

## 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

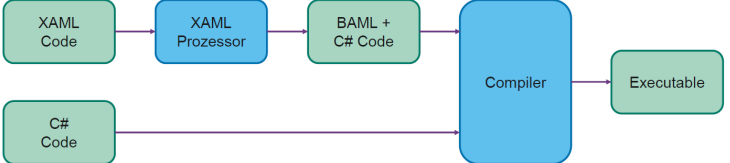
##### Deployment:

- 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

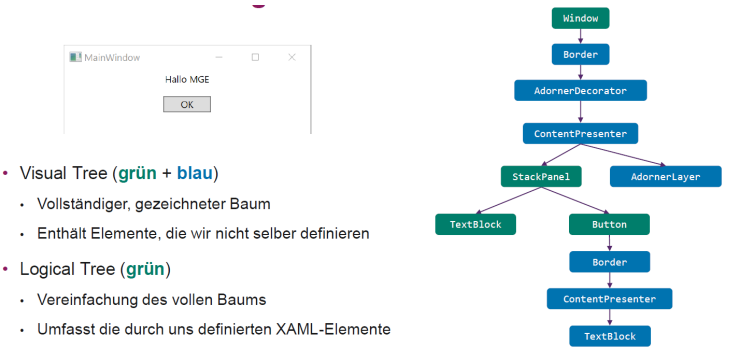
### 2.1 XAML Allgemein

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 LocationControl() {
    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}" />
// Markup 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

