WPF- / .NET-Einführung

.NET Framework: Single Plattform Impl. von .NET für Windows.
.NET Core: Cross Plattform, Open Source mit Fokus auf Desktop.
Mono: Cross Plattform, Open Source Implementierung von .NET auf Mobile
NuGet ist der Paket Manager in .NET. Pakete sind ZIP-Archive (.nupkg).
Pakete werden von öffentlichen oder privaten Hosts heruntergeladen.

WPF

WPF läuft unter .NET Framework und .NET Core
Open Source
Nachfolger von Windows Forms
Trotz .NET Core: nur auf Windows ausführbar
Device Independent Pixels (DIP)
1 UI Pixel = 1/96 Zoll (in WPF)

Hello World

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

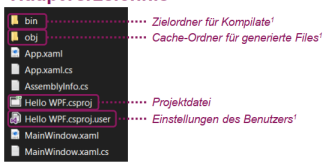
```
<Application x:Class="HelloWPF.App"
    xmlns="http://schemas.microsoft.com/
    winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:HelloWPF"
    StartupUri="MainWindow.xaml">
    <Application.Resources> </Application.Resources>
</Application>
```

```
<Window x:Class="HelloWPF.MainWindow"
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:HelloWPF"
mc:Ignorable="d"
Title="Hello WPF" Height="350" Width="525">
    <Grid>
        <Label x:Name="HelloWorldLabel"
        Content="Hello WPF!"
        HorizontalAlignment="Center"
        VerticalAlignment="Center" />
    </Grid>
</Window>
```

```
namespace HelloWPF {
    using System.Windows;
    public partial class App : Application{ }
}
```

```
namespace HelloWPF {
    using System.Windows;
    public partial class MainWindow : Window {
        public MainWindow() {
            InitializeComponent();
        }
}}
```

Hauptverzeichnis



bin-Verzeichnis

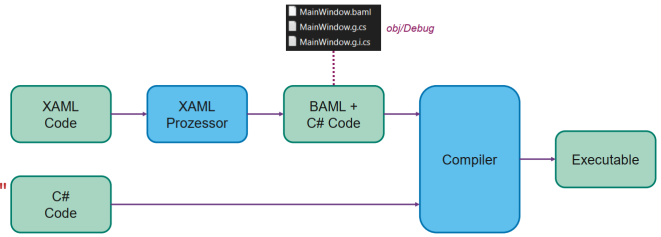
obj-Verzeichnis (Auszug)

Deployment

Framework-Dependent Executable (FDE):
• .NET Core muss manuell installiert werden
• Erzeugt sehr kleines Binary
Self-Contained Deployment (SCD):
• .NET Core wird in Binary integriert
• Erzeugt grosses Binary (150 MB für Hello World)
• Ziel-Architektur muss beim Build gewählt werden

XAML

(Extensible Application Markup Language)
• XML-basiert
• Hierarchisch strukturiert (Baum)
• Trennung von Layout und Code



Vorteile von XAML:
• Oft leichtgewichtiger
• Oft kürzer und besser lesbar
• Grafischer Designer inklusive Vorschau
• Layout durch spezifische Tools erstellbar

Beispiel Code im Codebehind

```
var button = new Button {
    Content = "OK", Height = 20,
    Width = 60, Margin = new Thickness(5)
};
var stackPanel = new StackPanel();
stackPanel.Children.Add(textBlock);
stackPanel.Children.Add(button);
this.Content = stackPanel;
```

Visual 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

Namespaces

Mit xmlns werden XML-Namespaces definiert
• Ohne Doppelpunkt: Standard-Namespace → Elemente können ohne Präfix verwendet werden
• Mit Doppelpunkt: Benannter 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

`mc:Ignorable="d"` : Teilt dem XAML-Parser mit, dass er alle Attribute aus dem Namespace «d» ignorieren soll

Named Elements

Elemente können benannt werden
• Ermöglicht Zugriff im Code Behind
• Attribut führt zu Property in generierter Klasse
Zwei identische Varianten*
• WPF-Attribut: Name
• XAML-Attribut: x:Name

```
    <TextBlock Name="WpfAttribute" Text="WPF" />
    <TextBlock x:Name="XamlAttribute" Text="XAML" />

this.WpfAttribute.Text = "...";
this.XamlAttribute.Text = "...";
```

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>
```

Type Converters

```
<local:LocationControl Center="10, 20" />

public class LocationControl : TextBlock {
    public Location Center {
        set => this.Text = "{value.Lat} {value.Long}";
    }
}

[TypeConverter(typeof(LocationConverter))]
public class Location {
    public double Lat { get; set; }
    public double Long { get; set; }
```

```
}

public class LocationConverter : TypeConverter
{
    public override object ConvertFrom(ITypeDescriptorContext
    context, CultureInfo culture, object value) {
        var valueAsString = (string) value;
        var valueArray = valueAsString.Split(',');
        return new Location {
            Lat = Convert.ToDouble(valueArray[0]),
            Long = Convert.ToDouble(valueArray[1])
        };
    }
}
```

Content Properties

Jedes XAML-Element kann genau eine Eigenschaften als seinen Inhalt definieren
Dieser Inhalt kann in verkürzter Syntax «in das Element hinein» geschrieben werden
Fördert die Lesbarkeit von Parent/ChildBeziehung
Einige Elemente können, neben reinem Text, auch andere Elemente enthalten

Markup Extensions

Erlauben die Erweiterung des XAML-Markup mit zusätzlicher Logik
Häufige Verwendung bei:
Styling / Data Binding

