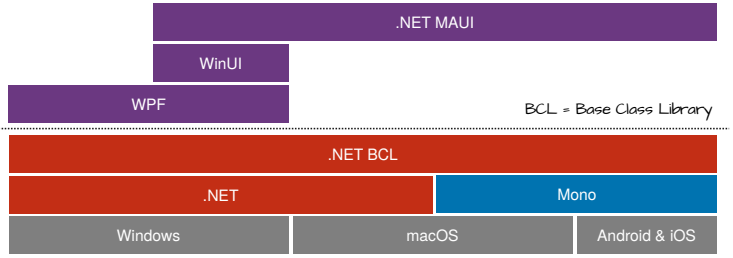
.NET-Familie (Auszug)

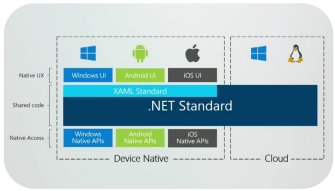
.NET 6.0



=> Jedes Jahr im November kommt eine neue Version
=> Gerade Versionsnummern (zB. .NET 6.0) haben "Long Time Support" (LTS)

XAML – Extensible Application Markup Language

- Beschreibungssprache von Microsoft zur Gestaltung **grafischer Oberflächen**
  - XML-basiert
  - Hierarchisch strukturiert (Baum)
  - Trennung von Layout und Code
- Verwendung in verschiedenen UI-Libraries
  - WPF, UWP, WinUI, Xamarin, .NET MAUI, …
- Unterschiedliche «Dialekte»
  - Standardisierung gestoppt (GitHub)
- Detailbetrachtung in **Block 09**

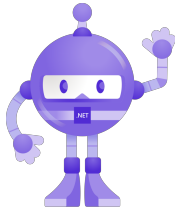


GrapeCity 2017: XAML Standard in .NET Core 2.0
https://bit.ly/2H9GQqC

=> Alles in XAML kann auch in C# definiert werden
=> XAML ist jedoch leichtgewichtiger und übersichtlicher

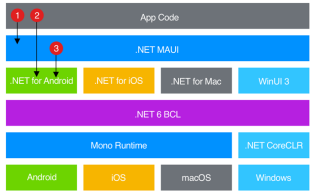
.NET MAUI – Multi-Platform App UI

- Cross-Platform UI-Framework
  - Mobile **Android** + **iOS**
  - Desktop **macOS** + **Windows**
- Veröffentlichung im Mai 2022
  - Auf Basis von **.NET 6** oder neuer
  - Evolution von **Xamarin.Forms**
- Klare Trennung zwischen Code und UI
  - Code in **C#**
  - User Interface in **XAML** (oder C#)



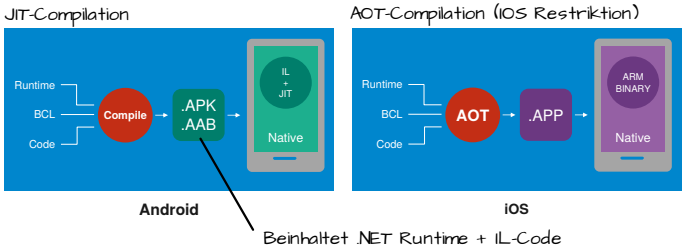
Funktionsweise

- Ziel: «***Write once, run anywhere***.»
- **Shared Code** – Code für alle Zielplattformen
  - (1) Eigener Code verwendet nur abstrakte Elemente aus MAUI
  - (3) MAUI bildet diese abstrakten Elemente auf plattform-spezifische Elemente ab
- **Plattform Code** – Code für eine Zielplattform
  - (2) Eigener Code verwendet Elemente aus einem der darunterliegenden UI-Frameworks
  - **Achtung:** dieser Code wird x-fach geschrieben!



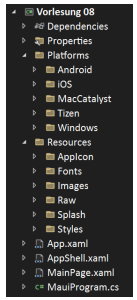
Microsoft 2022: How .NET MAUI works
https://bit.ly/3RgUFTe

Ausführung auf Android und iOS

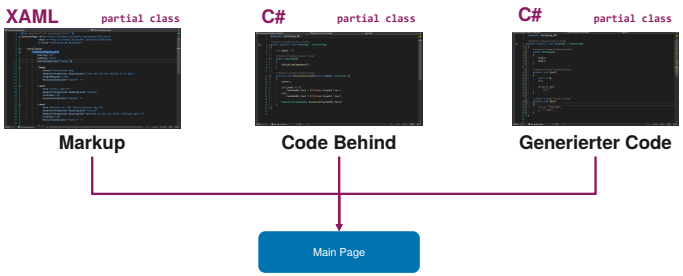


Hello .NET MAUI – Single Project

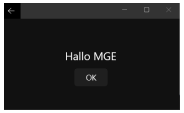
- `Platforms` Plattform-spezifischer (Startup-)Code
- `Resources` Für alle Plattformen verwendete Ressourcen
- `App.xaml` Einstiegspunkt in MAUI-Applikation
- `AppShell.xaml` Definition der visuellen Hierarchie mittels Shell
- `MainPage.xaml` Inhalt des ersten Fensters der App
- `MauiProgram.cs` Boostrapping der MAUI-Applikation (Builder)



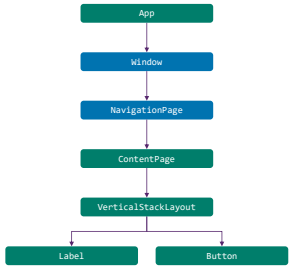
Verschmelzung von Darstellung und Code



Visual Tree und Logical Tree



- Visual Tree (**grün** + **blau**)
  - Vollständiger, visuell dargestellter Baum
  - Enthält Knoten, die wir nicht selber definieren
- Logical Tree (**grün**)
  - Vereinfachung des vollen Baums
  - Umfasst nur durch uns definierten Knoten



Namespaces

- Mit `xmlns` werden XML-Namespaces definiert z.B. "FontSize"
  - Ohne Doppelpunkt: **Standard-Namespace** → *Elemente können ohne Präfix verwendet werden*
  - Mit Doppelpunkt: **Benannter Namespace** → *Elemente müssen mit Präfix verwendet werden* z.B. "xClass"
- Übliche Namespaces in .NET MAUI
  - Der Standard-Namespace wird auf die .NET MAUI-Control Library gesetzt
  - `x` für XAML-spezifische Elemente
  - `local` für Elemente aus unserem eigenen Assembly (Projekt)

```
<Application xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:Vorlesung_09"
             x:Class="Vorlesung_09.App">
</Application>
```

XAML

Verwendung des Attributs «Class» aus dem Namespace «x». Hiermit wird die zugehörige Code Behind-Klasse definiert.

Named Elements

- Elemente können benannt werden
  - Ermöglicht Zugriff im Code Behind
  - Attribut führt zu Property in generierter Klasse
- In .NET MAUI existiert lediglich die Variante über das XAML-Attribut `x:Name`
- Andere Frameworks, z.B. WPF, kennen auch ein eigenes `Name`-Attribut
  - Beide Varianten führen schlussendlich zum selben Ergebnis
  - Es gibt kleine Unterschiede, die sind für uns aber nicht von Relevanz

```xaml
<Label x:Name="NamedElement" />
```
XAML

```
this.NamedElement.Text = "...";
```
Code Behind

Event Handler

- Reaktion auf Ereignisse der UI-Controls
  - Registrierung der Methode im XAML
  - Implementierung der Methode im Code Behind
- Methode übernimmt stets zwei Parameter
  - Sender (= Control) vom Typ `object`
  - Argumente abgeleitet von `EventArgs`
- Die Argumente enthalten Details zum Event

```xaml
<Button Clicked="Button_OnClicked" Text="Click Me!" />
```
XAML

```
private void Button_OnClicked(object sender, EventArgs args)
{
  // Wird beim Drücken des Buttons aufgerufen
}
```
Code Behind