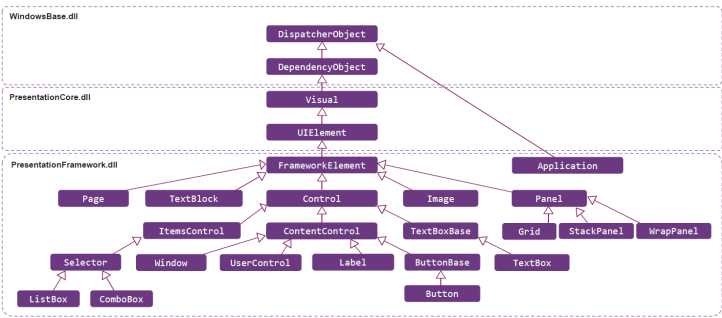
Setzt Eigenschaft auf einem Element, die zu einem anderen Element gehört. Die Eigenschaft wird sozusagen einem anderen Element angehängt.

```
<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

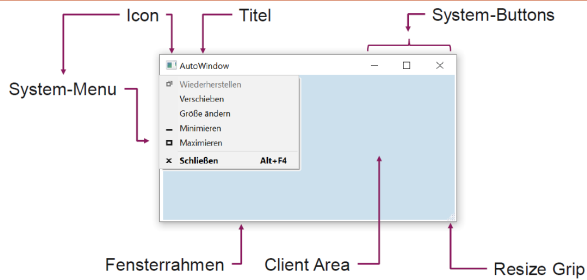


### 2.2.2 Application

**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 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.

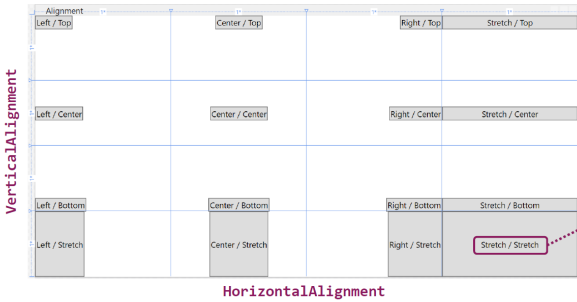
#### Dimensionen:

Auto: Automatische Grösse (wrap\_content)

px: Device Independent Pixels, lin == 96px

#### Ausrichtungen:

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



### Farben/Schriften:

Farbgebung mit Brushes ("Pinsel"): Foreground, Background, Border, Brush

Schriftbild: FontFamily, FontSize, FontStretch, FontStyle, FontWeight

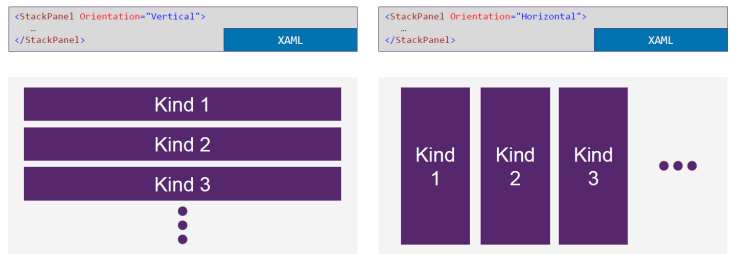
### 2.3 Layouts

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

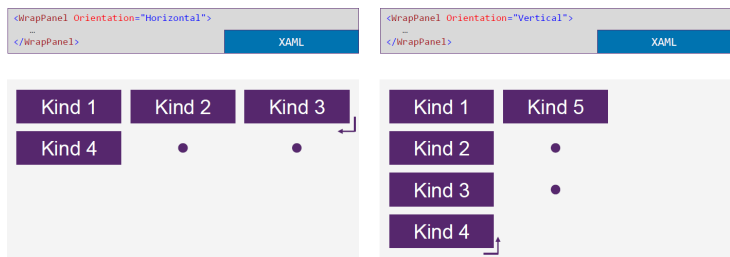
#### Verfügbare Layouts in WPF:

- 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

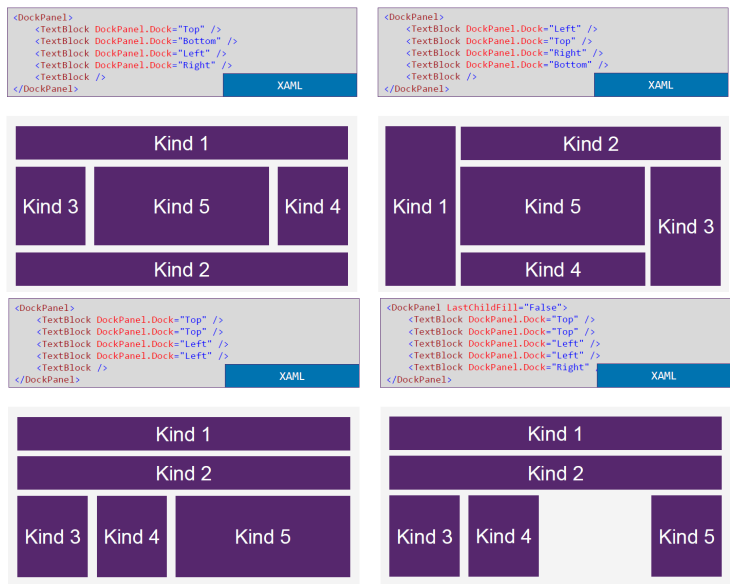
### 2.3.1 StackPanel



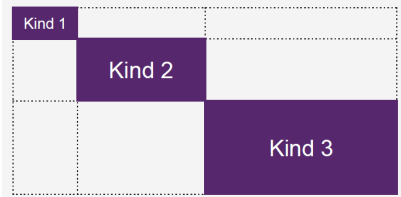
### 2.3.2 WrapPanel



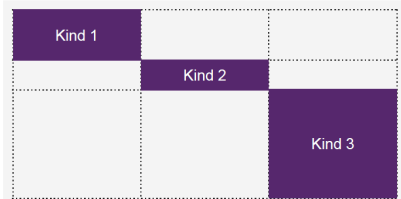
### 2.3.3 DockPanel



### 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" />
```

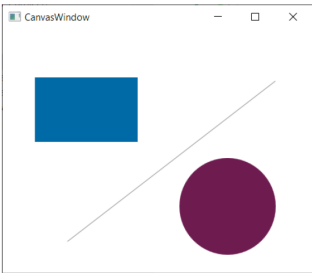
#### 3.1.2 Border

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 Properties)



```
<Canvas>
  <Rectangle Canvas.Left="40"
    Canvas.Top="60"
    Width="128"
    Height="80"
    Fill="#0066aa" />
  <Ellipse Canvas.Left="220"
    Canvas.Top="160"
    Width="120"
    Height="120"
    Fill="#661c50" />
  <Path Canvas.Left="80"
    Canvas.Top="64"
    Width="280"
    Height="200"
    Stroke="DarkGray"
    Stretch="Fill"
    Data="M1,0 L0,1"/>
</Canvas>
```

### 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.

Ressourcen besitzen 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>
  </StackPanel>
</Window>
```

```
</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. Aufg werden Basistypen benötigt wie strings, Zahlen, etc.