```
<TextBlock Text="{local:LocationExtension Lat=10,Long=20}" />

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;
}}
```

Attached Properties

Setzen einer Eigenschaft auf einem Element,
die zu einem anderen Element gehört Sprich: «Die Eigenschaft wird einem anderen Element angehängt.» Wird meist bei Layouts verwendet
Fördert die Lesbarkeit des XAML. Beispiel: Grid.Row=”0”

GUI Programmierung

Grundidee der Klassen-Hierarchie: Komplexer, aber stellt Funktionien hierarchisch zur Verfügung. Saubere Trennung und alle UI Elemente haben die selbe Basis-Funktionalität

Application

Einstiegspunkt in die Anwendung
• Main()-Methode in generiertem Code
• Erzeugt Application-Instanz
Definiert via StartupUri die erste View
• Starten von weiteren Fenstern
Erlaubt applikationsweite Aktionen
• Gemeinsame Ressourcen
• Beenden der Anwendung
• Reaktion auf Back- und Foregrounding

Window – Wichtige Eigenschaften

Title – Name des Fensters
Icon – Icon des Fensters
• Bild mit Build Action «Resource» hinzufügen
ShowInTaskbar – Sichtbarkeit in Taskleiste
WindowStyle – Aussehen des Fensters
WindowStartupLocation – Anzeigeposition
ResizeMode – Modus zur Grössenänderung

UIElement

Wichtigste Basisklasse für visuelle WPF-Elemente
Definiert grundlegende Eigenschaften, Methoden und Events • **IsEnabled** – Reagiert das Element auf Interaktionen?
• **IsFocused** – Ist das Element gerade aktiv?
• **Visibility** – Ist das Element sichtbar?
− Collapsed: Unsichtbar, belegt keinen Platz im UI
− Hidden: Unsichtbar, belegt aber weiterhin Platz im UI
− Visible: Sichtbar

UIElement is a base class for WPF core level implementations building on Windows Presentation Foundation (WPF) elements and basic presentation characteristics.

FrameworkElement

Erweitert UIElement um zusätzliche Funktionalität, unter anderem
• Name-Property für Zugriff
• Logical Tree
• Layout System
• Visuelles Styling
• Data Binding

Provides a WPF framework-level set of properties, events, and methods for Windows Presentation Foundation (WPF) elements. This class represents the provided WPF frameworklevel implementation that is built on the WPF core-level APIs that are defined by UIElement.

Grundlegende Eigenschaften
• Width, Height und Margin
• Aber kein Padding (folgt auf Control)
Tatsächliche Grösse kann abweichen
• ActualWidth und ActualHeight (Read-Only)
• Kennen wir ähnlich von Android
Zusätzliche Eigenschaften bei Platzknappheit
• MinWidth und MaxWidth
• MinHeight und MaxHeight

Dimensionen für Grössenangaben
• Zahl Device Independent Pixels 1in == 96px
• px Device Independent Pixels 1in == 96px
• in Inch / Zoll (Basis)
• cm Zentimeter 1in == 2.54cm
• pt Points / Punkte 1in == 72pt
• Auto Automatische Grösse (`wrap_content`)
Empfehlung
• Auto und Device Independent Pixels verwenden
• Ziel: Ähnliches Look & Feel auf allen Geräten

Vertical(Content)Alignment: left, center, right, stretch
Horizontal(Content)Alignment: top, center bottom, stretch

Control

Basis-Klasse für Controls mit Benutzerinteraktion
• Nicht alle Controls verwenden Control als Basis
• Layouts erben ebenfalls direkt von FrameworkElement
Erweitert FrameworkElement um zusätzliche Funktionalität
• Gestaltungsmöglichkeiten (Farben, Schriften, Ränder, …)
• Ausrichtungen der Kind-Elemente
• Control Templates

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

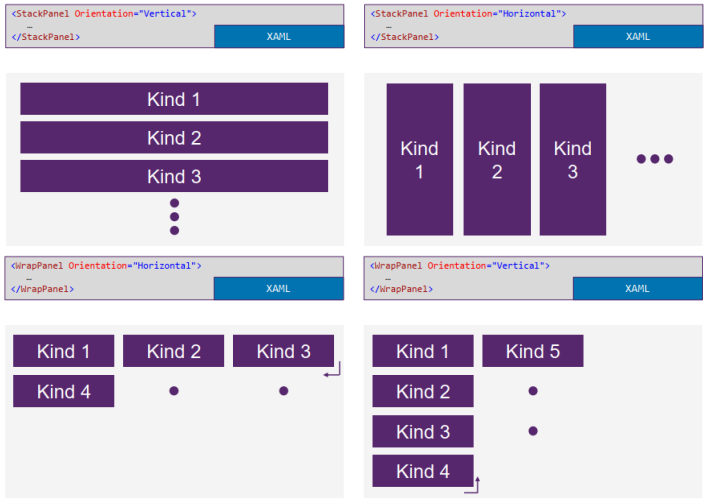
XAML verwendet eine andere Reihenfolge als HTML (t-r-b-l)

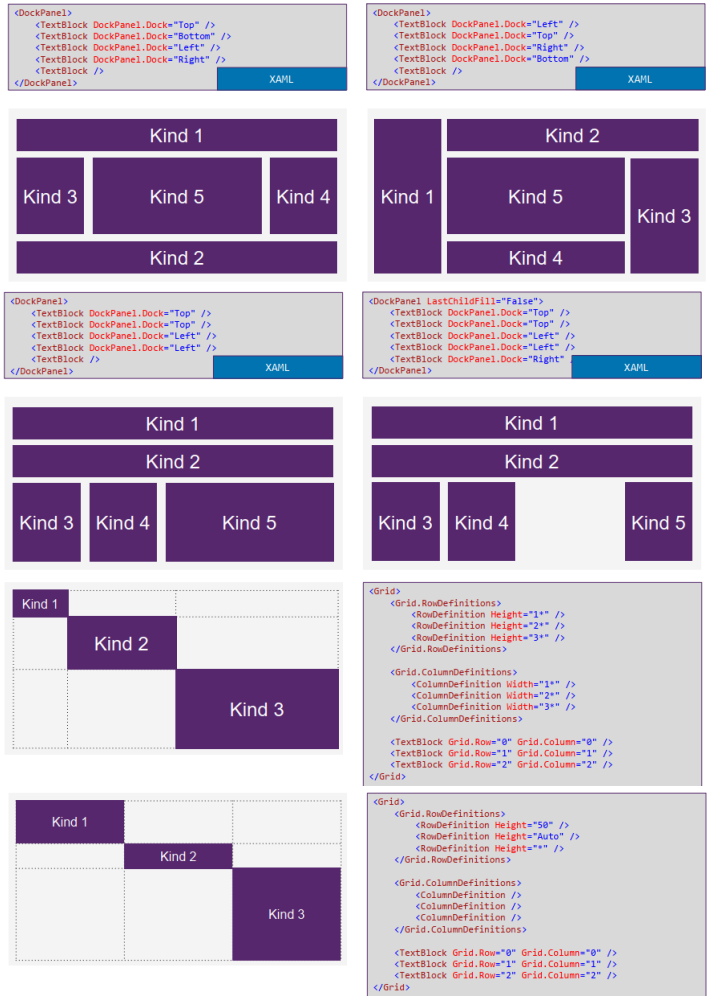
Farbgebung

Farbgebung mit Brushes («Pinsel»)
• Foreground
• Background
• BorderBrush

Layouts

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





Grid

Breiten- und Höhenangaben
• MinWidth und MinHeight
• MaxWidth und MaxHeight
• Kombinierbar mit Auto- und *-Grössen
Zellen und Reihen verbinden
• RowSpan und ColSpan
Grid erlaubt Stapelung von Elementen
• Mehrere Elemente innerhalb einer Zelle
• Unten im Layout = oben beim Zeichnen

Controls

Das Aussehen von Controls wird über Attribute beeinflusst
• Grössenangaben mit verschiedenen Dimensionen
• Ränder, Innen- und Aussenabstände
• Ausrichtungen im Eltern-Element und der Kind-Elemente
• Farben und Schriften

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

Border

Container für genau ein Element
• Controls oder Layouts
Verwendung zur Gruppierung oder Hervorhebung von Inhalten via …
• Rahmen
• Hintergrundfarbe
• Runde Ecken
• Sichtbarkeit

Canvas

2D-Zeichenfläche für einfache geometrische Objekte (Shapes)
Absolute Positionierung in X/Y-Raster, keine dynamischen Grössen
• Keinerlei Layout-Logik
• Kind-Elemente erhalten Attached Properties

