Long}";
   }
}
// Model:
[TypeConverter(typeof(LocationConverter))]
public class Location {
   public double Lat { get; set; }
   public double Long { get; set; }
}
// Type Converter:
public class LocationConverter : TypeConverter {
   public override object ConvertFrom(
     ITypeDescriptorContext context,
     CultureInfo culture,
     object value) {
       //Zur Kürzung des Beispiels auf Checks verzichtet:
       // - Ist value wirklich ein string?
       // - Enthält das Array exakt 2 Elemente?
       // - Sind die strings zu double konvertierbar?
       var valueAsString = (string) value;
       var valueArray = valueAsString.Split(',');
       return new Location {
          Lat = Convert.ToDouble(valueArray[0]),
          Long = Convert.ToDouble(valueArray[1])
       };
   }
}
```

2.1.6 Content Properties

Jedes XAML-Element kann genau eine Eigenschaften als seinen Inhalt definieren. Einige Elemente können, neben reinem Text, auch andere Elemente enthalten.

```
<Button Content="Label" />
<Button>Label</Button>
<Button Width="150" Height="60">
   <StackPanel>
      <TextBlock Text="Gross"
              TextAlignment="Center"
              FontSize="20" />
      <TextBlock Text="Und hier klein"
              FontSize="12"
              Foreground="#888888" />
   </StackPanel>
</Button>
```

2.1.7 Markup Extensions

Erlauben die Erweiterung des XAML-Markup mit zusätzlicher Logik. Die Logik wird in geschweiften Klammern platziert { ... }. Verwendet bei Styling und Data Binding.

```
// XAML:
<TextBlock Text="{local:LocationExtension Lat=10,Long
    =20}" />
// Marup Extension:
public class LocationExtension : MarkupExtension {
   public string Lat { get; set; }
   public string Long { get; set; }
   public override object ProvideValue(IServiceProvider
       s) {
      return this.Lat + " / " + this.Long;
   }
}
```

2.1.8 Attached Properties

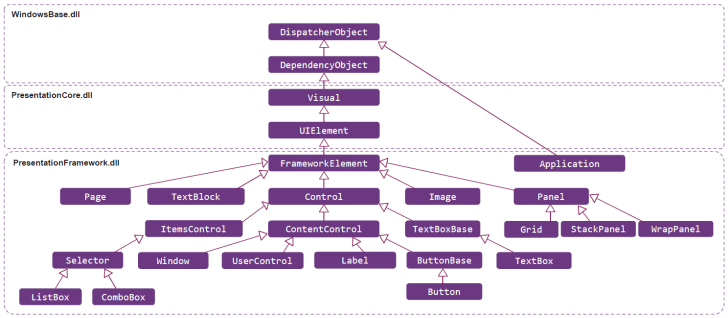
Setzt Eigenschaft auf einem Element, die zu einem anderen Element gehört. Die Eigenschaft wird sozusagen einem anderen Element angehängt.

```
<Grid>
   <Grid.RowDefinitions>
      <RowDefinition Height="30" />
      <RowDefinition Height="20" />
      <RowDefinition Height="10" />
   </Grid.RowDefinitions>

   <TextBlock Grid.Row="0" Name="G" Background="Green"
       />
   <TextBlock Grid.Row="1" Name="R" Background="Red" />
   <TextBlock Grid.Row="2" Name="B" Background="Blue" />
</Grid>
```

2.2 Grundelemente

2.2.1 Klassenhierarchie

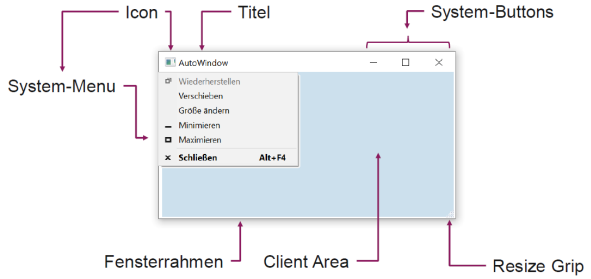


2.2.2 Application

Einstiegspunkt in die Anwendung. Main()-Methode in generiertem Code. Erzeugt Application-Instanz. Definiert via StartupUri die erste View.