// System.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 gewünschten 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





```

        OnPropertyChanged(nameof(FirstName));
    }
}
// Implementation von INotifyPropertyChanged
public event PropertyChangedEventHandler
    PropertyChanged;
// Event Invoker: Erzeugt Argumente und löst Event aus
protected virtual void OnPropertyChanged(string name) {
    var eventArgs = new PropertyChangedEventArgs(name);
    PropertyChanged?.Invoke(this, eventArgs);
}
}

```

#### Mit Basisklasse User.cs:

```

// User.cs
public class User : BindableBase {
    private string _firstName = "Joel";
    public string FirstName {
        get => _firstName;
        set => SetProperty(ref _firstName, value);
    }
}
// BindableBase.cs
public abstract class BindableBase :
    INotifyPropertyChanged {
    // Implementation von INotifyPropertyChanged
    public event PropertyChangedEventHandler
        PropertyChanged;
    // Event Invoker: Erzeugt Argumente und löst Event aus
    protected virtual void OnPropertyChanged(string name) {
        var eventArgs = new PropertyChangedEventArgs(name);
        PropertyChanged?.Invoke(this, eventArgs);
    }
    // CallMemberName wird in Namen des Property
    // umgewandelt
    protected bool SetProperty<T>(ref T field, T value, [
        CallerMemberName] string name = null) {
        if (Equals(field, value)) {
            return false;
        }
        field = value;
        OnPropertyChanged(name);
        return true;
    }
}

```

#### 4.4 Collections

Um eine Collection zu binden muss die Quelle **INotifyCollectionChanged** implementieren und die Ziel-Eigenschaft eine Collection erwarten.

##### 4.4.1 INotifyCollectionChanged

Ist Bestandteil des .NET Framework. Enthält (wie INPC) nur ein Event. Collection-Änderung wird in Event Args beschrieben. **ObservableCollection<T>** implementiert INPC und INCC.

##### 4.4.2 ItemsControl

Wichtigste Basisklasse für WPF-Elemente zur Anzeige von Collections. Definiert folgende Ziel-Eigenschaften:

- Items – Enthält angezeigte Elemente
- ItemsSource – Füllt Inhalt über Data Binding ab
- ItemTemplate – Definiert das Template für die Darstellung eines Items  
→ Bei Verwendung von ItemsSource wird Items zu einer read-only Eigenschaft umgewandelt. Keine Kombination!

```

// User.cs
public class User {
    public string FirstName { get; set; } = "Joel";
    public string LastName { get; set; } = "Schaltegger";
}
// XAML
<Window>
    <ListBox ItemsSource="{Binding}">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <StackPanel>
                    <TextBlock Text="{Binding LastName}" />
                    <TextBlock Text="{Binding FirstName}" />
                </StackPanel>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>
</Window>
// Code Behind
public partial class MainWindow : Window {
    private ObservableCollection<User> users;
    public MainWindow() {
        InitializeComponent();
        // Die Collection müsste natürlich befüllt werden
        ..
        users = new ObservableCollection<User>();
        this.DataContext = users;
    }
}

```

##### 4.4.3 Item Template als Resource

Das Item Template kann als Resource definiert werden. Das Template ist so wiederverwendbar und der XAML Code schlanker. Durch das Attribut **DataType** ist IntelliSense gewährleistet.

```

<Window>
    <Window.Resources>
        <DataTemplate x:Key="UserTemplate"
            DataType="local:User">
            <StackPanel>
                <TextBlock Text="{Binding LastName}" />
                <TextBlock Text="{Binding FirstName}" />
            </StackPanel>
        </DataTemplate>
    </Window.Resources>
    <ListBox ItemsSource="{Binding}"
        ItemTemplate="{StaticResource UserTemplate}" />
</Window>

```

##### 4.4.4 Selector

Erweitert **ItemsControl** um Logik zur Selektion von Elementen. Definiert folgende wichtigen Eigenschaften:

- SelectedIndex – Index des ausgewählten Elements
- SelectedItem – Ausgewähltes Element als Objekt
- SelectedValue – Wert des ausgewählten Elements
- SelectedValuePath – Objektpfad-Syntax zum Wert, der in SelectedValue zurückgeliefert wird

```

// User.cs
public class User {
    public string FirstName { get; set; } = "Joel";
    public string LastName { get; set; } = "Schaltegger";
}

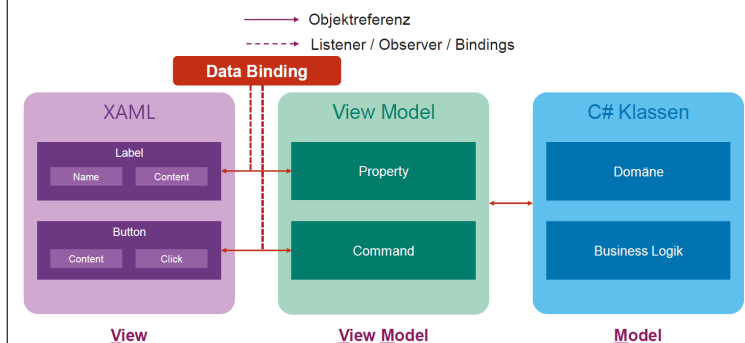
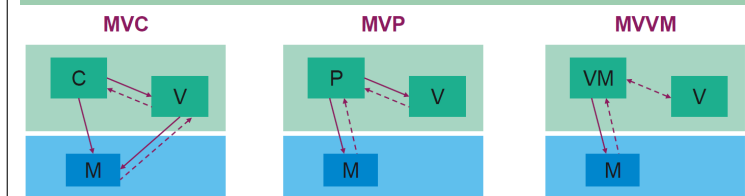
```

```

}
// XAML
<Window>
    <ListBox ItemsSource="{Binding Users}"
        ItemTemplate="{StaticResource UserTemplate}"
        SelectedIndex="{Binding SelectedUserIndex}"
        SelectedItem="{Binding SelectedUser}"
        SelectedValue="{Binding SelectedUserFirstName}"
        SelectedValuePath="FirstName" />
</Window>
// Code Behind
public partial class MainWindow : Window {
    public ObservableCollection<IUser> Users { get; }
    public IUser SelectedUser { get; set; }
    public int SelectedUserIndex { get; set; }
    public string SelectedUserFirstName { get; set; }
    public MainWindow() {
        InitializeComponent();
        // Wie gehabt - hier irgendwie "Users" befüllen
        this.DataContext = this;
    }
}

```

#### 5 MVVM



#### Ziel:

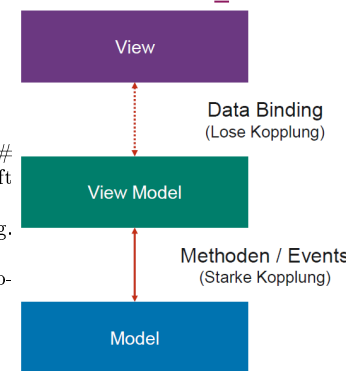
Trennung von Präsentation und Logik

#### Model:

Umfasst Domänen-/Businesslogik. C# Klassen (ggf. mit INPC oder INCC), oft durch Interfaces abstrahiert

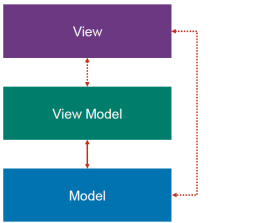
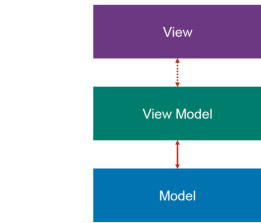
**View:** Kümmert sich um Darstellung, XAML oder Code Behind

**View Model:** Enthält Darstellungslogik. C# Klasse mit INPC



5.1 View Model in WPF

2 Hauptvarianten: Klassisch (links), Durchgriff (rechts):



Beispiel: Klassisch

```
public class User
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

```
public partial class UserView : Window
{
    public UserView()
    {
        InitializeComponent();
        var user = new User();
        DataContext = new UserViewModel(user);
    }
}
```