```
<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>
```

Shapes verwenden intern leichtgewichtige Model-Klassen für Geometrieberechnungen
Diese können auch selbst verwendet werden, beispielsweise für Window Clipping

Window Clipping

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

```
AllowsTransparency="True"
WindowStyle="None
    ...
<Window.Clip>
    <RectangleGeometry RadiusX="30"
                       RadiusY="30" Rect="0,0,400,200" />
</Window.Clip>
```

Resources

Ziel: Objekte zentral definieren und n-fach wiederverwenden
Beliebige Objekte, die in XAML definiert werden können
• Brush
• Color
• String
Besitzen eine eindeutige Identifikation
• Zuweisung des XAML-Attributs x:Key
• Erlaubt die spätere Referenzierung

Resource Dictionary

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

```
<Label><Label.Background>
    <StaticResource ResourceKey="OSTBrush" />
</Label.Background></Label>

<Label Background="{StaticResource OSTBrush}" />
```

Auflösung von Resources

Suchreihenfolge
1. Aktuelles Element und alle Parent-Elemente (aufwärts entlang dem Logical Tree)
2. In Application.Resources
3. In System-Ressourcen
Die Suche bricht beim ersten Treffer ab
• Reihenfolge im XAML entscheidend
• Vorsicht bei mehrfach verwendeten Schlüsseln

Statische und dynamische Resourcen

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}`

```
<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>

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

Beliebige Typen?

• Ein Resource Dictionary nimmt alle Elemente auf, die in XAML definierbar sind
• Häufig werden Basistypen benötigt (String, Zahlen)
• Diese können in XAML definiert werden

```
<Window xmlns:s="clr-namespace:System;assembly=System.Runtime">
```

x:Static – Zugriff auf CLR-Werte

• Gelegentlich ist es nötig, auf statische Werte der CLR zuzugreifen
• Konstanten im eigenen C#-Code
• Konstanten aus .NET

Zugriff via Markup Extension: x:Static

• Keine WPF-Resources
• Werte definiert in normalen Klassen

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

```
<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}" />
```

Eigenständige Resource Dictionaries

• Separate .xaml-Datei mit XML-Root `<ResourceDictionary>`
• In andere Dictionaries als Merged Dictionaries integrierbar
• Dictionaries in einem Startup-Event-Handler der App-Klasse laden

Externe Resources

Dictionaries können aus anderen Assemblies eingebunden werden
• Verwendung bei umfangreicheren Apps mit mehreren Projekten
• Verwendung bei 3rd Party Libraries wie MahApps.Metro
Mit `siteOfOrigin` werden Resources im Dateisystem referenziert
• Nützlich, wenn Resources erst zur Laufzeit bekannt sind
• Austauschen von Resources ohne Neukompilierung der App
• Pfad relativ zur ausgeführten .exe-Datei

Grenzen von Resources

Mit Resources können mehrfach verwendete Werte zentral verwaltet werden
Aber: wir müssen die Werte weiterhin bei jedem Element referenzieren
Das ist besonders ungünstig, wenn Werte für alle Elemente einer Applikation gelten sollen

Explizite Styles

Variante 1: Ohne Angabe des Typs

```
<Style x:Key="MyButtonStyle">
<Button Style="{StaticResource MyButtonStyle}"
```

Variante 2: Mit Angabe des Typs

```
<Style x:Key="MyButtonStyle" TargetType="Button">
```

Implizite Styles

```
<Style TargetType="Button">
<!--Style erhält automatisch den Key x:Key="{x:Type Button}"-->
```

Styles erweitern

Styles sind mit Inline-Attributen kombinierbar
• Eignet sich für einmalige Anpassungen, z.B. das Hervorheben eines bestimmten Buttons
Styles können auch vererbt werden
• Eine clevere Hierarchie kann den Umfang der Resourcen stark reduzieren

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

Komplexe Werte

Komplexe Werte in Styles sind über die Property Element Syntax möglich
• Setzen des Value-Attributs auf dem Setter-Element
• Ansonsten selbe Syntax wie im normalen XAML

```
<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>
```

Trigger

Trigger erlauben Stylings basierend auf dem Zustand eines Elementes
• Beliebige Attribute des Elements sind auswertbar
• Achtung: einige Elemente besitzen standardmässig bereits Trigger (z.B. 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>
```

Skins / Themes

WPF kennt kein Skin- oder Theme-Konzept
Kann mit den bestehenden Mechanismen aber problemlos nachgebaut werden
• Mehrere Dateien mit gleichen benannten Styles darin
• Laden des gewünschten Resource Dictionaries zur Laufzeit
• Zuweisung der Styles über DynamicResource

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
• Standard-Template pro Control vorhanden
• Eigene Templates möglich
Eigene Control Templates
• Option 1: Als ControlTemplate-Resource
• Option 2: Innerhalb eines Styles

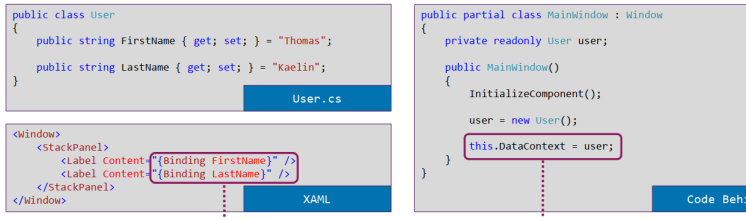
```
<Border BorderBrush="{TemplateBinding BorderBrush}">
    ...
    <ContentPresenter Content="{TemplateBinding Content}"
    Margin="5 0 10 0" VerticalAlignment="Center" />
</Border>
<Button Grid.Row="0" Content="Button mit Template"
Template="{StaticResource ButtonTemplate}"/>
```

Control Templates
`ContentPresenter`
• Platzhalter für den Content des Elementes
• Verwendung optional, aber sehr zu empfehlen
`TemplateBinding`
• Markup Extension für das Binding an Attribute
• Nur in Control Templates verwendbar

Keine WPF Design Guidelines

Empfehlung: Guidelines und Libraries nutzen, die sich andernorts bewährt haben

Databinding



Bindings

Verknüpfen Ziel und Quelle miteinander
• Binding für 1:1 Verknüpfungen
• MultiBinding für 1:n Verknüpfungen
• PriorityBinding für 1:n / 1:1 Verknüpfungen
Varianten zur Erzeugung von Bindings
• Attribute Syntax + Markup Extension
• Property Element Syntax
• C# Code
Ziel-Eigenschaften müssen Dependency Properties sein
• Erweiterte CLR-Properties für WPF

Path
• Name der Quell-Eigenschaft
• Objektpfad-Syntax möglich (z.B. x.y.z)
• Standardparameter der Markup Extension
Mode
• Richtung des Datenflusses
• Standardwert abhängig von Ziel-Eigenschaft
Converter
• Datenumwandlung zwischen Quelle und Ziel
• Umwandlung in beide Richtungen möglich