```
// XAML:
<Application x:Class="Vorlesung_09.App"
    StartupUri="MainWindow.xaml">
</Application>
// Code Behind:
public partial class App : Application{ }
// Generated Code:
// Nur ein Auszug
public partial class App : System.Windows.Application {
    public void InitializeComponent() {
        this.StartupUri = new System.Uri("MainWindow.xaml"
            , System.UriKind.Relative);
    }
    public static void Main() {
        Vorlesung_09.App app = new Vorlesung_09.App();
        app.InitializeComponent();
        app.Run();
    }
}
```

2.2.3 Window - Sichtbare Elemente



2.2.4 Window - Wichtige Eigenschaften

- Title – Name des Fensters
- Icon – Icon des Fensters
  - Bild mit Build Action "Resource"hinzufügen
  - Verschiedene Dateiformate unterstützt
- ShowInTaskbar – Sichtbarkeit in Taskleiste
- WindowStyle – Aussehen des Fensters
- WindowStartupLocation – Anzeigeposition
- ResizeMode – Modus zur Grössenänderung

2.2.5 UIElement

Wichtigste Basisklasse für visuelle WPF-Elemente.
Definiert grundlegende Elemente, Methoden und Events:
IsEnabled: Reagiert das Element auf Interaktionen?
IsFocused: Ist das Element gerade aktiv?
Visibility: Ist das Element sichtbar? → z.B. Collapsed (Unsichtbar, keinen Platz), Hidden (Unsichtbar, belegt Platz), Visible (Sichtbar), etc.

2.2.6 FrameworkElement

Erweitert UIElement um zusätzliche Funktionalität, unter anderem:

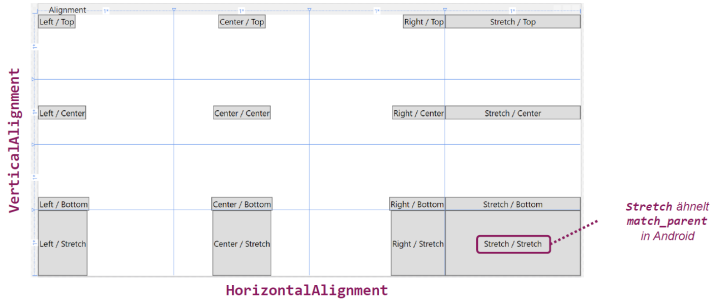
- Name-Property für Zugriff
- Logical Tree
- Layout System
- Visuelles Styling (Woche 10)
- Data Binding (Woche 11)

Grössenangaben:
Width, Height und Margin, Kein Padding. Zusätzlich MinWidth, MaxWidth und MinHeight, MaxHeight.

Dimensionen:
Auto: Automatische Grösse (wrap_content)
px: Device Independent Pixels, 1in == 96px

Ausrichtungen:

- `Alignment` beeinflusst die Ausrichtung **innerhalb des Eltern-Elements**



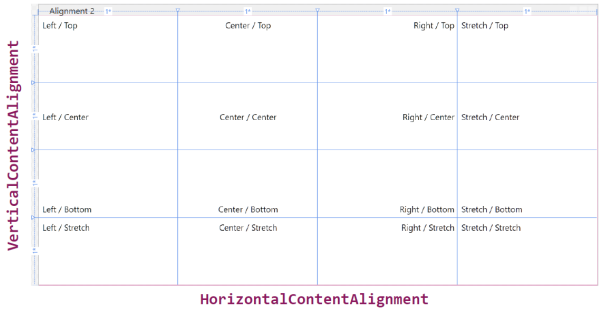
2.2.7 Control

Basis-Klasse für Controls mit Benutzerinteraktion.
Erweitert FrameworkElement um zusätzliche Funktionalität: Gestaltungsmöglichkeiten (Farben, Schriften, Ränder), Ausrichtungen der Kind-Elemente, Control Templates (Woche 10)

Rahmen/Ränder:
Neue Eigenschaften: padding (Innenabstand), BorderThickness (Rahmenstärker), CornerRadius (Radius für abgerundete Ecken)
Grössenangaben für Margin und Padding:
n - Selber Wert für alle Seiten
x,y - X für Horizontal, Y für Vertikal
l,t,r,b - Links, Oben, Rechts, Unten

Ausrichtung:

- `ContentAlignment` beeinflusst die Ausrichtung **der Kind-Elemente**



Farben/Schriften:
Farbgebung mit Brushes ("Pinsel"): Foreground, Background, BorderBrush
Schriftbild: FontFamily, FontSize, FontStretch, FontStyle, FontWeight

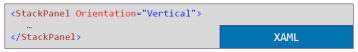
2.3 Layouts

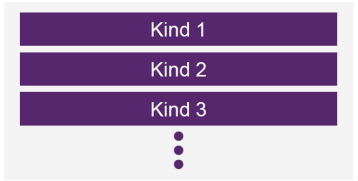
Layouts sind Container für Kind-Elemente. Haben eine Parent-Child Beziehung. Verschachtelung ist möglich.

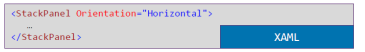
Verfügbare Layouts in WPF:

- StackPanel – Horizontale oder vertikale Auflistung
- WrapPanel – Wie Stack, aber mit Zeilen-/Spaltenumbruch
- DockPanel – Kinder werden an Seiten/im Zentrum "angedockt"
- Grid – Kinder werden den Zellen einer Tabelle zugeordnet

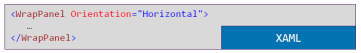
2.3.1 StackPanel

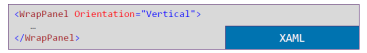


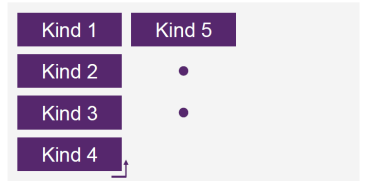




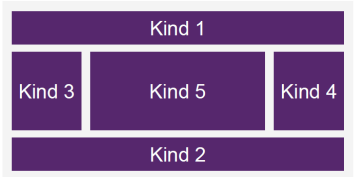
2.3.2 WrapPanel

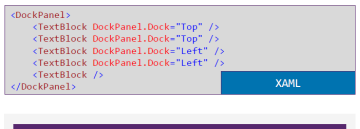


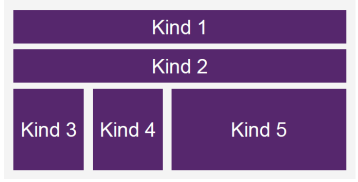


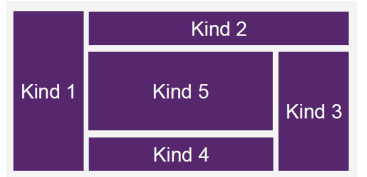


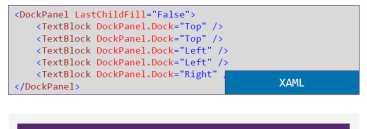
2.3.3 DockPanel

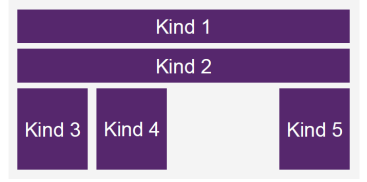




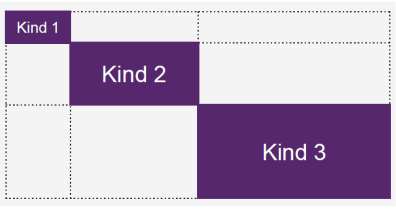








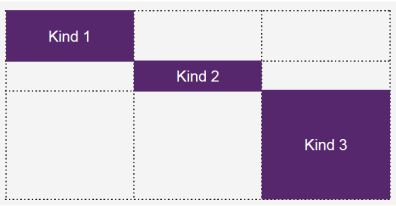
2.3.4 Grid




