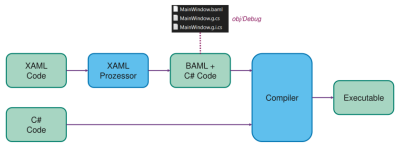
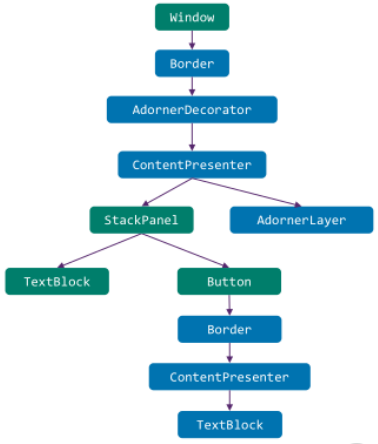
Code Behind
Every WPF Window has a XAML file for the graphical part and a C# (or any other .NET language) file. The code in the C# file is called the Code Behind. This file is used to implement the event handler.

XAML
XAML short for Extensible Application Markup Language is Markup Language from Microsoft for creating graphical UIs. XAML is based on XML.

Code to EXE



Logical View
The Logical Tree is the representation of your XAML. Only the controls **you** define are shown in the Logical Tree. In the figure only the green blocks are part of the Logical Tree.



Visual Tree The Visual Tree is the representation of your XAML including the elements you don't specify. The blue elements in the figure are added to the Tree. This Blue elements are part of the default control of a type. The Visual Tree in the figure is the whole tree.

Namespaces - XAML
In XAML you can define / import namespaces using the `xmln` keyword. Without a colon it is the default namespace. With colon, you have to specify the prefix to use the elements form the namespace.

Standard Namespaces - XAML
Some namespaces have some prefixes by conventions:

- default namespace points to the WPF Control Library
- `x` points to XAML specific elements
- `d` point to elements which are only required for the visual designer
- `mc` for elements of *Markup Compatibility*
- `local` for elements from you own assembly

!!Source 1!!

Event Handler - XAML
Every control has events. In the XAML you can just assign to the Name of the Event a string with name of the function / handler. This function is in the code behind class implemented.

!!Source 2!!

XAML Syntax's
In XAML we have two different kind of syntax's:

- the Attribute Syntax
- the Property Element Syntax

!!Source 3!!

Content Propery - XAML
Each XAML element can specify exactly one property for its content. This property can be set over The Attribute Syntax in XAML or you write the value between the start and end tag.

Attached Properties - XAML
Attached Properties are used to set the property on an element, but the property belongs to another element (The property is attached). This is mostly used in combination with Layouts. This is similar the Properties in Android witch are prefixed with `layout_`.

!!Source 4!!

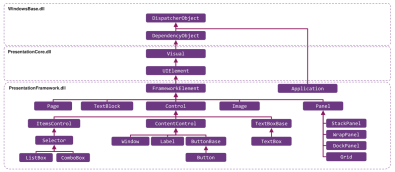
Type Converters
Type Converters are used to convert a string into a complex type. This is very often used to simplify the XAML. To implement a custom type converter you have to inherit from `TypeConverter`

!!Source 5!!

Markup Extensions - XAML
Markup Extensions are used in the XAML markup to use additional logic. They are very common for styling and data binding. To implement a custom Markup Extension you have to inherit from `MarkupExtension`.

!!Source 6!!

Class Hierarchy



TODO?!

Dimensions

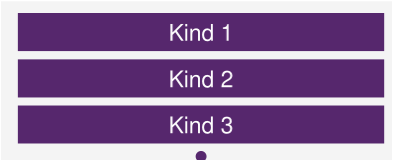
- no unit = device independent pixels (DIP), 1in
- px
- in
- cm
- pt
- Auto

Use only `Auto` and `DIP`.

- `n`: same value for all sides
- `x,y`: x for horicontal , y for vertical
- `l,t,r,b`: left, top, right, bottom

Panels / Layout
Panels / Layouts are Container for other child elements. They are very similar to the Layouts in Android. The panels could be nested, but you should not do this because of performance impacts.