```
<Binding Path="FirstName" Mode="TwoWay"
    Converter="{StaticResource MyCnv}" />}
```

Mode

• OneTime Einmalige Aktualisierung des Ziels beim Setzen der Quelle
• OneWay Ziel wird bei Änderungen der Quelle aktualisiert
• OneWayToSource Quelle wird bei Änderungen des Zieles aktualisiert
• TwoWay Änderungen werden in beide Richtungen propagiert
• Default Wert abhängig von Ziel-Eigenschaft

Value Converter

Datenumwandlung zwischen Quelle und Ziel
• Beispiel 1: bool zu Visibility
• Beispiel 2: Strings umformatieren
Interface IValueConverter
• Convert(…) Quelle zu Ziel
• ConvertBack(…) Ziel zu Quelle
Erzeugung von Converter
• Option 1: In Resources (StaticResource)
• Option 2: In Code (x:Static)

```
public class ReverseConverter : IValueConverter {
    public object Convert(object value, ...) {
        var stringValue = (string) value;
        var reversedChars = stringValue.Reverse().ToArray();
        var reversedString = new string(reversedChars);
        return reversedString;
    }
    public object ConvertBack(object value, ...) {
        return Convert(value, ...);
}}
```

Weitere Eigenschaften

`Delay` Verzögerung in Millisekunden bei Updates vom Ziel zu Quelle
`StringFormat` Formatangabe für Bindings mit dem Zieltyp string
`FallbackValue` Ergebnis, wenn Binding fehlschlägt (z.B. falscher Pfad)
`TargetNullValue` Ergebnis, wenn Quell-Eigenschaft null liefert
`UpdateSourceTrigger` Zeitpunkt, zu welchem das Quell-Element aktualisiert wird. Mit Explicit selbst laden.
z.B. LostFocus oder PropertyChanged; Standard abhängig vom Ziel

Multi Binding

Verwendung analog zu Binding
• Path, Mode, Converter, etc.
Unterschiede
• Beliebig viele Quell-Eigenschaften
• Nur Property Element Syntax (oder C# Code)
• Converter mit IMultiValueConverter
Je nach Ziel-Eigenschaft ist ein Converter zwingend nötig (z.B. Label.Content)

```
<MultiBinding StringFormat="{}{0} {1} ({2} Jahre)">
    <Binding Path="FirstName" />
    <Binding Path="LastName" />
    <Binding Path="Age" />
</MultiBinding>
```

Data Context

Property der Klasse FrameworkElement
Setzt die Standardquelle für Bindings
Falls undefiniert: Traversierung des Logical Trees nach oben bis zum ersten Treffer
Jeder Path ist relativ zum DataContext
Beliebige Objekte möglich
• C#-Klassen, WPF-Elemente, etc.
• Typischerweise: View Models

überschreiben

Der Data Context lässt sich für einzelne Elemente anpassen
• Option 1: Im Code Behind das Property DataContext für das Element setzen
• Option 2: Source im Binding setzen
Dies ist eher unüblich – meist wird ein Data Context pro Window verwendet

```
public MainWindow() {
    InitializeComponent();
    var user = new User();
    this.LastNameLabel.DataContext = user;
    this.FirstNameLabel.DataContext = user;
}
```

```
<Window.Resources>
    <local:User x:Key="MyUser" />
</Window.Resources>
<Label Content="{Binding Source={StaticResource MyUser},
Path=FirstName}" />
```

Weitere Quellen

Mit `RelativeSource` werden Elemente im Visual Tree referenziert
• Markup Extension `{RelativeSource ...}`
• Suche beginnt beim definierenden Element
• Beispiel: Binding an die Eigenschaft Title des beinhaltenden Window-Objektes

```
<Label Content="{Binding RelativeSource={RelativeSource
FindAncestor, AncestorType=Window}, Path=Title}" />
```

Mit `ElementName` werden Elemente über Namen referenziert
• Namen müssen im gleichen Namensraum vorliegen
• Beispiel: Binding an die Eigenschaft Text des Objektes mit Name MyText

```
<TextBox Name="MyText" Text="Hallo MGE" />
<TextBox Text="{Binding ElementName=MyText, Path=Text}" />
```

Design Time Support

Der XAML Designer kennt den Typ des Objekts im Data Context standardmässig nicht
• Das heisst: keine Autovervollständigung verfügbar (IntelliSense)
Als Abhilfe kann das Attribut d:DataContext beim Window gesetzt werden
• Variante 1: Objekterzeugung in XAML und Markup Extension `{d:DesignInstance ...}`

```
d:DataContext="{
d:DesignInstance Type=local:User,IsDesignTimeCreatable=True}"
```

• Variante 2: Objekterzeugung in C# und Markup Extension `{x:Static …}`

```
d:DataContext="{x:Static local:DesignerData.User}
```

Aktualisierung von Daten

POCOs als Data Context

Als Datenquelle können beliebige Objekte verwendet werden, also auch POCOs*
Unsere Bindings funktionieren – allerdings nur mit Einschränkungen (siehe Beispiel)
Mögliche Lösungen
• Observer Pattern
• Observables bei Data Binding
Die Observer-Variante in .NET heisst INotifyPropertyChanged

```
<Window>
    <Label Content="{Binding Age}" />
    <Button Content="Alter ++" Click="Increment" />
</Window>
```

```
public partial class MainWindow : Window {
    private readonly User user;
    private void Increment(object sender, RoutedEventArgs e) {
        user.Age++;
}}
public class User {
    public string FirstName { get; set; } = "Thomas";
    public string LastName { get; set; } = "Kaelin";
    public int Age { get; set; } = 36;
}
```

INotifyPropertyChanged

Bestandteil des .NET Framework
• Interface mit nur einem Event
• Name des geänderten Property in EventArgs

```
public interface INotifyPropertyChanged {
    event PropertyChangedEventHandler PropertyChanged;
}
public delegate void PropertyChangedEventHandler(
object sender, PropertyChangedEventArgs eventArgs);
public class PropertyChangedEventArgs : EventArgs {
    public PropertyChangedEventArgs(string propertyName) {
        this.PropertyName = propertyName;
    }
    public virtual string PropertyName { get; }
}
```

Ohne Hilfmittel