Wir lernen in den Blöcken 11 + 12 bessere Alternativen kennen

Syntaxen*

Attribute Syntax

```xaml
<Label Text="Attribute Syntax"
       TextColor="Red"
       Background="Blue" />
```
XAML

«Type Converter» wandeln die `Strings` in passende Objekte um

Property Element Syntax

```xaml
<Label Text="Property Element Syntax">
    <Label.TextColor>
        <Color>Red</Color>
    </Label.TextColor>
    <Label.Background>
        <SolidColorBrush>
            <SolidColorBrush.Color>
                <Color>Blue</Color>
            </SolidColorBrush.Color>
        </SolidColorBrush>
    </Label.Background>
</Label>
```
XAML

** Soll gemäss Duden tatsächlich der korrekte Plural von Syntax sein …*

Content Properties

- Jedes XAML-Element kann **genau eine** Eigenschaft als seinen Inhalt definieren
- Dieser Inhalt kann in verkürzter Syntax «in das Element hinein» geschrieben werden
- Fördert die Lesbarkeit von Parent/Child-Beziehung
- Einige Elemente können, neben reinem Text, auch andere Elemente enthalten
  - Beispiel: `VerticalStackLayout`
  - Content Property: `Children`
  - Vermeidet aufwändige Property Element Syntax

```xaml
<Label Text="Inhalt" />
```
XAML

```xaml
<Label>Inhalt</Label>
```
XAML

```xaml
<VerticalStackLayout>
    <VerticalStackLayout.Children>
        <Label Text="Inhalt" />
        <Label>Inhalt</Label>
    </VerticalStackLayout.Children>
</VerticalStackLayout>
```
XAML

```xaml
<VerticalStackLayout>
    <Label Text="Inhalt" />
    <Label>Inhalt</Label>
</VerticalStackLayout>
```
XAML

Attached Properties

- Setzen einer Eigenschaft auf einem Element, welches zu einem anderen Element gehört
- Sprich: «Die Eigenschaft wird einem anderen Element ***angehängt***.»
- Wird meist bei Layouts verwendet
  - Layouts müssen Werte für Gestaltung kennen
  - Die Kind-Elemente definieren diese Werte
  - Pendant in Android: `layout_`-Attribute
- Fördert die Lesbarkeit des XAML

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

    <Label Grid.Row="0" x:Name="R" Background="Re" />
    <Label Grid.Row="1" x:Name="G" Background="Green" />
    <Label Grid.Row="2" x:Name="B" Background="Blue" />
</Grid>
```
XAML

Attached Properties in C#:

```
Grid.SetRow(R, 0)
Grid.SetRow(G, 1)
Grid.SetRow(B, 2)
```


Type Converters

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

Custom Control – Block 14

```
public class LocationControl : Label
{
    public Location Center
    {
        set => this.Text = $"{value.Lat} / {value.Long}";
    }
}
```
Control

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

```
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])
        };
    }
}
```
Type Converter

Markup Extensions

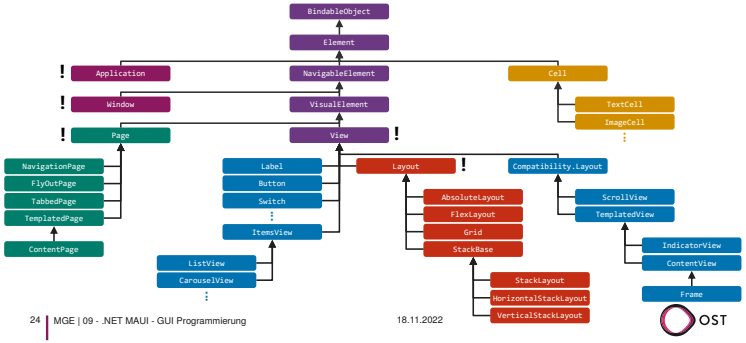
- Erlauben die Erweiterung des XAMLs mit **zusätzlicher Logik**
- Die aufzurufende Funktion wird zwischen geschweiften Klammern platziert: `{ … }`
- Häufige Verwendung für
  - **Styling** (Block 10)
  - **Data Binding** (Block 11)
- Eigene Extensions sind auch möglich
  - Beispiel: Ausgabe von formatierten Koordinaten

```xaml
<Label Text="..." FontFamily="{x:Null}" />
```
XAML

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

```
public class LocationExtension : IMarkupExtension<string>
{
    public string Latitude { get; set; }

    public string Longitude { get; set; }

    public string ProvideValue(IServiceProvider sp)
        => Latitude + " / " + Longitude;

    object IMarkupExtension.ProvideValue(IServiceProvider sp)
        => ProvideValue(sp);
}
```
Markup Extension

Klassenhierarchie (vereinfacht)



Application

- Legt via `MainPage` den ersten im Hauptfenster angezeigten Screen fest
- Erzeugung und Verwaltung von **Fenstern**
  - Erzeugt das Hauptfenster in `CreateWindow()`
  - Ermöglicht das Öffnen weiterer Fenster
  - **Achtung:** iOS unterstützt kein Multi-Window
- Erlaubt die Verarbeitung von **Lifecycle-Events** in überschreibbaren Methoden
  - Bei Multi-Window: für **alle** Fenster!
- Ermöglicht die zentrale Verwaltung von **app-weiten XAML-Ressourcen** (Block 10)

```
public partial class App : Application
{
    public App()
    {
        InitializeComponent();
        MainPage = new FirstPage();
    }

    // Konfiguration Hauptfenster (optional)
    protected override Window CreateWindow(…)
    {
        var window = base.CreateWindow(…);
        window.Title = "Name des Fensters";
        return window;
    }

    // Verarbeitung von Lifecycle Events (optional)
    protected override void OnStart() { … }

    protected override void OnResume() { … }

    protected override void OnSleep() { … }
}
```
App.xaml.cs

Window

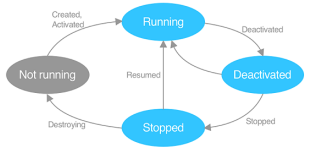
- Die Klasse `Window` repräsentiert ein **Fenster** innerhalb der Anwendung
- Fenster stellen `Page`-Objekte visuell dar
  - Hauptfenster: initialisiert durch `MainPage`
  - Zusatzfenster: initialisiert durch Konstruktor
- Zusatzfenster werden über das `Application`-Singleton angezeigt
- **Lifecycle-Events** für das Fenster können in Events oder Methoden verarbeitet werden
  - Für das Überschreiben der Methoden ist eine eigene Ableitung von `Window` nötig

```csharp
var window = new Window(new FirstPage())
{
    Title = "Neues Fenster"
};

// Abonnieren von Lifecycle-Events
window.Created += (_, _) => { … }
window.Activated += (_, _) => { … }
window.Deactivated += (_, _) => { … }
window.Stopped += (_, _) => { … }
window.Resumed += (_, _) => { … }
window.Destroying += (_, _) => { … }