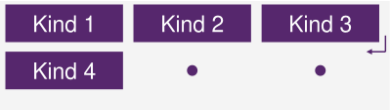
StackPanel
It's a layout in the WPF.



!!Source 7!!

WrapPanel

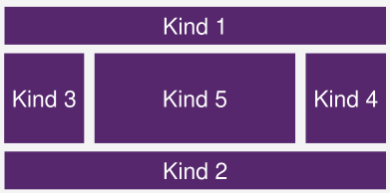
It's a layout in the WPF. If no space is left, the next element is wrapped on the next line.



!!Source 8!!

DockPanel

It's a layout in the WPF. The last element fills the remaining space. You can prevent this behavior using `LastChildFill ="False"`.



!!Source 9!!

Grid

It's a layout in the WPF. Using `RowSpan` and `ColSpan` multiple Rows / Columns can be merged into one.

!!Source 10!!

Resources in XAML

Every object which could be described in XAML (Brush, Color, String) could be a resource. A resource is similar to a variable in C# code. Every resource has a unique ID (`x:Key`).

Resource Dictionary

A Resource Dictionary is a container (key-value) to store resources. The attribute `x:Key` is the key. A local Resource Dictionary is available on all Elements inheriting from `FrameworkElement`. The `Application` class is used for global resources.
The Resource Dictionary is access over the Property Element Syntax.

!!Source 11!!

Resource Search Order

Search Order for Resources:

1. In the Logical-Tree from bottom to top
2. In **Application.Resources**
3. In **System.Ressourcen**

The search is abort at the first hit. If the Key is not found a Runtime-Exception is thrown.

Static Resources

Static Resources are only evaluated once during the compile time. During the runtime it is not possible to change its value.

!!Source 11!!

Dynamic Resources

Dynamic Resources can be evaluated multiple times. The evaluation is during the runtime and is updated when the value is changed.

!!Source 11!!

Resource Dictionary in other file

Sometimes you want to extract your Resource Dictionary into a separate XAML file. This file contains the `<ResourceDictionary>` tag as XML root. In the other file you can include separate XAML file using `MergedDictionaries`:

!!Source 14!!

Access static C# from XAML

Sometimes you have to access static values in the C# code. For example constants from .NET or form your own code. You can access the value over th markup extension `x:Static`.

!!Source 15!!

Styles in WPF

Styles are a XAML object which group multiple setters together. The style can be used inside a Resource Dictionary.
Without styles, you have to set every property, for example the background, on each element. This leads to code duplication.

!!Source 17 / 18!!

Explicit Styles

Explicit Styles required the property `x:Key`. !!Source 17!!

Implicit Styles

Implicit Styles must not have the property `x:Key` explicit set. But requires the property `TargetType`. The property `x:Key` gets automatically the value `x:Key ="{x:Type <Type>}`

!!Source 18!!

Extend your styles

You can specify a base style, for example for your buttons. Then you have two possibilities to extend your style:

1. Assign the style to the target type and set the specific Properties manually on the element
2. Create a new style which is based on the base style.

!!Source 19!!

Triggers in WPF

Triggers allow you to modify your styling of an element based on its state.

!!Source 20!!

Themes in WPF

WPF does not know the concept of skins / themes. But using Resource Dictionaries this concept could be created. The desired theme / resource dictionary is loaded at runtime using `DynamicResource`.

Control Templates

Control Templates are used to describe how a control should be display → which element should be inserted in the What is the Visual Tree in WPF?. You can access the current template using the attribute `Control.Template`.

Define Custom Control Template

You can define the custom Control Template:

- as a Resource
- inside a style

The attribute `ContentPrester` is used to define which attribute is used for the content. The markup extension `TemplateBinding` is used to bind attributes. This markup extension can only be used inside the Control Template.

GUI Guidelines

WPF does not have any guidelines like Android. However, you could / should use the same guidelines / libraries which are used in other places.