```
public class User : INotifyPropertyChanged {
private string _firstName = "Thomas";
    public string FirstName {
        get => _firstName;
        set {
            if (_firstName != value) {
                _firstName = value;
                OnPropertyChanged(nameof(FirstName));
    }}}
    public event PropertyChangedEventHandler PropertyChanged;
    protected virtual void OnPropertyChanged(string name) {
        var eventArgs = new PropertyChangedEventArgs(name);
        PropertyChanged?.Invoke(this, eventArgs);
}}
```

Mit Hilfmittel

```
public abstract class BindableBase : INotifyPropertyChanged {
    public event PropertyChangedEventHandler PropertyChanged;
    protected virtual void OnPropertyChanged(string name) {
        var eventArgs = new PropertyChangedEventArgs(name);
        PropertyChanged?.Invoke(this, eventArgs);
    }
    protected bool SetProperty<T>(ref T field, T value,
    // Compilezeit name des Porperties
    [CallerMemberName] string name = null) {
        if (Equals(field, value))
            return false;
        field = value;
        OnPropertyChanged(name);
        return true;
}}
public class User : BindableBase {
    private string _firstName = "Thomas";
    public string FirstName {
        get => _firstName;
        set => SetProperty(ref _firstName, value, nameof(FirstName));
}}
```

Fody

Fody basiert auf IL Weaving
• Anpassung der IL zum Kompilationszeitpunkt
• Fody «webt» aufgrund bestimmter Kriterien dynamisch IL Code ein

PropertyChanged.Fody
• Vereinfacht INPC-Implementierungen
• Wandelt normale Properties automatisch um
• Verfügbar als NuGet-Paket

Anmerkungen

Vorsicht bei zusammengesetzten Eigenschaften (z.B. FullName => FirstName + LastName)
• Event für FullName muss bei jeder Änderung von FirstName oder LastName ausgelöst werden
• Das kann ganz schön kompliziert werden
• Einige Libraries – wie Fody – helfen auch hierbei
Unsere Model-Klassen sollten idealerweise keine technologischen Details enthalten
• Dazu zählt auch die Implementierung von INPC
• Üblicherweise implementiert das View Model INPC – so bleibt das Model «sauber»

Collections

Datenbindung an Collections

Bisher haben wir nur «einfache» Objekte gebunden – was bei Collections?
Quelle muss INotifyCollectionChanged implementieren
Die Ziel-Eigenschaft muss eine Collection erwarten
Solche Eigenschaften haben Elemente, die der Darstellung von Collections dienen

INotifyCollectionChanged

Bestandteil des .NET Framework
• Enthält – wie INPC – ebenfalls nur ein Event
• Collection-Änderung in Event Args beschrieben (Hinzufügen, Löschen, etc.)

`ObservableCollection<T>`
• Implementiert INPC und INCC
• Eigene Implementierung von INCC seltener

ItemsControl

• Wichtigste Basisklasse für WPF-Elemente zur Anzeige von Collections
• Definiert folgende Ziel-Eigenschaften (Auswahl)
• 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!
Represents a control that can be used to present a collection of items

```
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;
}}
public class User {
    public string FirstName { get; set; } = "Thomas";
    public string LastName { get; set; } = "Kaelin";
}

<Window>
    <ListBox ItemsSource="{Binding}">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <StackPanel>
                    <TextBlock Text="{Binding LastName}"/>
                    <TextBlock Text="{Binding FirstName}"/>
                </StackPanel>
</DataTemplate></ListBox.ItemTemplate></ListBox></Window>
```

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.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}" />
```

Selector

Erweitert ItemsControl um Logik zur Selektion von Elementen
Definiert folgende wichtigen Eigenschaften (Auswahl)
• `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

Represents a control that allows a user to select items from among its child elements.

```
<ListBox ItemsSource="{Binding Users}"
        ItemTemplate="{StaticResource UserTemplate}"
        SelectedIndex="{Binding SelectedUserIndex}"
        SelectedItem="{Binding SelectedUser}"
        SelectedValue="{Binding SelectedUserFirstName}"
        SelectedValuePath="FirstName" />

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;
}}
```

MVVM im Überblick

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 die Darstellung
• XAML
• Code Behind
View Model enthält Darstellungslogik
• C# Klasse mit INPC

V - View

Kümmert sich um die Darstellung
• Sollte möglichst «dumm» sein
• Logik und Zustände gehören ins View Model
Bedeutung «dummer Views»
• Faustregel: Die Applikation sollte vollständig über das View Model bedienbar sein (z.B. mit Tests)
• Beispiel 1: Verarbeitung von Selektionen im View Model (z.B. SelectedItem)
• Beispiel 2: Steuerung von Sicht- und Verfügbarkeiten durch das View Model
Prinzipien des Software Engineerings auch für die View anwenden
• Clean Code (DRY, KISS, YAGNI, SOLID, etc.)
• Beispiel 1: Definition von wiederverwendbaren Styles anstelle von Inline-Attributen
• Beispiel 2: Auslagern von Item Templates als Resources zur besseren Lesbarkeit

Bestandteile in WPF: XAML, Code Behind, Resources, Value Converter, Markup Extensions, Eigene Controls

VM – View Model

Enthält komplette Logik der Darstellung
Typische Aufgaben
• Formatierung von Model-Eigenschaften
• Halten von Zuständen
• Validierung von Benutzereingaben
• Delegation von Benutzeraktionen an Model

Bestandteile in WPF
• C#-Klassen mit INPC

View Models in WPF

Beispiel Klassisch

```
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 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 class UserViewModel : BindableBase {
    private User _user;
    public UserViewModel(User user) {
        User = user;
    }
    public IUser User {
        get => _user;
        private set => SetProperty(ref _user, value);
}}
```

```
<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>
```

	Klassisch	Durchgriff
MVVM-Implementierung «nach Lehrbuch»	Ja	Nein
Saubere Trennung der Bereiche	Ja	Nein
Änderungen am Model haben Einfluss auf View Model	Ja	Nein[1]
Änderungen am Model haben Einfluss auf View	Nein[1]	Ja
Model frei von technologischen Details	Ja[1]	Nein[1]
Tendenz zu versteckter Darstellungslogik	Klein	Gross[2]
Umfang des Codes	Grösser	Kleiner
Fleissarbeit («Glue Code»)	Mehr	Weniger

AutoMapper- «Klassisch» verbessert

Hauptnachteil der Variante «Klassisch» ist der zusätzliche Glue Code AutoMapper entschärft das Problem • Objekt-Objekt-Mapper (O/O-Mapping) • 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);
```

Aktionen in View Models

Data Binding erlaubt die Verknüpfung von Eigenschaften, nicht aber von Method
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

ICommand

`Execute(Object parameter)`
• Enthält den Code der auszuführenden Aktion
• Beispiel: Alter eines Benutzers verringern
`CanExecute(Object parameter)`
• Prüft, ob die Aktion ausgeführt werden kann
• Steuert bei einigen Controls die Verfügbarkeit (IsEnabled)
• Beispiel: true, falls Alter grösser als 0, sonst false
`CanExecuteChanged`
• Auszulösen, wenn Bedingung in CanExecute() sich ändert
• Beispiel: Nach jeder Änderung des Alters

Beispiel 1 – Ohne Hilfsmittel