```
<Window>
<StackPanel>
    <TextBox Text="{Binding FirstName}" />
    <TextBox Text="{Binding LastName}" />
    <TextBlock Text="{Binding FormattedName}" />
</StackPanel>
</Window>
```

Beispiel: Durchgriff

```
public class User : BindableBase
{
    private string _firstName = string.Empty;
    private string _lastName = string.Empty;

    public string FirstName
    {
        get => _firstName;
        set => SetProperty(ref _firstName, value);
    }

    public string LastName
    {
        get => _lastName;
        set => SetProperty(ref _lastName, value);
    }
}
```

```
public partial class UserView : Window
{
    public UserView()
    {
        InitializeComponent();
        var user = new User();
        DataContext = new UserViewModel(user);
    }
}
```

```
public class UserViewModel : BindableBase
{
    private string _first;
    private string _last;

    public UserViewModel(User user)
    {
        _first = user.FirstName;
        _last = user.LastName;
    }

    public string FirstName
    {
        get => _first;
        set
        {
            if (SetProperty(ref _first, value))
            {
                OnPropertyChanged(nameof(FormattedName));
            }
        }
    }

    public string LastName { - } // Analog zu FirstName
    public string FormattedName => $"{_first} {_last}";
}
```

```
<Window>
<StackPanel>
    <TextBox Text="{Binding User.FirstName}" />
    <TextBox Text="{Binding User.LastName}" />
    <TextBlock>
        <TextBlock.Text>
            <MultiBinding StringFormat="{0} {1}">
                <Binding Path="User.FirstName" />
                <Binding Path="User.LastName" />
            </MultiBinding>
        </TextBlock.Text>
    </TextBlock>
</StackPanel>
</Window>
```

```
public class UserViewModel : BindableBase
{
    private User _user;

    public UserViewModel(User user)
    {
        User = user;
    }

    public User User
    {
        get => _user;
        private set => SetProperty(ref _user, value);
    }
}
```

|   | Klassisch | Durchgriff |
|---|-----------|------------|
| MVVM-Implementierung «nach Lehrbuch»              | Ja        | Nein       |
| Saubere Trennung der Bereiche                     | Ja        | Nein       |
| Änderungen am Model haben Einfluss auf View Model | Ja        | Nein¹      |
| Änderungen am Model haben Einfluss auf View       | Nein¹     | Ja         |
| Model frei von technologischen Details            | Ja¹       | Nein¹      |
| Tendenz zu versteckter Darstellungslogik          | Klein     | Gross²     |
| Umfang des Codes                                  | Grösser   | Kleiner    |
| Fleissarbeit («Glue Code»)                        | Mehr      | Weniger    |

5.1.1 AutoMapper - Besseres Klassisch

Hauptnachteil der Variante "Klassisch" ist der zusätzliche Glue Code. **AutoMapper** ist ein **Objekt-Objekt-Mapper**. Abfüllen des View Models aus dem Model und bei Bedarf zurück. Integration als NuGet Paket.

**Beispiel:** View Model aus Model erzeugen

```
var config = new MapperConfiguration(cfg => cfg.
    CreateMap<User, UserViewModel>());
var mapper = config.CreateMapper();
var viewModel = mapper.Map<UserViewModel>(user);
```

5.1.2 Aktionen in View Models

Data Binding erlaubt die Verknüpfung von Eigenschaften, nicht aber von Methoden. Methoden müssen in Objekte verpackt werden.

**ICommand** definiert die Schnittstelle für solche Objekte. View Models stellen **ICommand**-Objekte zur Verfügung.

Command-Eigenschaft von Controls wird an **ICommand**-Objekte gebunden: Button, Checkbox, RadioButton etc.

5.1.3 ICommand

**Execute(Object parameter):** Enthält den Code der auszuführenden Aktion. Bsp: Alter eines Benutzers verringern

**CanExecute(Object parameter):** Prüft, ob die Aktion ausgeführt werden kann. Steuert bei einigen Controls die Verfügbarkeit (IsEnabled). Bsp: true, falls Alter grösser als 0, sonst false

**CanExecuteChanged:** Auszulösen, wenn Bedingung in **CanExecute()** sich ändert. Bsp: Nach jeder Änderung des Alters

**Beispiel ohne Hilfsmittel:**