// Anzeigen des Fensters via Application-Singleton
Application.Current.OpenWindow(window);
```

C#

App Lifecycle

- MAUI leitet wichtige **Lifecycle-Events** der darunterliegenden Plattformen weiter
- Verarbeitung möglich in
  - `Window` – komplett, via Events oder Methoden
  - `Application` – reduziert, nur via Methoden
- Vorsicht im Multi-Window-Modus: `Application` erhält Events für **alle** Fenster
- Empfehlung: Window-Variante verwenden

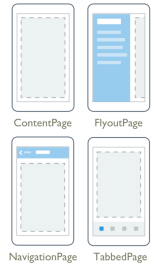


.NET MAUI	Android	iOS
Created	OnPostCreate()	FinishedLaunching()
Activated	OnResume()	OnActivated()
Deactivated	OnPause()	OnResignActivation()
Stopped	OnStop()	DidEnterBackground()
Resumed	OnRestart()	WillEnterForeground()
Destroying	OnDestroy()	WillTerminate()

Microsoft Docs 2022: App Lifecycle
https://bit.ly/3Dgg3E7

Pages

- Elemente zur Strukturierung und Gestaltung ganzer Screens
  - `ContentPage` Leere Screen ohne Zusatzelemente
  - `FlyoutPage` Slide-in Menu von links («Hamburger-Menu»)
  - `NavigationPage` Hierarchische Navigation mit Toolbar
  - `TabbedPage` Wechsel zwischen Inhalten mit Tabs
- Füllen typischerweise ihr Eltern-`Window` vollständig aus
- Die **Verschachtelung** von Pages ist möglich und üblich
  - `ContentPage`-Objekte innerhalb von `NavigationPage`
  - `FlyoutPage` in Kombination mit `NavigationPage`
- Achtung: Bedienbarkeit der Anwendung im Auge behalten



Microsoft Docs 2022: Pages
https://bit.ly/3RULbgF

FlyoutPage
.Flyout (Menü) = ContentPage
.Detail (Inhalt) = NavigationPage
=> Damit das Menü-Icon automatisch dargestellt wird.

Exkurs – Navigation in .NET MAUI

- Option 1 – `Application.MainPage`
  - Austausch der angezeigten Page
  - Nur für sehr einfache Apps genügend
- Option 2 – `NavigableElement.Navigation`
  - Erlaubt hierarchische Navigation (Stack)
  - Modale Navigation immer
  - Modeless Navigation nur mit `NavigationPage`
- Option 3 – Shell
  - Navigation auf Basis von URIs
  - Viele Eigenheiten und Spezialfälle
  - Einblick in Block 14 – bis dahin out-of-scope

```csharp
public partial class App : Application
{
    public App()
    {
        InitializeComponent();
        MainPage = new NavigationPage(new FirstPage());
    }
}
```

App.xaml.cs

```csharp
private async void ButtonClicked(object s, EventArgs e)
{
    await Navigation.PushAsync(new SecondPage());
}
```

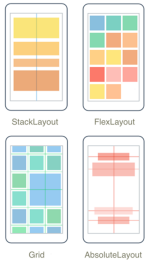
FirstPage.xaml.cs

```csharp
private async void ButtonClicked(object s, EventArgs e)
{
    await Navigation.PopAsync();
}
```

SecondPage.xaml.cs

Layouts

- Elemente zur Ausrichtung von Gruppierung von Views
  - `StackLayout` Horizontale oder vertikale Anordnung
  - `FlexLayout` Ähnlich wie Stack, mit Wrapping und mehr Gestaltung
  - `Grid` Anordnung in Zeilen und Spalten
  - `AbsoluteLayout` Absolute oder proportionale Anordnung im Layout
- Layouts sind **Container** für Kind-Elemente
  - Parent-Child Beziehung (Composite Design Pattern)
  - Verschachtelung möglich
  - Dieses Konzept kennen wir bereits aus Android



Microsoft Docs 2022: Layouts
https://bit.ly/3RwKOwm

Stack Layout

- Das `StackLayout` existiert (vermutlich) aus Kompatibilitätsgründen mit Xamarin
- In .NET MAUI sollten die optimierten Ableitungen verwendet werden
  - `VerticalStackLayout`
  - `HorizontalStackLayout`
- Abgesehen vom fehlenden `Orientation`-Attribut verhalten sich die Klassen gleich

```xaml
<VerticalStackLayout>
    …
</VerticalStackLayout>
```

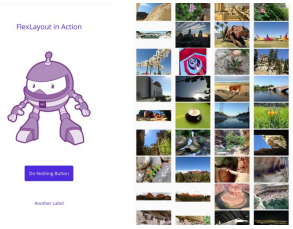
XAML

```xaml
<HorizontalStackLayout>
    …
</HorizontalStackLayout>
```

XAML

Flex Layout

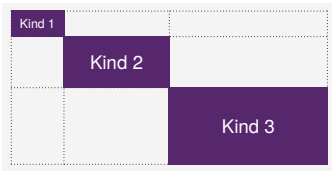
- Das `FlexLayout` ist eine ***flexible*** Variante des Stack Layouts
  - Komplizierter in der Anwendung
  - Langsamer beim Rendering
- Wichtige Eigenschaften des Layouts
  - `Direction` Richtung der Kinder
  - `Wrap` Automatischer Umbruch
  - `JustifyContent` Verteilung in Hauptrichtung
  - `AlignItems` Verteilung in Nebenrichtung
  - … und mehr …



Microsoft Docs 2022: FlexLayout
https://bit.ly/3eBTdN8

Grid

Zahl: Fixe Grösse
Auto: An Inhalt anpassen
*: Verfügbaren Platz füllen
Zahl*: Relative Grösse
Standard: Gleichmässige Verteilung (1*)



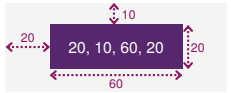

```xaml
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="1*" />
        <RowDefinition Height="2*" />
        <RowDefinition Height="3*" />
    </Grid.RowDefinitions>

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

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

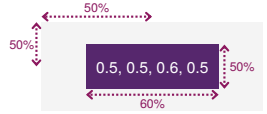
XAML

Attached Properties

Absolute Layout

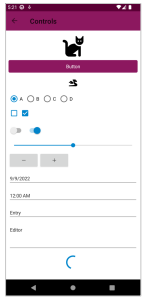
- Oft verwendet, um **Overlays** zu gestalten
- Variante 1 – **Absolute Werte**
  - Verwendung von absoluten Angaben
  - Positionierung in Bezug zu linker, oberer Ecke
  - Selten verwendet wegen Gerätevielfalt
- Variante 2 – **Proportionale Werte**
  - Verwendung von proportionalen Angaben
  - Bezug auf Grössen des Elternelements
  - Kombinierbar mit absoluten Werten





Views (≈ Controls in Android)

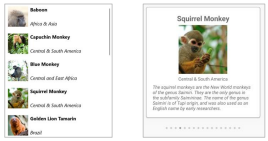
- Elemente, die für den Benutzer «interagierbar» sind
  - **Darstellung** `Label`, `Image`, `Border`, `Frame`, `ScrollView`, …
  - **Eingaben** `Entry`, `CheckBox`, `Slider`, `Switch`, …
  - **Aktionen** `Button`, `ImageButton`, `SearchBar`, …
  - **Aktivitätsanzeige** `ActivityIndicator` und `ProgressBar`
  - **Collections** `Picker`, `ListView`, `CarouselView`, …
- In anderen UI-Frameworks Controls oder Widgets genannt
- Werden mittels Handlers auf native Views abgebildet (Block 10)



Grössenangaben in Device-Independent Units (Relative Einheiten)

Collections und Collection-Views

- Collections enthalten mehrere Elemente desselben Typs, z.B. `Array<T>` oder `List<T>`
- Collection-Views stellen diese Inhalte dar
  - `ListView` Einfache Listen
  - `TableView` Gruppierte Listen
  - `Picker` Auswahl: 1 von N
  - `CarouselView` Horizontales Swiping
- Wichtige Eigenschaften dieser Views sind
  - `ItemsSource` Die darzustellende Collection
    nicht verfügbar für `TableView`
  - `ItemTemplate` Vorlage für Darstellung eines Items
    nicht verfügbar für `TableView` und `Picker`



Microsoft Docs 2022: Display Collections

Plattformspezifische Anpassungen

- Wunsch: eine UI-Definition, die auf allen Plattformen und Gerätetypen gut aussieht
- In Realität ist das nicht immer möglich – manchmal müssen wir optimieren
- Zwei Markup Extensions helfen
  - `OnPlatform` Werte pro Betriebssystem
  - `OnIdiom` Werte pro Gerätetyp
- Empfehlung: Default-Wert immer setzen

```
<Label Text="{OnPlatform '???',
              Android='Android',
              WinUI='Windows'}" />
```
XAML

```
<Label Text="{OnIdiom '???',
              Phone='Phone',
              Desktop='Desktop'}" />
```
XAML

Resource

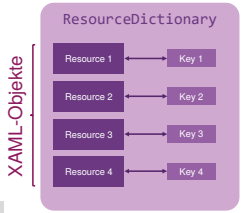
Auch Basistypen wie Zahlen, Strings, etc. (Einbindung des System.Runtime Namespace nötig)

- **Beliebiges Objekt**, das in XAML definiert werden kann
  - Brush
  - Color
  - String
  - …
- Besitzt eine **eindeutige Identifikation**
  - Zuweisung des XAML-Attributs `x:Key`
  - Erlaubt die spätere Referenzierung

=> Nicht zu verwechseln mit "Resource-Files" (Schriften, Bilder, Icons, etc.)
=> Resource-Files befinden sich zB. unter "Resources/Fonts".