```
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);
}}
public class UserViewModel : BindableBase {
    public UserViewModel(User user) {
        DecreaseAgeCommand = new DecreaseAgeCommand(this);
    }
    public int Age {get => {};set => {};}
    public ICommand DecreaseAgeCommand { get; }
}
```

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

Beispiel 2 – Mit Hilfsklasse

```
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);
}}
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();
}}
```

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

-> Theoretisches Problem: Logik in normalen Klassen strukturieren
Ein «Relay Command» ist in fast allen Frameworks enthalten
• Beispiel: RelayCommand in MVVM Light
• Beispiel: DelegateCommand in Prism.WPF

Commands mit Parametern

Commands können Parameter übernehmen
• `Execute(Object parameter)`
• `CanExecute(Object parameter)`
Der Parameter wird in der View gebunden
• Attribut `CommandParameter`
• Typischerweise Binding auf ein Property
• Alternativen möglich (z.B. statische Werte)
Generisches Relay Command als Hilfsmittel
• Prism.WPF: `DelegateCommand<T>`

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

Tipps

Zuteilung von Logik:

Ist die Logik Teil der Domäne oder wird in der Applikation mehrfach verwendet? -> Model
Ist die Logik unabhängig vom verwendeten UI Framework? -> View Model
Otherwise: View
Erzeugung von Views und View Models
• Wer erstellt die View Models?
• Wer erstellt die Views?
Erzeugung des View Models
• Erzeugen in View
• Dependency Injection über Data Context
• Dependency Injection über Konstruktor

```
var view = new WindowB();
view.Show();
```
Code Behind A

```
public WindowB()
{
    InitializeComponent();
    DataContext = new ViewModel1B();
}
```
Code Behind B

```
var vm = new ViewModel1B();
var view = new WindowB();
view.DataContext = vm;
view.Show();
```
Code Behind A

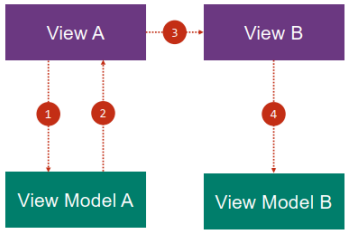
```
public WindowB()
{
    InitializeComponent();
}
```
Code Behind B

```
var vm = new ViewModel1B();
var view = new WindowB(vm);
view.Show();
```
Code Behind A

```
public WindowB(ViewModel1B vm)
{
    InitializeComponent();
    DataContext = vm;
}
```
Code Behind B

Erzeugung von Views und View Models (3)

- 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



Bestimmte Aufgaben können nur in der (WPF-)View erledigt werden
➔ Mehr dazu in Woche 13.

MVVM Frameworks in .NET

Prism Library
• Framework für WPF- und Xamarin-Applikationen
• Ursprünglich von Microsoft entwickelt, jetzt Open Source
MVVM Light
• Leichtgewichtiges Toolkit für .NET UI Technologien
• 1-Mann Open Source Projekt von Laurent Bugnion
MvvmCross
• Framework für .NET UI Technologien (insb. Xamarin)
• Open Source Projekt bei GitHub

Hilfsmittel für das Model – Entity Framework

• 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 attributierten POCO's

Hilfsmittel für das Model – Anderes

SQLite.net
• Einfacher O/R-Mapper auf Basis von SQLite-Datenbanken
• Leichtgewichtiger als Entity Framework
• Arbeitet ebenfalls mit attributierten POCO's
• Spannend insbesondere für Mobile Apps auf Basis von Xamarin
• Integration via NuGet gemäss Anleitung
Json.NET / Newtonsoft.Json
• Standard-Bibliothek zur Verarbeitung von JSON
• Nützlich für Web Services oder die Persistierung von Objekten als JSON-Files
• Integration via NuGet gemäss Anleitung

Architektur

Nutzen von Schichten

Hauptgründe für Schichten
• Fachliche, technische oder organisatorische Grenzen
• Positiver Einfluss auf SW-Qualitätsmerkmale
• Sorgen bei grossen Projekt für Überblick
Anzahl und Namen der Schichten sekundär
• Unterschiede je nach Technologie
• Unterschiede je nach Firma
• Unterschiede je nach Architekt

Horizontale und vertikale Schnitte

Horizontale Schnitte
• Traditioneller Ansatz
• Geeignet für «Technologie Teams»
• Austausch von Technologien einfacher
Vertikale Schnitte
• Modernerer Ansatz
• Geeignet für «Feature Teams»
• Austausch von Technologien schwieriger

Technologische Grenzen

Model und View Model sollen technologie-neutral sein
• Testbarkeit von Model und View Model
• Wiederverwendbarkeit von Model und View Model
• Austauschbarkeit von View

Schön und gut, aber …
• Wie zeigen wir Fehlermeldungen an?
• Wie öffnen wir neue Fenster?
Generell:
• Wie erledigen wir technologie-spezifische Aufgaben?

Dependency Injection

Client kennt Service nur als Interface
Injector erzeugt Service und Client
Injector injiziert Abhängigkeiten via
• Konstruktor
• Methode
• Property

Grundmuster für DI

1. Interface für Verhalten definieren 2. Interface im View Model verwenden 3. Interface im Plattform-Projekt implementieren 4. Service im Plattform-Projekt erzeugen 5. View Model im Plattform-Projekt erzeugen 6. Service in View Model injizieren

Vor- und Nachteile von DI

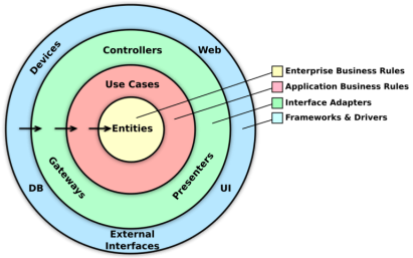
Vorteile
• Geringere Kopplung zwischen Klassen
• Zwang zur Separation of Concerns
• Austauschbarkeit von Services
• Erhöhte Testbarkeit
• Weniger Glue Code im Client
Nachteile
• Zusätzliche Komplexität
• Erschwertes Debugging
• Parameterlisten bei vielen Abhängigkeiten
• Mehr Glue Code beim Injector

DI Container

• Nützlich bei grossen Projekten mit vielen, voneinander abhängigen Services
• Kümmern sich um die Erzeugung der Clients mit ihren benötigten Services
• Konfiguration des Containers nötig («Mapping von Interface zu Service»)
• Lebensdauer konfigurierbar (z.B. Singleton)

«Clean Architecture»

Domäne bildet den Kern unserer Software
Innere Ringe kennen äussere Ringe nicht
Technische «Details» liegen ganz aussen
Basiert auf Dependency Injection
• Interfaces in den inneren Ringen
• Implementierungen in den äusseren Ringen
• Konfiguration der Mappings im äussersten Ring



Testing

Unit Tests profitieren stark von DI
• Test-Projekt als «Plattform-Projekt»
• Implementierung von Fake-Services
• Injektion von Fake-Services
Implementierung von Service
• Variante 1: selber implementieren
• Variante 2: Mocking Library verwenden

Klasse Application

Bisher verwendet als • Einstiegspunkt in unsere Applikation • Container für applikationsweite Ressourcen Statische Eigenschaft Current liefert Singleton-Objekt • Erlaubt Zugriff von überall auf Properties, Methoden, etc. • Nützlich für FindResource() und Shutdown() Lifetime Events • Innerhalb von Application: Methoden überschreiben • Ausserhalb von Application: Event Handler registrieren

Mehrsprachigkeit – Variante «Resources»

• Strings in Resource Dictionaries
• Pro Sprache eine XAML-Datei
• Zugriff im XAML über DynamicResource …
• Zugriff in C# über FindResource()
• Anpassen der Resource Dictionaries bei Sprachänderung im Code Behind