- Material Design in XAML
- MahApps.Metro

Data Binding
Data Binding in WPF works similar to the Data Binding in Android. In WPF you use the markup extension `Binding` to bind property from the code behind. To make the data from the code behind available for the XAML you have to set the `this.DataContext`.

Binding Types
In WPF exist three different biding types:

- `Binding` for 1:1 linking
- `MultiBinding` for 1:n linking
- `PriorityBinding` for 1:n / 1:1 linking At the start the first source which response is used (turns a 1:n to 1:1 linking)

Binding Properties
A Binding object in WPF has the following properties:

- `Path`: Name of the source property
- `Mode`: Direction of the data flow
- `Converter`: Conversion between source and destination. Is only used in Binding. It is not the same as type converters.

Binding Modes

- `OneTime`: One-time update (very efficient, because no monitoring of the elements)
- `OneWay`: Destination is updated when source is changed
- `OneWaytoSource`: Source is updated when destination is changed (input fields)
- `TwoWay`: Changes are propagated in both ways
- `Default`: Value depends on destination property

!!Source 22!!
Value Converter
The Value Converter is a support object used in data binding in WPF. The Value Converter converts the between source and destination so that it can be displayed on a from. It also converts the value from an input field back to the class representation.

Value Converter Implementation
To implement a Value Converter you must implement the `IValueConverter` interface. The `IValueConveter` interface provides the following methods:

- `Convert()`: Convert from source to destination
- `ConvertBack()`: Convert from destination into source

For a multi binding converter you must implement the `IMultiValueConverter` interface. The Value Converter can be created:

- as a resource
- in code

!!Source 23!!
Multi Binding
Multi Binding is a special case of the Data Binding. In contrast to normal Data Binding multi binding can bind multiple source properties.
!!Source 24!!
DataContext
The `DataContext` is used to define the default source for data binding. If the `DataContext` is not set the logical tree is traversed to the top.
Even it is possible to set the `DataContext` for single elements this is not very common. Normally you have one `DataContext` for one Window.

WPF Element as Source
Using the markup extension `RelativeSource` you refer to elements in the visual tree. Try to avoid this markup extension because it is difficult to read and complex. The markup extension `ElementName` can be used to refer to elements by their name.
!!Source 25!!
Notify UI about data changes
For Data Binding in WPF you can use any object as model. However, the UI will not react on model changes. To notify the UI you have to implement the `INotifyPropertyChanged` interface. During the creation of the binding the destination register itself on the Event `PropertyChanged` If the property is changed the Event is fired and the UI can be updated.
!!Source 26!!
IL Weaving
Using IL Weaving you modify the IL Code (MSIL) at compile time. This should only be used in exceptional cases because debugging is damn hard.

Data binding using collections
To bind a Collection to the UI you have to use a special collection type: `ObservableCollection<T>`. This collection must be filled and then as the DataContext set.
The `ObservableCollection<T>` implements the `INotifyCollectionChanged` interface. This notifies the UI only if the collection is changed (add / remove). The model itself should implement the `INotifyPropertyChanged` interface. Otherwise, the changes on the model are not reflected on the UI. The model does not have to implement the `INotifyPropertyChanged` interface. But normally it should.
To display the data inside the collection the `ItemTemplate` and `DataTemplate` must be overwritten (see example).

Selectors for Collection in Data binding

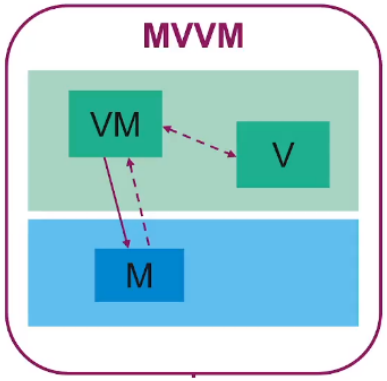
- `SelectedIndex`: Index in Collection
- `SelectedItem`: Element as Object
- `SelectedValue`: Selected Value
- `SelectedValuePath`: Object path to the value which is returned in `SelectedValue`