Resource Dictionary

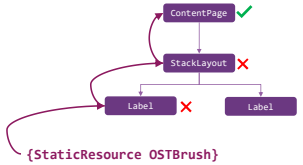
- Container zur Speicherung von **Resources**
  - Key-Value Speicher (~ Dictionary)
  - Attributs `x:Key` wird als Schlüssel verwendet
- In allen `VisualElement`-Ableitungen als Property `Resources` enthalten
  → **lokale Ressourcen**
- Ebenfalls auf der `Application`-Klasse als Property `Resources` definiert
  → **globale Ressourcen**




```
<Label Text="Background via Type Converter"
       Background="#6E1C50" />


<Label Text="Background via Brush">
  <Label.Background>
    <SolidColorBrush Color="#6E1C50"/>
  </Label.Background>
</Label>
```

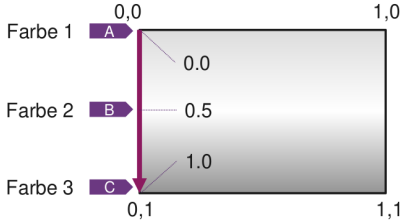

Auflösung von Resources

- Suchreihenfolge
  1. Aktuelles Element und alle Parent-Elemente
     (entlang dem Visual Tree in Richtung der Wurzel)
  2. In `Application.Resources`
- Die Suche bricht **beim ersten Treffer** ab
  - Reihenfolge im XAML-Tree entscheidend
  - Vorsicht bei mehrfach verwendeten Schlüsseln
- Falls Schlüssel **nicht gefunden** wird, so …
  - wird der Fehler ignoriert (XAML)
  - wird eine Exception geworfen (C#)




```
<Image Source="ost_logo.jpg" Aspect="AspectFit" />
<Label Text="Roboto (Regular)"
       FontFamily="RobotoRegular" />
```

```
<Label Text="Linear Gradient Brush">
  <Label.Background>

    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
A     <GradientStop Offset="0.0" Color="#6E1C50" />
B     <GradientStop Offset="0.5" Color="#AE5C90" />
C     <GradientStop Offset="1.0" Color="#6E1C50" />
    </LinearGradientBrush>

  </Label.Background>
</Label>
```
XAML



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}`


```
<ContentPage>
    <ContentPage.Resources>
        <SolidColorBrush x:Key="OSTBrush" Color="#6E1C50" />
    </ContentPage.Resources>

    <VerticalStackLayout>
        <Label Text="OK"
               TextColor="White"
               Background="{DynamicResource OSTBrush}" />

        <Button Text="Update"
                Clicked="UpdateResource" />
    </VerticalStackLayout>
</ContentPage>
```
XAML

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

Eigenständige Resource Dictionaries

- Separate `.xaml`-Datei mit XML-Root `<ResourceDictionary>`
  - Code Behind-Datei ist optional
  - Erzeugung gemäss Anleitung
- In andere Dictionaries als so genannte **Merged Dictionaries** integrierbar
- Prioritäten bei Schlüsselkollisionen
  1. Lokale Resources
  2. Merged Resources
     Später definierte Merges überschreiben frühere Merges!

```
<ResourceDictionary>
  <SolidColorBrush x:Key="OSTBrush2" Color="#6E1C50" />
</ResourceDictionary>
```
MyDictionary.xaml

Bei «Merges» zwingend – sonst optional

```
<ContentPage>
  <ContentPage.Resources>
    <ResourceDictionary>
      <SolidColorBrush x:Key="OSTBrush" Color="#6E1C50" />

      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="MyDictionary.xaml"/>
      </ResourceDictionary.MergedDictionaries>

    </ResourceDictionary>
  </ContentPage.Resources>

  <Label Text="Externe Resource"
         TextColor="White"
         Background="{StaticResource OSTBrush2}" />
</ContentPage>
```
MainWindow.xaml

x:Static – Zugriff auf statische Werte

- Gelegentlich ist es nötig, auf **statische Werte im C#-Code** zuzugreifen
  - Konstanten aus .NET
  - Konstanten im eigenen Code
- Zugriff via Markup Extension: `x:Static`
  - **Keine** XAML-Resources
  - Werte definiert in C#-Code

```
namespace Resources.Examples;

public static class MyRes
{
    public static readonly SolidColorBrush OSTBrush3 =
        new(Color.FromArgb("6E1C50"));
}
```
C#

```
<ContentPage xmlns:local="clr-namespace:Resources.Examples">

    <Label Text="Verwendung von x:Static"
           Background="{x:Static local:MyRes.OSTBrush3}" />

</ContentPage>
```
XAML

Explizite Styles

- Explizit darum, weil wir die Resource explizit den verwendenden Views **zuweisen müssen**

```
<ContentPage>
  <ContentPage.Resources>

    <Style x:Key="MyButtonStyle" TargetType="Button">
      <Setter Property="TextColor" Value="Black" />
      <Setter Property="BackgroundColor" Value="LightBlue" />
      <Setter Property="BorderColor" Value="Black" />
      <Setter Property="BorderWidth" Value="2" />
    </Style>

  </ContentPage.Resources>

  <HorizontalStackLayout>

    <Button Style="{StaticResource MyButtonStyle}"
            Text="OK" />

    <Button Style="{StaticResource MyButtonStyle}"
            Text="Cancel" />

  </HorizontalStackLayout>
</ContentPage>
```

XAML

Typ der View, auf welcher der Style angewendet werden soll

Zuweisung des Styles an alle verwendenden Views

Implizite Styles

- Ohne Key wirkt der Style automatisch **für alle Controls** des angegeben Typs

```
<ContentPage>
  <ContentPage.Resources>

    <Style TargetType="Button">
      <Setter Property="TextColor" Value="Black" />
      <Setter Property="BackgroundColor" Value="LightBlue" />
      <Setter Property="BorderColor" Value="Black" />
      <Setter Property="BorderWidth" Value="2" />
    </Style>

  </ContentPage.Resources>

  <HorizontalStackLayout>

    <Button Text="OK" />

    <Button Text="Cancel" />

  </HorizontalStackLayout>

</ContentPage>
```

XAML

Kein Style-Attribute mehr nötig

Theming im Eigenbau

- .NET MAUI kennt **kein** spezifischen **Theming**
- Der Mechanismus kann aber nachgebaut werden
  1. Mehrere Resource Dictionaries mit identischen Schlüsseln darin
  2. Laden des Standard-Theme als Merged Dictionary
  3. Zugriff auf alle Ressourcen via `DynamicResource`
  4. Laden eines neuen Dictionaries beim Wechsel des Themes