```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="1*" />
        <RowDefinition Height="2*" />
        <RowDefinition Height="3*" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="1*" />
        <ColumnDefinition Width="2*" />
        <ColumnDefinition Width="3*" />
    </Grid.ColumnDefinitions>

    <TextBlock Grid.Row="0" Grid.Column="0" />
    <TextBlock Grid.Row="1" Grid.Column="1" />
    <TextBlock Grid.Row="2" Grid.Column="2" />
</Grid>
```




```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="50" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <TextBlock Grid.Row="0" Grid.Column="0" />
    <TextBlock Grid.Row="1" Grid.Column="1" />
    <TextBlock Grid.Row="2" Grid.Column="2" />
</Grid>
```

3 GUI-Design

3.1 Controls

Das Aussehen von Controls wird über Attribute beeinflusst.

3.1.1 Image

- Bilddatei zum Projekt hinzufügen
  - Build Action: Resource
  - Integration in Binärdatei des Projekts
- Source für Dateipfad
  - Relativer Pfad beginnend bei XAML-Datei
  - Verwendung von Ordnern möglich
- Stretch für Kontrolle der Skalierung
  - Uniform: Bildverhältnis beibehalten (Standard)
  - Fill: Fläche füllen, Bildverhältnis ignorieren
  - UniformToFill: Fläche füllen, Bildverhältnis beibehalten
  - None: Bild gemäss Originalgrösse darstellen