```
<ResourceDictionary>
    <system:String x:Key="Key1">Translation 1</system:String>
</ResourceDictionary>

<Window>
    <Label Content="{DynamicResource Key1}" />
</Window>

public void LoadTranslations(string key) {
    var uri = new Uri("/MyApp;component/Trans.{key}.xaml",
    UriKind.RelativeOrAbsolute);
    var rd = new ResourceDictionary { Source = uri };
    foreach (var rdKey in rd.Keys) {
        Resources[rdKey] = rd[rdKey];
}}
```

Mehrsprachigkeit – Variante «RESX»

Strings in RESX-Dateien
Pro Sprache eine RESX-Datei
Zugriff im XAML über x:Static …
Zugriff in C# über generierte Klasse
Anpassen der «Culture» bei Sprachänderung in beliebigen C# Code

```
<?xml version="1.0" encoding="utf-8"?>
<root>
    <data name="Key1" xml:space="preserve">
        <value>Translation 1</value>
</data></root>

<Window xmlns:t="clr-namespace:MyApp.Translations">
    <Label Content="{x:Static t:Translations.Key1}" />
</Window>

public void LoadTranslations(string key) {
    // Key kann z.B. "de" oder "en-US" sein
    Translations.Culture = new CultureInfo(key);
}
```

Mehrsprachigkeit – Vergleich

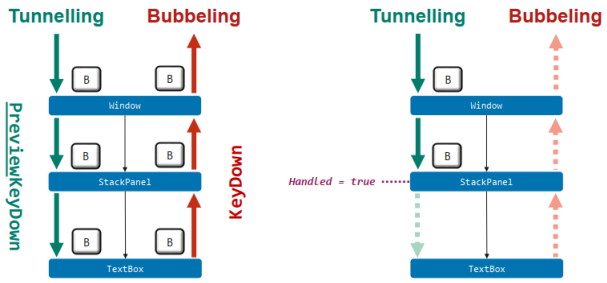
	Resources	RESX
Dateiformat	XAML	RESX
Zugriff in XAML	`{DynamicResource …}`	`{x:Static …}`
Zugriff in C#	`FindResource()`	Generierte Klasse
Visueller Editor für Sprachdateien	Nein	Ja
Aufwand zum Ändern der Sprache	Mittel	Klein
Automatische Aktualisierung der Texte im GUI	Ja	Nein[1]
Zugriff in Projekten ohne WPF	Nein[2]	Ja

Mehrsprachigkeit – Gedanken

Für den Zugriff im C#-Code lohnt sich die Erstellung eines «Translation Service»
Dadurch bleibt der Code unabhängig von einem konkreten Übersetzungsmechanismus
Dies eröffnet die Möglichkeit, Übersetzungen als Properties auf dem View Model zu halten
Umstrittenes Thema – meine persönliche Meinung: es lohnt sich!

Routed Events

UI Ereignisse, auf die reagiert werden kann
• Maustaste wurde gedrückt
• Maus wurde bewegt
• Taste auf Tastatur wurde gedrückt
Bewegen sich in zwei Phasen durch den Visual Tree
• **Tunneling** – Abwärts bis zum fokussierten Element
• **Bubbeling** – Aufwärts vom fokussierten Element
Bewegung kann von jedem Element gestoppt werden
• Setzen von RoutedEventArgs.Handled auf `true`



Routed Events – Beispiel

Möglichst wenig Logik im Code Behind
Bei Routed Events ist ein sparsamer und überlegter Einsatz in Ordnung
• Typischerweise UI-spezifische Logik
• Dient der Verbesserung der User Experience

```
<Window PreviewKeyDown="Window_OnPreviewKeyDown"
    KeyDown="Window_OnKeyDown">
    <StackPanel PreviewKeyDown="StackPanel_OnPreviewKeyDown"
                KeyDown="StackPanel_OnKeyDown">
        <TextBox PreviewKeyDown="TextBox_OnPreviewKeyDown"
                 KeyDown="TextBox_OnKeyDown" />
</StackPanel></Window>

private void Window_OnPreviewKeyDown(
object sender, KeyEventArgs e) {}
private void StackPanel_OnPreviewKeyDown(
object sender, KeyEventArgs e) {}
private void TextBox_OnPreviewKeyDown(
object sender, KeyEventArgs e) {}
```

Background Execution

WPF Threading Model

In jeder WPF-Applikation gibt es mindestens zwei Threads
• UI Thread: Verwaltet UI, empfängt Ereignisse und führt Aktionen aus
• Rendering Thread: Läuft im Hintergrund, zeichnet Controls auf den Screen
Dadurch werden Controls auch dann neu gezeichnet, wenn der UI Thread blockiert ist
• Aber: der UI Thread kann keine neuen Ereignisse mehr verarbeiten

Mechanismen in .NET

NET kennt verschiedene Mechanismen für Backgrounding
• Klasse Task
• Keywords async / await
• Parallel LINQ (PLINQ)
In MGE verwenden wir die Klasse Task
• Einfaches Konzept
• Ähnlichkeit zu Runnables in Java
Moderne Apps verwenden async / await

Klasse Task

Verwendung der Klasse Task
• Statische Methode Task.Run(Action)
• Der UI Thread läuft parallel weiter
Nur der UI Thread darf das UI verändern
• Andernfalls: Exception
• Kennen wir aus Android
• Lösung in WPF: Dispatcher-Klasse