Einstellungen des Betriebssystems

- Mögliche Verbesserung für unseren Code: Berücksichtigung der **OS-Einstellungen**
  - Laden eines passenden Theme beim App-Start
  - Wechsel des Theme bei Änderungen im OS
- Glücklicherweise kennt MAUI passende APIs
  - Aktuelle Systemeinstellung auslesen `Application.Current.RequestedTheme`
  - Event für Benachrichtigung bei Änderungen `Application.Current.RequestedThemeChanged`

```
public partial class ThemingPage : ContentPage
{
  public ThemingPage()
  {
    InitializeComponent();
    LoadThemeBasedOnOperatingSystem();

    Application.Current.RequestedThemeChanged += (_, _) =>
    {
      LoadThemeBasedOnOperatingSystem();
    };
  }

  private void LoadThemeBasedOnOperatingSystem()
  {
    var activeTheme = Application.Current.RequestedTheme;
    var isLightThemeActive = activeTheme == AppTheme.Light;

    if (isLightThemeActive)
      ActivateTheme<LightTheme>();
    else
      ActivateTheme<DarkTheme>();
  }

  …
}
```

ThemingPage.xaml.cs

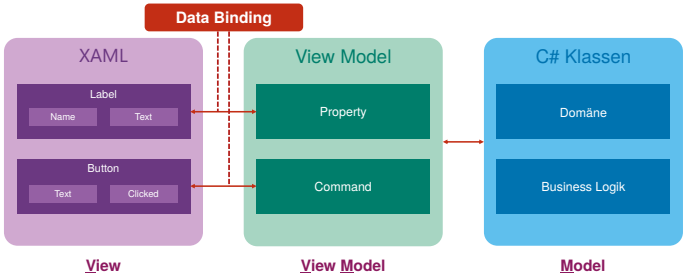
Markup Extension: `AppThemeBinding`

- Unsere eigene Lösung hat Vorteile
  - Erweiterbarkeit um zusätzliche Themes
  - Saubere Trennung der Resources nach Theme
- Für einfache Apps kann diese Lösung aber zu kompliziert sein
  - Nur Dark-/Light-Themes benötigt
  - Ein gemeinsames Dictionary gewünscht
- In diesen Fällen kann die Markup Extension `AppThemeBinding` verwendet werden
  - Automatische Anpassung auf Basis des OS

```
<Label Text="{AppThemeBinding Dark='Theme: Dark',
                              Light='Theme: Light',
                              Default='Theme: ???'}" />
```

ThemingPage.xaml

.NET MAUI ab nächster Woche



Data Binding in .NET MAUI

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

    public string LastName { get; set; } = "Kälin";
}
```

User.cs

```
<ContentPage>
    <StackLayout>
        <Label Text="{Binding FirstName}" />
        <Label Text="{Binding LastName}" />
    </StackLayout>
</ContentPage>
```

XAML

Data Binding mit ***Markup Extension***

```
public partial class MainPage : ContentPage
{
    private readonly User _user;

    public MainWindow()
    {
        InitializeComponent();

        _user = new User();

        this.BindingContext = _user;
    }
}
```

Code Behind

Datenquelle setzen

Binding – Eigenschaften

- `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

```
<!-- Attribute Syntax + Markup Extension -->

<Label Text="{Binding Path=FirstName,
                      Mode=TwoWay,
                      Converter={StaticResource MyCnv}}" />


<!-- Property Element Syntax -->

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

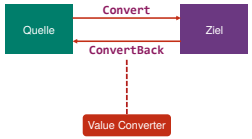
XAML

Binding – 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

Binding – Value Converter

- Hilfsobjekt zur Datenumwandlung zwischen Quelle und Ziel
  - Beispiel 1: `bool` zu `Color`
  - Beispiel 2: Strings umformatieren
- Implementieren `IValueConverter`
  - `Convert(…)` Quelle zu Ziel
  - `ConvertBack(…)` Ziel zu Quelle
- Erzeugung von Converter
  - Option 1: In Resources (`StaticResource`)
  - Option 2: In Code (`x:Static`)



Multi Binding

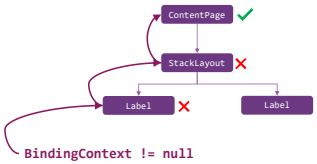
{} ist nur nötig, wenn das erste Zeichen im Format-String eine geschweifte Klammer ist (Escaping)

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

- Verwendung analog zu `Binding`
  - `Path`, `Mode`, `Converter`, etc.
- Unterschiede
  - Beliebig viele Quell-Eigenschaften
  - Nur Property Element Syntax (oder C# Code)
  - Converter mit `IMultiValueConverter`
- **Wichtig:** Sofern die Ziel-Eigenschaft nicht vom Typ `string` ist, muss zwingend ein **Multi Value Converter** verwendet werden

Binding Context

- Property der Klasse `BindableObject`
- Setzt die **Standardquelle** für Bindings
- Falls undefiniert: Traversierung des **Logical Trees** nach oben bis zum ersten Treffer
- Jeder `Path` ist relativ zum `BindingContext`
- Beliebige Objekte als Quelle möglich
  - C#-Klassen, MAUI-Elemente, etc.
  - Typischerweise: View Models (Block 12)



Binding Context überschreiben

- Der Binding Context lässt sich für einzelne Elemente anpassen
  - Option 1: Im Code Behind das Property `BindingContext` für das Element setzen
  - Option 2: Attribut `Source` im Binding setzen
- Dies ist eher unüblich – meist wird ein Binding Context pro `Page` verwendet

```
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();

        var user = new User();

        LastNameLabel.BindingContext = user;
        FirstNameLabel.BindingContext = user;
    }
}
```
Code Behind

```
<ContentPage>
    <ContentPage.Resources>
        <local:User x:Key="MyUser" />
    </ContentPage.Resources>

    <Label Text="{Binding Source={StaticResource MyUser},
                          Path=FirstName}" />

</ContentPage>
```
XAML

Weitere Quellen

- Mit `{RelativeSource …}` werden Elemente im Visual Tree referenziert
  - Suche beginnt beim aufrufenden Element
  - Beispiel: Binding an die Eigenschaft `Title` des beinhaltenden `ContentPage`-Objektes

```
<Label Text="{Binding Source={RelativeSource AncestorType={x:Type ContentPage}}, Path=Title}" />
```

- Mit `{x:Reference …}` werden Elemente über Namen referenziert
  - Namen müssen im gleichen Namensraum vorliegen
  - Beispiel: Binding an die Eigenschaft `Text` der View mit Name `MyText`

```
<TextBox Name="MyText" Text="Hallo MGE" />
<TextBox Text="{Binding Source= {x:Reference Name=MyText}, Path=Text}" />
```


POCOs als Binding Context

Das Label wird in diesem Beispiel stets den Initialwert «38» anzeigen!

```
<ContentPage>
    <Label Text="{Binding Age}" />
    <Button Text="Alter ++" Clicked="Increment" />
</ContentPage>
```
XAML

```
public partial class MainPage : ContentPage
{
    private readonly User _user;

    private void Increment(object sender, EventArgs e)
    {
        _user.Age++;
    }
}
```
Code Behind

```
public class User
{
    public string FirstName { get; set; } = "Thomas";
    public string LastName { get; set; } = "Kälin";
    public int Age { get; set; } = 38;
}
```
User.cs

- Als Datenquelle können **beliebige Objekte** verwendet werden, also auch POCOs*
- Unsere Bindings funktionieren – allerdings nur mit Einschränkungen (siehe Beispiel)
- Wir kennen bereits mögliche Lösungen
  - Block 06 – Observer Pattern
  - Block 07 – Observables bei Data Binding
- Die Observer-Variante in .NET heisst `INotifyPropertyChanged`

* *Plain Old CLR Objects (Wikipedia)*

Weitere Varianten: BindableBase (Generische Basisklasse)
Fody (IL-Weaving)

Variante 1 – Ohne Hilfsmittel

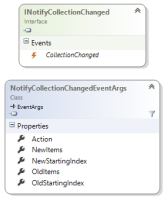
Property mit zugehörigem Backing Field (3)
Implementation von `INotifyPropertyChanged` (1)
Event Invoker: Erzeugt Argumente und löst Event aus (2)

```
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);
    }
}
```
User.cs

An dieser Stelle wird das GUI über die Änderung informiert.