!!Source 27!!
MVVM
The MVVM Pattern consists of three parts:

- *Model*: contains the Domain / business logic
- *View*: contains the graphical UI and user input
- *View Model*: contains the logic form the UI and ensures the communication between Model and View
  - This is often achieved over Data Binding

The benefits of the MVVM pattern are:

- View Model is easy to test because it does not contain any UI classes
- the View has only visual tasks
- Changes in the Model do not directly affect the View



ViewModel in WPF !!Source 28!!

Bind Action to ViewModel

Using Data Binding you can only bind properties. Therefore methods must be packed in a object. For this the `ICommand` interface is used. !!Source 29!!

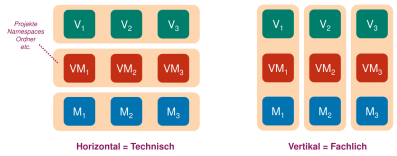
RelayCommand

The `RelayCommand` implements the `ICommand` interface and can be used to implement the actions directly on the ViewModel. It is an alternative to the normal `ICommand` usage. !!Source 30!!

Rule of Thumb MVVM

Separation in Software Engineering

The Horizontal approach is the traditional one. Changing the technology is easier. This approach is used when each team works on one technology.
The Vertical approach is the modern approach. It fits best for features teams. Changing the technology can be difficult when everywhere should be used the same technology.



Dependency Inversion Principle

DIP = Dependency Inversion Principle

> One should depend upon abstractions, [not] concretions.
>
> – Robert C. Martin

You want to implement against

- abstract classes
- interfaces
- delegates

You would never solder a lamp directly to the electrical wiring.

Localization in WPF

In WPF you can create localization using

- Resources
- over RESX files (recommended)

In general, it is a good idea to create a *Translation Service* for the access in C# code. With this approach the translation is technology independent.

	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

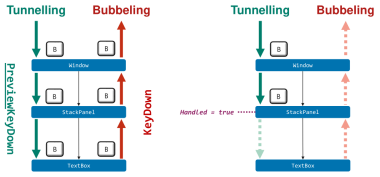
[1] Nur mit Eigenbau. Eigene Wrapper-Klasse mit INPC implementieren, die das UI beim Sprachwechsel benachrichtigt
[2] Nur mit Eigenbau. Translation-Service definieren und in WPF-Projekt implementieren

Routed Events

A UI Event in WPF moves in two phases through the Visual Tree:

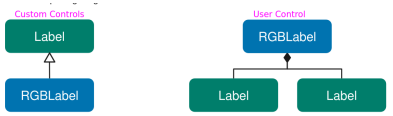
- *Tunneling*: Top to bottom to focused element
- *Bubbeling*: Bottom to top from focused element

The traversing can be stopped at any element. You only have to set `RoutedEventArgs.Handled` to `false`.



Custom Controls / User Controls

A Custom Control extends an already existing control. The User Control is a composition from two existing controls.
In practice the custom control are more common because you have more control over it (styling). The User Control can also be achieved with Custom Control. You only have to overwrite the Control Template.



Threading Model

Every WPF application has minimal two threads:

- UI Thread: Manages UI, receives events, execute actions
- Rendering Thread: runs in background, draws controls

Thanks to this mechanism, controls are draw event when the UI Thread is blocked. Only the UI Thread is allowed to change the UI.
!!Source 31!!

UI update from background thread

In WPF only the UI Thread is allowed to update the UI. Using a Dispatcher you can run an operation on the UI Thread from the background. In a WPF Project you can access the dispatcher:

- Code Behind: `this.Dispatcher`
- Other classes: `Application.Current.Dispa`

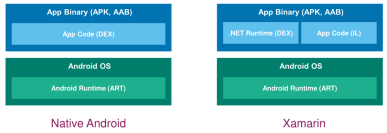
Data Binding does the dispatching automatically. But the `ICommand.CanExecuteChanged` Event must be dispatched manually. This could be done if you extend the RelayCommand. !!Source 32!!