```
Task.Run(() => {});
```

Dispatcher

Erlaubt priorisiertes Abarbeiten von Aufgaben in einem Thread
Delegieren von Aufgaben an den Dispatcher
• Invoke() – Synchron, Aufrufer läuft erst nach Abarbeitung der Aufgabe weiter
• BeginInvoke() – Asynchron, Aufrufer läuft parallel zur Aufgabe weiter

Zugriff auf den Dispatcher

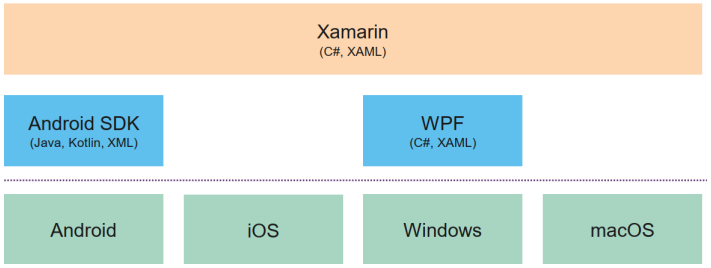
Fast alle WPF-Elemente erben von DispatcherObject
• Eigenschaft Dispatcher
Verwendung in WPF-Projekten
• Code Behind: this.Dispatcher
• Andere Klassen: Application.Current.Dispatcher
Verwendung in Nicht-WPF-Projekten
• Abstrahierung durch Interface nötig
• Implementierung des Interface im WPF-Projekt

Auswirkungen auf View Models

Die gute Nachricht
• Data Binding erledigt das Dispatching zum UI Thread automatisch
• Properties können also ohne Weiteres von einem Background Thread verändert werden
Die schlechte Nachricht
• ICommand.CanExecuteChanged-Events müssen manuell dispatched werden
• Tipp: RelayCommand entsprechend erweitern

```
public sealed class RelayCommand : ICommand {
    public static Action<Action> Dispatch { get; set; }
    public void RaiseCanExecuteChanged() {
        Dispatch(() =>
            CanExecuteChanged?.Invoke(this, EventArgs.Empty));
}}
```

Xamarin



Gemeinsame Codebasis in C# / XAML oder F# / XAML
Zielplattformen: Android, iOS, iPadOS, watchOS, tvOS, macOS
Bei der Kompilierung werden native Apps erzeugt
100%ige Verfügbarkeit der nativen APIs in C#1
Benutzung gewohnter .NET-Tools (Visual Studio, NuGet, etc.)

Shared Code
• Von allen Plattformen geteilt
• Idealerweise möglichst gross
• Interfaces zur Abstraktion von Plattform Details
• .NET Standard Projekt
Platform Code
• Ein Projekt pro Ziel-Plattform
• Idealerweise möglichst klein
• Implementierung der Plattform-Interfaces
• Projekttyp abhängig von Ziel-Plattform

Xamarin.Essentials

Sammlung an Platform Services
• Sensoren Batterie, Kompass
• Schnittstellen Berechtigungen, Telefon
• Utilities Threading, Umrechnungen
Spart viel Zeit und Nerven
• Eine Schnittstelle für alle Plattformen
• Tipp: Trotzdem hinter Interface abstrahieren
Integration via NuGet

Xamarin Traditional

Definition des UI pro Zielplattform
• Verwendung der nativen Konzepte
• Android: XML, Activities, Fragmente, …
Vorteile
• Performance
• Voller Funktionsumfang der Zielplattform
• Portierbarkeit bestehender Apps
Nachteile
• Mehrfache Implementierung des UI
• Viele unterschiedliche Technologien

Xamarin Traditional – MVVM

Dominierende Design Patterns
• MVC in Android und iOS
• MVVM in .NET (XAML)
Wünschenswert wäre MVVM für alles
• Option 1: AndroidX (nur Android)
• Option 2: MvvmCross
MvvmCross
• Data Binding für native UIs (Beispiel Android)

Xamarin.Forms

Definition des UI im Shared Code
• Verwendung von XAML
• Tipp: eigenes Projekt für Xamarin.Forms
Vorteile
• UI muss nur einmalig implementiert werden
• Weniger Technologien
Nachteile
• Einschränkungen bei UI Gestaltung
• Leichte Einbussen bei Performance
• Schwierige Portierbarkeit bestehender Apps

Xamarin.Forms – Renderer

Xamarin.Forms enthält diverse Controls
• Beispiel: Button
Renderer-Klassen erledigen das Mapping von XAML-Controls auf native Controls
• Bestandteil von Xamarin.Forms
• Beispiel: ButtonRenderer für Android
Anpassungsmöglichkeiten
• Styling
• Eigene Renderer
• Eigene XAML-Controls

Xamarin.Forms – Vergleich zu WPF

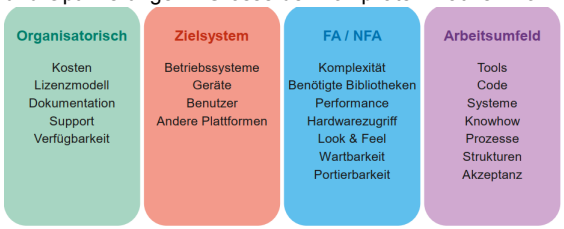
Gemeinsamkeiten
• Aufteilung in XAML und Code Behind
• Application-Klasse
• Resources und Styles
• Markup Extensions
• Data Binding
• Commands
• Value Converter

Unterschiede
• Control Libraries
• Anzahl UI-Projekte
• XAML Dialekt
− MainPage statt MainWindow
− BindingContext statt DataContext
− IsVisible statt Visibility
− Margin mit Komma statt Spaces

Hilfsklassen in Xamarin.Forms
− Service Locator (Dependency Injection)
− Navigation Service

7 Jahre Xamarin im Rückblick

Positive Erlebnisse
• C# und .NET • Visual Studio auf Windows • ReSharper • Einfache Apps mit Xamarin.Forms • Vielfalt an NuGet-Paketen • Verfügbare Dokumentation • Open Source
Negative Erlebnisse
• Updates • Visual Studio auf macOS • Native Libraries integrieren • Komplexe Apps mit Xamarin.Forms • Qualität der NuGet-Pakete • Performance und Optimierungen • Grösse der Kompilate • Native Knowhow unerlässlich



Ausblick

Spannende Technologien – Flutter

Fakten
• Open Source SDK von Google
• Mobile, Desktop und Web
• Programmierung in Dart
• Eigene Rendering-Engine (Skia)
• Open Source bei GitHub
Design-Ziele
• Ausdrucksstarke und flexible Oberflächen
• Native Performance
• Schnelle Entwicklung

Spannende Technologien – MAUI

• MAUI – Multi-Platform App UI
• .NET UI Framework von Microsoft
• Framework für Mobile und Desktop
• Weiterentwicklung von Xamarin.Forms
• Basierend auf .NET 6 (ab Ende 2021)
• Open Source bei GitHub

Einige Konzepte finden Sie in fast allen UI-Frameworks

• Trennung von Layout und Logik
• Adaptive Layouts
• Main- und Background-Threads
• Lose Kopplung zwischen Schichten