INotifyCollectionChanged

- Bestandteil der .NET Class Library
  - 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



> Objekte **in** der Collection müssen Änderungen an ihren Daten via INPC ebenfalls kommunizieren

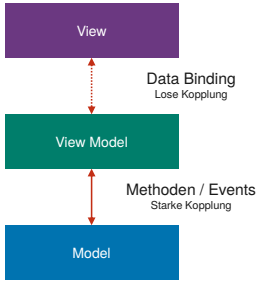
Cells

- Definieren den **Inhalt** des Item Templates
- Vier vorgefertigte Typen
  - `EntryCell` 1x Label und 1x Entry
  - `ImageCell` 1x Image und 2x Label
  - `SwitchCell` 1x Label und 1x Switch
  - `TextCell` 2x Label
- Eigene Typen via `ViewCell`-Element
  - Fungiert als simpler Container
  - Enthält Layouts und Views


```
<ViewCell>
    <VerticalStackLayout>
        <Label FontAttributes="Bold">
            <Label.Text>
                <MultiBinding StringFormat="{}{0} {1}">
                    <Binding Path="FirstName" />
                    <Binding Path="LastName" />
                </MultiBinding>
            </Label.Text>
        </Label>
        <Label Text="{Binding Age, StringFormat='{0} Jahre'}"
               FontSize="10" />
    </VerticalStackLayout>
</ViewCell>
```
XAML

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
  - .NET MAUI-Control Library*
- **View Model** enthält Darstellungslogik
  - C# Klasse mit INPC



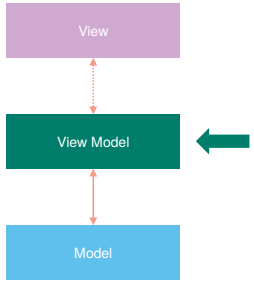
* Controls, Value Converters, Resources, etc.

=> Die View sollte möglichst "dumm" sein.
=> Verhalten und Logik gehören ins View Model.

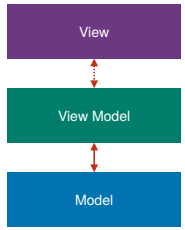
VM – View Model

- Enthält komplette **Logik der Darstellung**
- Typische Aufgaben
  - Formatierung von Model-Eigenschaften
  - Halten von Zuständen
  - Validierung von Benutzereingaben
  - Delegation von Benutzeraktionen an Model
- Bestandteile in .NET MAUI
  - C#-Klassen mit INPC

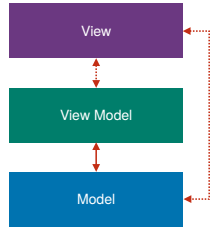
Hauptthema heute



Hauptvarianten



Klassisch



Durchgriff

View kennt nur das ViewModel
(ViewModel.FirstName)

View kennt auch das Model
(ViewModel.User.FirstName)

Vergleich der Hauptvarianten

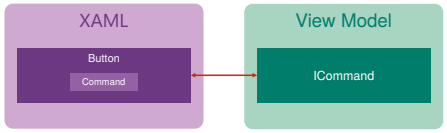
	Klassisch	Durchgriff
MVVM-Implementierung «nach Lehrbuch»	Ja	Nein
Saubere Trennung der Bereiche	Ja	Nein
Änderungen am Model haben Einfluss auf View Model	Ja	Nein[1]
Änderungen am Model haben Einfluss auf View	Nein[1]	Ja
Model frei von technologischen Details	Ja[1]	Nein[1]
Tendenz zu «versteckter» Darstellungslogik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

[1] *Im Normalfall*

[2] *Mögliche Verstecke: Model-Klassen, Value Converters, Markup Extensions, ...*

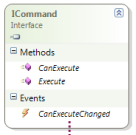
Aktionen in View Models

- Data Binding erlaubt die **Verknüpfung von Eigenschaften**, nicht aber von Methoden
- Lösung: Methoden in Objekte verpacken
  - `ICommand` definiert eine Schnittstelle für solche Objekte
  - View Models stellen für Aktionen `ICommand`-Objekte zur Verfügung
- `Command`-Eigenschaft von Views wird an `ICommand`-Objekte gebunden
  - `Button`
  - `ImageButton`
  - `SearchBar`
  - … und mehr …



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 Views 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);
    }
}
```

1
2

Command

```
// 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; }
}
```

View Model

```
<ContentPage>
    <StackLayout>
        <Button Text="Decrease Age"
                Command="{Binding DecreaseAgeCommand}" />
    </StackLayout>
</ContentPage>
```

XAML

Diskussion 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
    ➔ Einfache Lösung: Logik in normalen Klassen strukturieren
- Ein «Relay Command» ist in fast allen MVVM-Libraries enthalten
  - Beispiel: `RelayCommand` im .NET Community Toolkit

* *Auch "Delgate Command" genannt*

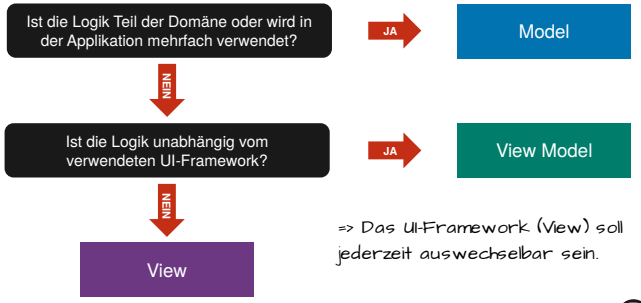
Commands mit Parametern

- Commands können **Parameter** übernehmen
  - `Execute(Object parameter)`
  - `CanExecute(Object parameter)`
- Der Parameter wird in der View gebunden
  - Attribut `CommandParameter`
  - Typischerweise Binding auf ein VM-Property
  - Alternativen möglich (z.B. statische Werte)
- Generisches Relay Command als Hilfsmittel
  - `RelayCommand<T>` im .NET Community Toolkit

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

XAML

Faustregeln zur Zuteilung von Logik



=> Das UI-Framework (View) soll jederzeit auswechselbar sein.

Erzeugung von Views und View Models

Im Code Behind der View

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

```
public PageB()
{
    InitializeComponent();
    BindingContext = new ViewModelB();
}
```
Code Behind B

Dependency Injection über BindingContext

```
var vm = new ViewModelB();
var view = new PageB();
view.BindingContext = vm;
view.Show();
```
Code Behind A

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

Dependency Injection über Konstruktor

```
var vm = new ViewModelB();
var view = new PageB(vm);
view.Show();
```
Code Behind A