```
<Image Source="../Bilder/Logo.jpg" Stretch="Uniform" />
```

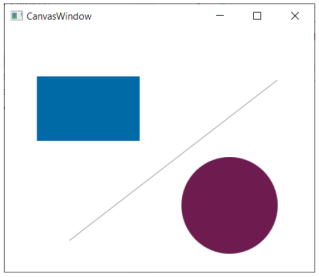
3.1.2 Border

Container für genau ein Element (Controls, oder Layouts). Verwendung zur Gruppierung oder Hervorhebung von Inhalten via Rahmen, Hintergrundfarbe, Runde Ecken, Sichtbarkeit

```
<Border BorderThickness="2"
   BorderBrush="Black"
   Background="#f0f0f0">
   <StackPanel>
      <Button Content="Button 1" Margin="5" />
      <Button Content="Button 2" Margin="5" />
      <Button Content="Button 3" Margin="5" />
   </StackPanel>
</Border>
```

3.1.3 Canvas

2D-Zeichenfläche für einfache geometrische Objekte (Shapes). Absolute Positionierung in X/Y-Raster (Keine Layout-Logik. Kind-Elemente erhalten Attached Porperties)




```
<Canvas>
    <Rectangle Canvas.Left="40"
               Canvas.Top="60"
               Width="128"
               Height="80"
               Fill="#006aa6" />

    <Ellipse Canvas.Left="220"
             Canvas.Top="160"
             Width="120"
             Height="120"
             Fill="#6E1C50" />

    <Path Canvas.Left="80"
          Canvas.Top="64"
          Width="260"
          Height="200"
          Stroke="DarkGray"
          Stretch="Fill"
          Data="M1,0 L0,1"/>
</Canvas>
```

XAML

3.1.4 Window Clipping

Die Form eines Window kann mittels Clipping beliebig verändert werden. Das Window muss dazu jedoch korrekte Einstellungen haben:

```
<Window AllowsTransparency="True"
      WindowStyle="None"
      Width="400"
      Height="200"
      Background="#6E1C50">

   <Window.Clip>
      <RectangleGeometry RadiusX="30"
                         RadiusY="30"
                         Rect="0,0,400,200" />
   </Window.Clip>
   <Grid>
      <Label Content="Clipped Window"
           HorizontalAlignment="Center"
           VerticalAlignment="Center"
           Foreground="White" />
   </Grid>
</Window>
```

3.2 Resources

Beliebige Objekte, die in XAML definiert werden können: Brush, Color, String.
Resourcen besitzten eine eindeutige Identifikation. Zuweisung des XAML-Attributs mit x:key. Dies erlaubt später eine Referenzierung.

3.2.1 Resource Directory

- Container zur Speicherung von Resources
- Zugriff über Schlüssel der Resource (x:Key)
- Teil aller FrameworkElement-Ableitungen (Zugriff über Property Element Syntax)
- Beispiele:
  - Application.Resources
  - Window.Resources
  - Button.Resources
  - Label.Resources

3.2.2 Verwendung von Resources

Ziel: Objekte zentral definieren und n-fach wiederverwenden

```
// XAML
<Window>
  <Window.Resources>
    <SolidColorBrush x:Key="OSTBrush" Color="#6E1C50" />
  </Window.Resources>
  <StackPanel>
    <Label Content="Variante 1" Foreground="White">
      // Property Element Syntax
      <Label.Background>
        <StaticResource ResourceKey="OSTBrush" />
      </Label.Background>
    </Label>
    <Label Content="Variante 2"
          Foreground="White"
          // Attribute Syntax mit Markup Extension
          Background="{StaticResource ResourceKey=OSTBrush
              }" />
    <Label Content="Variante 3"
          Foreground="White"
          // Attribute Syntax mit Markup Extension
          Background="{StaticResource OSTBrush}" />
  </StackPanel>
</Window>