Xamarin

Xamarin is used to write apps in C# / F# for:

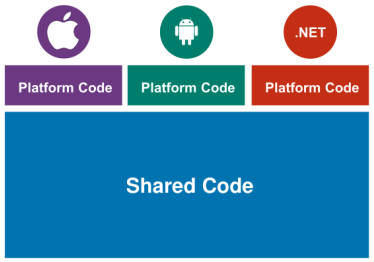
- Android Open Source Project
- iOS
- iPadOS
- watchOS
- tvOS
- macOS

During the compilation Native apps are created. When compiled for Android the .NET Runtime is packed together. So the runtime runs a runtime which runts the app.



Xamarin Structure

You have *Shared Code* and *Platform Code*. The Shared Code should be as big as possible while the platform code should be as small as possible.



Xamarin Essentials

Xamarin.Essentials is a collection of platform services like:

- Sensors (battery, ...)
- Interfaces (permissions, phone, ...)
- Utilities (Threading, conversion, ...)

It is one interface for all platform.

Xamarin Traditional

In Xamarin Traditional you write the definition for the UI for each target platform using the native concepts. The benefits are that it is faster, and you have total access to the underlying SDK. The drawback is you have to write the same UI for each platform.

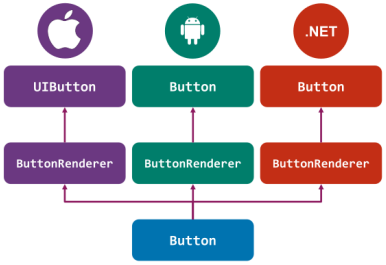
Xamarin.Forms

In Xamarin.Forms you write the UI as a part of the shared code using XAML. The benefit is you write the UI only once. The drawback is you have some limitations in the design of the UI and slightly slower performance then Native.

However, in general you should use Xamarin.Forms instead of Xamarin Traditional.

Xamarin Renderer

Xamarin has many controls for example the button. Render classes are used to map the Xamarin Button to the platform specific buttons. Using Renderer you can adjust the styling. You can also write custom renderers for existing controls and for custom XAML Controls.



1 Code

Source 1 - Default Namespaces

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
    presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
            compatibility/2006"
        xmlns:local="clr-namespace:Vorlesung_09"
        mc:Ignorable="d" <!-- Tells XAML that d should not be used
            during compilation -->
        />
```

Source 2 - Event Handlers

```
<Button Click="MyButton_OnClick" Text="Click Me!" />

void MyButton_OnClick(object sender, RoutedEventArgs evtArgs) {}
```

Source 3 - XAML Syntax's

```
<!-- Attribute Synatx -->
<Button Content="Mein Button"/>

<!-- Propery Element Synax -->
<Button>
  <Button.Background>
    <SolidColorBrush Color="Blue"/>
  </Button.Background>
  <Button.Content>
    Mein Button
  </Button.Content>
</Button>
```

Source 4 - Attached Properties

```
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="30" />
    </Grid.RowDefinitions>
    <TextBlock Grid.Row="0" Name="G" Background="Green" />
</Grid>
```

Source 5 - Type Converter

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

Source 6 - Markup Extensions

```
<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;
    }
}
```

Source 7 - StackPanel

```
<!-- Orientation could be Vertical or Horiztonal -->
<StackPanel Orientation="Vertical">
</StackPanel>
```

Source 8 - WrapPanel

```
<!-- Orientation could be Vertical or Horiztonal -->
<WrapPanel Orientation="Horizontal">
</WrapPanel>
```

Source 9 - DockPanel

```
<!-- Top | Bottom | Left | Right -->
<DockPanel>
  <TextBlock DockPanel.Dock="Top" />
  <TextBlock DockPanel.Dock="Left" />
</DockPanel>
```

Source 10 - Grid

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="1*" />
    <RowDefinition Height="2*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <TextBlock Grid.Row="0" Grid.Column="0" />
</Grid>
```