```
public PageB(ViewModelB vm)
{
    InitializeComponent();
    BindingContext = vm;
}
```
Code Behind B

Software mit Schnitten in Teile zerlegen

- Hauptgründe für Schnitte
  - Fachliche, technische oder organisatorische Grenzen
  - Positiver Einfluss auf SW-Qualitätsmerkmale
  - Sorgen bei grossen Projekten für Überblick
- Anzahl und Namen der Schnitte sekundär
  - Unterschiede je nach Technologie
  - Unterschiede je nach Firma
  - Unterschiede je nach Architekten
  - …

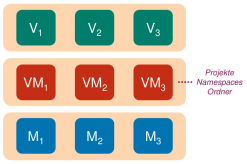
Nützliche .NET MAUI Libraries:

.NET Community Toolkit (Allgemeiner, beinhaltet zB. MVVM-Implementationen)

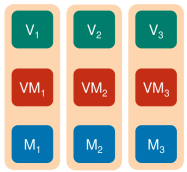
.NET MAUI Community Toolkit (Spezifisch .NET MAUI, beinhaltet zB. Views und Layouts)

Horizontale und vertikale Schnitte

- **Horizontale Schnitte**
  - Traditioneller Ansatz
  - Geeignet für «Technologie Teams»
  - Austausch von Technologien einfacher[1]
- **Vertikale Schnitte**
  - Modernerer Ansatz
  - Geeignet für «Feature Teams»
  - Austausch von Technologien schwieriger[1]



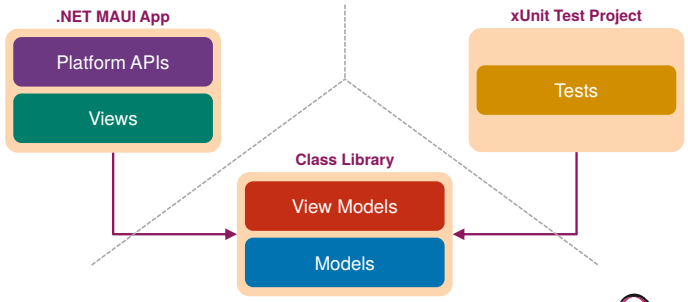
Horizontal = Technisch



Vertikal = Fachlich

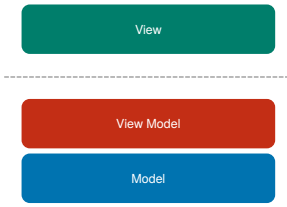
[1] *Sofern überall dieselben Technologien verwendet werden*

Beispiel – Technologie Trennung in einer Solution



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?



SOLID – Dependency Inversion Principle

«One should depend upon abstractions, [not] concretions.»

– Robert C. Martin

- Programmierung gegen Abstraktionen
  - Abstrakte Klassen
  - Interfaces
  - Delegates

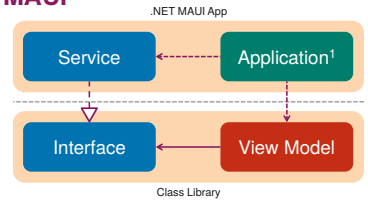
Would you solder a lamp directly to the electrical wiring in a wall?

Derick Bailey 2009 – SOLID Development Principles – In Motivational Pictures http://bit.ly/3cjlqVu

zB. Observer Pattern, Data Binding, Callback Methoden, etc.

Grundmuster für DI in .NET MAUI

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



In Tests wird der Service durch ein «Test Double» ersetzt

[1] *Alternative: ContentPage*

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

Testing

- Unit Tests profitieren stark von DI
  - Test-Projekt als zusätzliche «View»
  - Implementierung von «Test Doubles»
  - Injektion von Test Doubles
- Implementierung von **Test Doubles**
  - Variante 1: selber implementieren
  - Variante 2: Mocking Library verwenden
- Bekannte Mocking Libraries
  - FakeItEasy
  - Moq

```
// Beispiel verwendet xUnit und FakeItEasy

public class AsciiArtViewModelTests
{
    [Fact]
    public void Create_WhenNoImageWasChosen_ShowsError()
    {
        // Arrange
        var fakeService = A.Fake<IErrorService>();
        var testee = new AsciiArtViewModel(fakeService);

        // Act
        testee.CreateCommand.Execute(null);

        // Assert
        A.CallTo(
            () => fakeService.ShowErrorAsync("…","…"))
            .MustHaveHappened();
    }
}
```

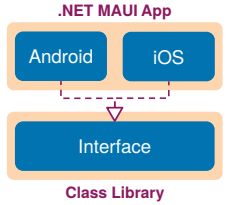
Test Code

Zugriff auf plattform-spezifische APIs

- Erkenntnis aus dem letzten Kapitel
  - Technologie-neutral Abstraktionen
  - Technologie-spezifische Implementierungen
- **Anwendungsfälle** für den Mechanismus
  - Fehlermeldungen anzeigen
  - Navigationen ausführen
  - Datei auswählen oder speichern
  - Berechtigungen prüfen
  - SMS versenden
  - Haptisches Feedback geben
  - Foto aufnehmen
  - … und viele mehr …

Umsetzung mit Plattform-APIs

- Existieren keine APIs in MAUI, machen wir die plattform-spezifischen Services selber
  - Eine Implementierung pro Zielplattform
  - Gutes Wissen über alle Zielplattformen nötig
- Umsetzungsvarianten
  - Conditional Compilation
  - Multi-Targeting



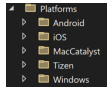
Umsetzung mit Plattform-APIs

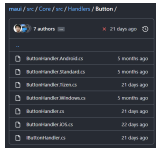
- **Conditional Compilation**
  - Alle Implementierungen in einer Datei
  - Trennung der Plattformen über Direktiven: `#if`, `#elif`, `#else` und `#endif`
  - Führt schnell zu schlecht lesbarem Code

```
#if ANDROID
    IWindowManager windowManager = …
#elif IOS
    UIInterfaceOrientation orientation = …
#else
    throw new NotImplementedException();
#endif
```

C#

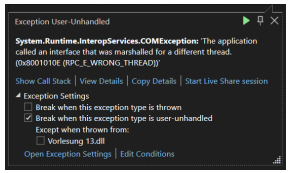
- **Multi-Targeting**
  - Implementierung in mehreren Dateien
  - Trennung über Projekteinstellungen
  - Bevorzugter Weg in .NET MAUI





Main / Background Threads

- Viele UI-Frameworks kennen diese Trennung
  - **Main Thread** für die Aktualisierung des GUI
  - **Background Thread** für langlaufende Aktionen
- Gilt auch für .NET MAUI, da es auf den plattform-spezifischen APIs aufsetzt
- Stolperfallen
  1. Langlaufende Operationen auf dem Main-Thread blockieren die GUI
  2. Aktualisierungen des GUI aus Background-Threads führen zu Exceptions



Aktualisierung des GUI vom Background-Thread

- MAUI enthält eine **Platform Integration** für das Problem
- Via Klasse `MainThread` lässt sich Code an den Main Thread delegieren
- Synchrone und asynchrone Varianten
  - `MainThread.BeginInvokeOnMainThread()`
  - `MainThread.InvokeOnMainThreadAsync()`

```
Task.Run(() =>
{
    // Ausführung auf Background-Thread
    // Läuft *parallel* zum Main-Thread!
    Thread.Sleep(2000);

    MainThread.BeginInvokeOnMainThread(() =>
    {
        // Ausführung auf Main-Thread
        // Hier darf das GUI aktualisiert werden
    });
});

// Main-Thread läuft hier *parallel* zum Task weiter
```

C#

Auswirkungen auf View Models

- Die gute Nachricht
  - **Data Binding** erledigt die Delegation zum Main-Thread automatisch
  - Properties können also ohne Weiteres von einem Background-Thread verändert werden

```
// IsCalculating ist ein einfaches ViewModel-Property
IsCalculating = true;

Task.Run(() =>
{
    // Keine Delegation an Main-Thread nötig
    IsCalculating = false;
});
```

C#

- Die schlechte Nachricht
  - `ICommand.CanExecuteChanged`-Events müssen an den Main-Thread delegiert werden
  - Tipp: `RelayCommand` entsprechend erweitern

```
// Einmalige Initialisierung in Application:
RelayCommand.Dispatch = MainThread.BeginInvokeOnMainThread;

public sealed class RelayCommand : ICommand
{
    public static Action<Action> Dispatch { get; set; }

    public void RaiseCanExecuteChanged()
    {
        Dispatch(() =>
            CanExecuteChanged?.Invoke(this, EventArgs.Empty));
    }
}
```

C#

Vergleich der Varianten

	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 .NET MAUI	Nein[2]	Ja

[1] Angezeigter Screen muss "neu geladen" werden
[2] Translation-Service muss definiert und in .NET MAUI-Projekt implementiert werden → technologische Grenze

Eine erste, eigene View

- Leere Ableitung von MAUI-View
  - Übernimmt alle Attribute der Basisklasse
  - Eigene Attribute sind möglich (folgt später)
- Verwendung via Klassennamen nach Import des Namespaces im XAML
  - Präfix wie gewohnt frei wählbar
  - Auch als Typ für implizite Styles möglich


```
public class AlertLabel : Label
{
}
```

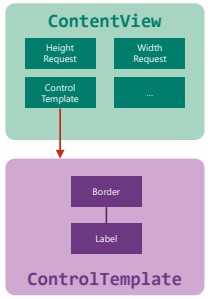
AlertButton.cs