```
// Command
public class DecreaseAgeCommand : ICommand {
    private readonly UserViewModel _viewModel;
    public DecreaseAgeCommand(UserViewModel viewModel) {
        _viewModel = viewModel;
    }

    public bool CanExecute(object parameter) {
        return _viewModel.Age > 0;
    }

    public void Execute(object parameter) {
        _viewModel.Age--;
        OnCanExecuteChanged();
    }

    public event EventHandler CanExecuteChanged;
    protected virtual void OnCanExecuteChanged() {
        CanExecuteChanged?.Invoke(this, EventArgs.Empty);
    }
}

// View Model:
// View Model zwecks Lesbarkeit gekürzt
public class UserViewModel : BindableBase {
    public UserViewModel(User user) {
        DecreaseAgeCommand = new DecreaseAgeCommand(this);
    }

    public int Age {
        get => ... ;
        set => ... ;
    }

    public ICommand DecreaseAgeCommand { get; }
}
```

```
// XAML
<Window>
    <StackPanel>
        <Button Content="Decrease Age"
            Command="{Binding DecreaseAgeCommand}" />
    </StackPanel>
</Window>
```

**Beispiel mit Hilfsklasse:**

```
// RelayCommand
public sealed class RelayCommand : ICommand {
    private readonly Action _execute;
    private readonly Func<bool> _canExec;
    public RelayCommand(Action execute, Func<bool> canExec) {
        _execute = execute;
    }
}
```

```
_canExec = canExec;
}

public bool CanExecute(object parameter) => _canExec();
public void Execute(object parameter) => _execute();
public event EventHandler CanExecuteChanged;
public void RaiseCanExecuteChanged() {
    CanExecuteChanged?.Invoke(this, EventArgs.Empty);
}
}
```

```
// View Model
public class UserViewModel : BindableBase {
    public UserViewModel(User user) {
        DecreaseAgeCommand = new RelayCommand(
            OnDecreaseAge, CanDecreaseAge);
    }

    public int Age {
        get => ... ;
        private set => ... ;
    }

    public ICommand DecreaseAgeCommand { get; }
    private bool CanDecreaseAge() => Age > 0;
    private void OnDecreaseAge() {
        Age--;
        DecreaseAgeCommand.RaiseCanExecuteChanged();
    }
}
```

5.1.4 Relay Command

**Vorteile des Relay Command:** **ICommand** Interface einmalig implementiert. Universell verwendbar. Command-Code näher beim View Model.

**Nachteile des Relay Command:** Keine wiederverwendbaren Command-Klassen.

5.1.5 Commands mit Parametern

**Commands können Parameter übernehmen:**

**Execute(Object parameter)** und **CanExecute(Object parameter)**. Der Parameter wird in der View gebunden: Attribut **CommandParameter**.

```
<Window>
    <StackPanel>
        <Button Content="Show Details"
            Command="{Binding ShowDetailsCommand}"
            CommandParameter="{Binding SelectedUser}" />
    </StackPanel>
</Window>
```

5.2 Tipps für die Umsetzung

5.2.1 Zuteilen der Logik



5.2.2 Erzeugung von Views und View Models

Das View Model koordiniert den Applikationsfluss.

Beispiel: Öffnen eines Fensters:

- 1. Command auf View Model aufrufen (Binding)
- 2. Command-Logik löst bei Erfolg ein "Navigation Event" für die Event aus
- 3. Der Event Handler in View A erzeugt View B und zeigt diese an (Window.Show)
- 4. View B erzeugt in Konstruktor sein View Model und übernimmt den Applikationsfluss

5.2.3 Hilfsmittel für Model

- Empfohlene O/R-Mapping-Technologie für .NET Anwendungen
- Unterschiedliche Entwicklungsansätze
  - **Model-First:** Erstellung des Modells in visuellem Editor
  - **Database-First:** Erstellung der Datenbank mit SQL
  - **Code-First:** Erstellung des Modells mit attribuierten POCO's

6 Architektur und fortgeschrittene Themen

7 Xamarin und Ausblick