// C#
// via FrameworkElement.FindResource( ... )
var brush = FindResource("OSTBrush") as Brush;
```

3.2.3 Auflösung von Resources

Suchreihenfolge (bricht beim ersten Treffer ab):
1. Aktuelles Element und alle Parent-Elemente
2. In Application.Resources
3. In System-Ressourcen

3.2.4 Statische und dynamische Ressourcen

Statische Resources:

- Einmalige Auswertung der Resource
- Auswertung bei Kompilierung
- Unveränderlich zur Laufzeit
- Extension: {StaticResource Key}

Dynamische Resources:

- Mehrfache Auswertung der Resource
- Auswertung bei Ausführung
- Veränderlich zur Laufzeit
- Extension: {DynamicResource Key}

```
// XAML
<Window>
  <Window.Resources>
    <SolidColorBrush x:Key="OSTBrush" Color="#6E1C50" />
  </Window.Resources>
  <StackPanel>
    <Label Content="OK"
          Foreground="White"
          Background="{DynamicResource OSTBrush}" />
    <Button Content="Update"
           Click="UpdateResource" />
  </StackPanel>
</Window>

// Code Behind
private void UpdateResource(object sender,
     RoutedEventArgs e) {
   Resources["OSTBrush"] = new SolidColorBrush(Colors.
       Blue);
}
```

3.2.5 Beliebige Typen

Ein Resource Dictionary nimmt alle Elemente auf, die in XAML definierbar sind. Aufig werden Basistypen benötigt wie strings, Zahlen, etc.

```
// Stystem.Runtime enthält Basistypen
```

```
<Window xmlns:s="clr-namespace:System;assembly=System.
    Runtime">
 <Window.Resources>
  <s:Double x:Key="MarginVertical">2</s:Double>
  <s:Double x:Key="MarginHorizontal">5</s:Double>
  <Thickness x:Key="Margin"
     Top="{StaticResource MarginVertical}"
     Bottom="{StaticResource MarginVertical}"
     Left="{StaticResource MarginHorizontal}"
     Right="{StaticResource MarginHorizontal}" />
 </Window.Resources>
</Window>
```

3.2.6 Zugriff auf CLR-Werte

Gelegentlich ist es nötig, auf statische Werte der CLR zuzugreifen. Zugriff via Markup Extension: x:Static. Keine WPF-Resources. Werte definiert in normalen Klassen.

```
// C#
public static class MyRes {
   public static SolidColorBrush OSTBrush = new
       SolidColorBrush(Color.FromRgb(110, 28, 80));
}

// XAML
<Label Content="x:Static"
  Background="{x:Static local:MyRes.OSTBrush}"
  Foreground="{x:Static SystemColors.ControlLightBrush}"
  FontFamily="{x:Static SystemFonts.CaptionFontFamily}"
  FontSize="{x:Static SystemFonts.CaptionFontSize}" />
```

3.2.7 Eigenständige Resource Dictionaries

In separater .xaml-Datei mit XML-Root <ResourceDictionary>. In andere Dictionaries als Merged Dictionary integrierbar.

```
// MyDictionary.xaml
<ResourceDictionary>
 <SolidColorBrush x:Key="OSTBrush2" Color="#6E1C50" />
</ResourceDictionary>
// MainWindow.xmal
<Window>
 <Window.Resources>
  // Für Merges zwingend, sonst optional
  <ResourceDictionary>
   <SolidColorBrush x:Key="OSTBrush" Color="#6E1C50" />
   <ResourceDictionary.MergedDictionaries>
    <ResourceDictionary Source="MyDictionary.xaml"/>
   </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
 </Window.Resources>
 <Label Content="Externe Resource"
  Foreground="White"
  Background="{StaticResource OSTBrush2}" />
</Window>
```

3.2.8 Externe Resources

Dictionaries können via Pack URI aus anderen Assemblies eingebunden werden.

3.3 Styles

Resources können mehrfach verwendet werden, müssen aber bei jedem Element referenzieren. Das gibt Duplizierten Code.

3.3.1 Explizite Styles

```
<Window>
 <Window.Resources>
  // Mit oder ohne TargetType
  <Style x:Key="MyButtonStyle" TargetType="Button">
   <Setter Property="Background" Value="Blue" />
   <Setter Property="Foreground" Value="Black" />
   <Setter Property="BorderBrush" Value="Black" />
   <Setter Property="BorderThickness" Value="1" />
  </Style>
 </Window.Resources>
 <StackPanel>
  <Button Style="{StaticResource MyButtonStyle}"
     Content="OK" />
  <Button Style="{StaticResource MyButtonStyle}"
     Content="Cancel" />
 </StackPanel>
</Window>
```

3.3.2 Implizite Styles

Ohne Key wirkt der Style für alle Controls des angegeben Typs.