```
<ContentPage xmlns:cc="clr-namespace:V_14.CustomControls">

  <ContentPage.Resources>
    <Style TargetType="cc:AlertLabel">
      <Setter Property="TextColor" Value="White" />
      <Setter Property="BackgroundColor" Value="DarkRed" />
    </Style>
  </ContentPage.Resources>

  <VerticalStackLayout Spacing="10">
    <cc:AlertLabel Text="Label 1" />
    <cc:AlertLabel Text="Label 2" />
  </VerticalStackLayout>

</ContentPage>
```

MyPage.xaml

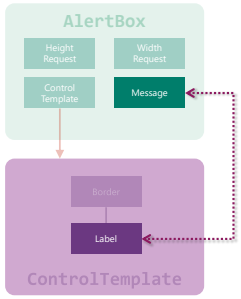
Control Templates

- Die View soll neu abgerundete Ecken haben
  - Ein `Label` selbst kann das nicht
  - Wir müssen dieses in eine `Border` verpacken
- Anders gesagt: wir müssen die **visuelle Repräsentation** der View steuern können
  - Elemente, die im **Visual Tree** eingefügt werden
- Die View muss dazu von `ContentView` ableiten
  - Struktur in `ContentView.ControlTemplate`
  - Element vom Typ `ControlTemplate`



Bindable Properties

- Die View soll ausserdem ein Property mit dem Namen `Message` bekommen
  - Benötigt so genannte Bindable Properties
  - Spezielle Properties mit Binding-Support
  - Normale .NET Properties funktionieren **nicht**
- Der Wert von `Message` soll im Label des Control Templates angezeigt werden
  - Verknüpung von Label mit AlertBox.Message
  - Markup Extension `{TemplateBinding …}`
  - Nur für Gebrauch in Control Templates



Beispiel – Control Templates und Bindable Properties

```
public partial class AlertBox : ContentView
{
  public AlertBox()
  {
    InitializeComponent();
  }

  public static readonly BindableProperty MessageProperty =
    BindableProperty.Create(
      nameof(Message),    // Name
      typeof(string),     // Datentyp
      typeof(AlertBox),   // Besitzer
      string.Empty);      // Initialwert

  public string Message
  {
    get => (string)GetValue(MessageProperty);
    set => SetValue(MessageProperty, value);
  }
}
```

AlertBox.xaml.cs

```
<ContentView>
  <ContentView.ControlTemplate>
    <ControlTemplate>
      <Border BackgroundColor="DarkRed"
              StrokeShape="RoundRectangle 10,10,10,10"
              Padding="5">
        <Label Text="{TemplateBinding Message}"
               TextColor="White"
               HorizontalTextAlignment="Center" />
      </Border>
    </ControlTemplate>
  </ContentView.ControlTemplate>
</ContentView>
```

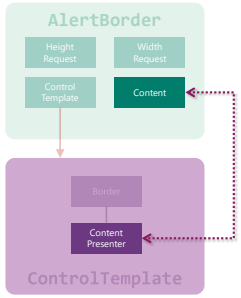
AlertBox.xaml


```
<ContentPage xmlns:cc="clr-namespace:V_14.CustomControls">

  <VerticalStackLayout Spacing="10">
    <cc:AlertBox Message="Box 1" />
    <cc:AlertBox Message="Box 2" />
  </VerticalStackLayout>

</ContentPage>
```

MyPage.xaml

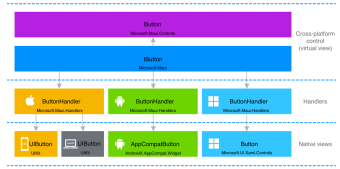
Content Presenter

- Letzter Schritt: beliebige XAML-Elemente statt einfachem Text darstellen
- `ContentPresenter` löst dieses Problem
  - XAML-Element nur für Control Templates
  - Fungiert als Platzhalter
  - Gibt den Inhalt der `Content`-Eigenschaft aus
- Kein eigenes Bindable Property nötig



Handlers – Anpassung der nativen View

- Bisher haben wir uns innerhalb der Grenzen von .NET MAUI bewegt
- Was aber, wenn die **native Repräsentation** einer MAUI-View beeinflusst werden soll?
- Dazu dienen **Handler**
  - Abbildung der MAUI-View auf die native Control
  - Standard-Handler für alle Views vorhanden
  - Anpassung oder eigene Handler sind möglich



Microsoft Docs 2022: .NET MAUI Handlers
https://bit.ly/3xJ9mal

> Die Anpassung oder Erstellung eigener Handler benötigt umfangreiches Wissen der Zielplattform(en). Wir thematisieren deshalb Handler im Rahmen des Kurses nicht weiter.

Was ist die Shell?

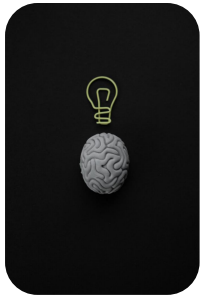
- **Vereinfachung**[1] von Anforderungen
  - Definition der Hierarchie in XAML
  - Navigation mit URIs
  - Suchfelder für Listen
- Die Verwendung ist **optional**
  - Bestandteil des Visual Studio-Templates
  - Bei nicht Verwendung einfach ausbaubar

Im Code: `MainPage = new AppShell();`

[1] *Zumindest ist Microsoft dieser Ansicht… mehr dazu gleich!* 😉

Persönliche Meinung zur Shell

- Die Shell vereinfacht einige Anforderungen
- Dies resultiert aber auch in
  - Einschränkungen im Funktionsumfang
  - Mehr Komplexität, um Einschränkungen zu umgehen
- In **eigenen Projekten** verzichte ich bisher auf die Shell
  - Meiste Funktionalität auch ohne Shell möglich
  - Kein Mix aus Shell und Non-Shell-Elementen nötig
  - Mehr Kontrolle und Flexibilität
  - Einfachere Trennung von UI- und Non-UI Code
  - Portierbarkeit bestehender Xamarin-Apps

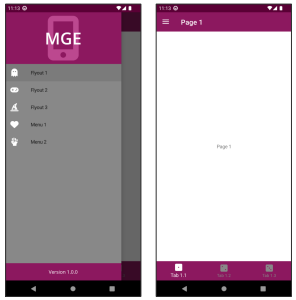


Trends: Flutter (Open Source GUI Framework von Google)
Ionic (Open Source GUI Framework für hybride Apps)
Model-View-Update (Alternative zu MVVM)
Jetpack Compose & SwiftUI (Alternativen zu XML-Sprachen)

Strukturierung

- Hauptvarianten
  - Flyout
  - Tabs
  - Kombination: Flyout und Tabs
- Anhand der Struktur im XAML wird ein passendes UI erzeugt
- Um schlankeres XAML zu ermöglichen, sind viele der Tags optional

> Empfehlung: Zwecks Lesbarkeit stets vollständigen Baum schreiben



MAUI vs. WPF

Einige Gemeinsamkeiten

- XAML-Features
  - Attribut Syntax und Property Element Syntax
  - Attached Properties
  - Markup Extensions
  - Type Converters
  - Logical Tree und Visual Tree
  - Resources, Resource Dictionaries, Styles, …
- `Application`-Klasse als Einstiegspunkt
- MVVM inkl. Data Binding und Commands
- Main- und Background-Threads
- Mehrsprachigkeit mit Resources oder RESX

Einige Unterschiede

- WPF unterstützt nur Windows
- Kein `MauiProgram` als Builder
- Keine `Platform`-Ordner in Solution
- Keine Pages, sondern nur `Window`
- Andere XAML-Namespaces
- Navigation zwischen Windows
- Benennung von Views, bzw. Controls
  Beispiel: `StackLayout` heisst `StackPanel`
- `BindingContext` heisst `DataContext`
- Kein Hot Reloading, dafür visueller Designer