```
<Window>
 <Window.Resources>
  <Style TargetType="Button"> // Style erhält
      automatisch den key x:key="{x:Type Button}"
   <Setter Property="Background" Value="Blue" />
   <Setter Property="Foreground" Value="Black" />
   <Setter Property="BorderBrush" Value="Black" />
   <Setter Property="BorderThickness" Value="1" />
  </Style>
 </Window.Resources>
 <StackPanel>
  // Keine Style Attribute mehr nötig
  <Button Content="OK" />
  <Button Content="Cancel" />
 </StackPanel>
</Window>
```

3.3.3 Styles erweitern

Styles sind mit Inline-Attributen kombinierbar (eignet sich für einmalige Anpassungen). Styles können auch vererbt werden. → Kann Umfang der Ressourcen reduzieren

```
// Mit Inline Attribute
<Button Style="{StaticResource NormalButton}"
   Background="Red"
   Content="Cancel" />

// Vererbung
<Window.Resources>
 <Style x:Key="NormalButton" TargetType="Button">
  ...
 </Style>
 <Style x:Key="DangerButton"
    BasedOn="{StaticResource NormalButton}"
    TargetType="Button">
  <Setter Property="Background" Value="Red" />
 </Style>
</Window.Resources>
...
<Button Style="{StaticResource DangerButton}"
   Content="Cancel" />
```

3.3.4 Komplexe Werte

Können über setzten des Value-Attributs auf dem Setter Element verwendet werden. Beispiel eines Gradient Hintergrund:

```
<Window.Resources>
 <Style x:Key="BrushButton" TargetType="Button">
  <Setter Property="Background">
   <Setter.Value>
    <LinearGradientBrush StartPoint="0,0" EndPoint="
       0,1">
     <GradientStop Offset="0" Color="Red" />
     <GradientStop Offset="0.5" Color="Yellow" />
     <GradientStop Offset="1" Color="Red" />
    </LinearGradientBrush>
   </Setter.Value>
  </Setter>
 </Style>
</Window.Resources>
<Button Style="{StaticResource BrushButton}" Content="
    Brush" />
```

3.3.5 Trigger

Trigger erlauben Stylings basierend auf dem Zustand eines Elementes. Beliebige Attribute des Elements sind auswertbar. Beispiel: Veränderung des Cursors abhängig vom Button-Label:

```
<Window.Resources>
 <Style x:Key="TriggerButton" TargetType="Button">
  <Style.Triggers>
   <Trigger Property="Content" Value="Link">
    <Setter Property="Cursor" Value="Hand" />
   </Trigger>
   <Trigger Property="Content" Value="Edit">
    <Setter Property="Cursor" Value="Pen" />
   </Trigger>
  </Style.Triggers>
 </Style>
</Window.Resources>
<Button Style="{StaticResource TriggerButton}" Content="
    Link" />
<Button Style="{StaticResource TriggerButton}" Content="
    Edit" />
```

3.3.6 Themes

WPF hat kein Themes-Konzept. Kann aber nachgebaut werden:
Dazu gleiche x:Key styles in mehreren Resource Dictionaries definieren. Laden des gwünschten Dictionary zur Laufzeit und Zuweisung der Styles über DynamicResource.

3.4 Control Templates

- Control Templates beschreiben die visuelle Repräsentation von XAML-Controls (Elemente, die im Visual tree eingefügt werden)
- Der Zugriff auf das aktuelle Template erfolgt über das Attribut Control.Template
- Eigene Control Templates: Option1: Als ControlTemplate-Resource. Option2: Innerhalb eines Styles

ContentPresenter:
Platzhalter für den Content des Elements
TemplateBinding:
Markup Extension für das Binding an Attribute. Nur in Control Templates verwendbar.

3.5 Guidelines

Keine expliziten, bzw. ernstzunehmenden Guidelines für WPF verfügbar. Empfehlung: Guidelines und Libraries nutzen, die sich andernorts bewährt haben. z.B. Material Design in XAML, MahApps.Metro