Source 11 - Resource Dictionary

```
<Window>
  <Window.Resources>
    <SolidColorBrush x:Key="OSTBrush" Color="#6E1C50" />
  </Window.Resources>
  <Label Content="Variante 1" Foreground="White">
    <Label.Background>
      <StaticResource ResourceKey="OSTBrush" />
    </Label.Background>
  </Label>
  <Label Content="Variante 2"
         Foreground="White"
         Background="{StaticResource ResourceKey=OSTBrush}" />
  <Label Content="Variante 3"
         Foreground="White"
         Background="{StaticResource OSTBrush}" />
  <Label Content="Variant 4"
         Foreground="White"
         Background="{StaticResource OSTBrush}" />
</Window>

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

Source 14 - Additional Resource Dictionary

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

<!-- MyForm.xaml -->
<Window>
  <Window.Resources>
    <ResourceDictionary>
      <SolidColorBrush x:Key="OSTBrush" Color="#6E1C50" />
      <ResourceDictionary.MergedDictionaries> <!-- Include -->
        <ResourceDictionary Source="MyDictionary.xaml"/>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Window.Resources>
  <Label Content="Externe Resource"
         Foreground="White"
         Background="{StaticResource OSTBrush2}" />
</Window>
```

Source 15 - Static C# Values

```
public static class MyRes
{
    public static readonly SolidColorBrush OSTBrush =
        new(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}" />
```

Source 17 - Explicit Styles

```
<Window>
  <Window.Resources>
    <!-- TargetType is optional when used explicitly -->
    <Style x:Key="MyButtonStyle" TargetType="Button>
      <Setter Property="Button.Background" Value="Blue" />
      <Setter Property="Button.Foreground" Value="Black" />
      <Setter Property="Button.BorderBrush" Value="Black" />
      <Setter Property="Button.BorderThickness" Value="1" />
    </Style>
  </Window.Resources>
  <StackPanel>
    <Button Style="{StaticResource MyButtonStyle}"
            Content="OK" />
    <Button Style="{StaticResource MyButtonStyle}"
            Content="Cancel" />
  </StackPanel>
</Window>
```

Source 18 - Implicit Style

```
<Window>
  <Window.Resources>
    <!-- TargetType required -->
    <Style TargetType="Button>
      <Setter Property="Button.Background" Value="Blue" />
      <Setter Property="Button.Foreground" Value="Black" />
      <Setter Property="Button.BorderBrush" Value="Black" />
      <Setter Property="Button.BorderThickness" Value="1" />
    </Style>
  </Window.Resources>
  <StackPanel>
    <Button Content="OK" /> <!-- No style assigned -->
    <Button Content="Cancel" />
  </StackPanel>
</Window>
```

Source 19 - Extended Style

```
<!-- (1) combination approach -->
<Button Style="{StaticResource NormalButton}"
        Background="Red"
        Content="Cancel" />

<!-- (2) based on approach -->
<Window.Resources>
  <Style x:Key="NormalButton" TargetType="Button">
    ...
  </Style>
  <Style x:Key="DangerButton"
         BasedOn="{StaticResource NormalButton}" <!-- here you inhert
                 from NormalButton -->
         TargetType="Button">
    <Setter Property="Background" Value="Red" />
  </Style>
</Window.Resources>
...
<Button Style="{StaticResource DangerButton}"
        Content="Cancel" />
```

Source 20 - Triggers

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

Source 22 - Binding Modes

```
<TextBox Text="{Binding Result, Mode=OneWay}" />
```

Source 23 - Value Converter

```
public class ReverseConverter : IValueConverter
{
    public object Convert(object value, /*More stuff*/)
    {
        var stringValue = (string) value;
        var reversedChars = stringValue.Reverse().ToArray();
        var reversedString = new string(reversedChars);
        return reversedString;
    }
    public object ConvertBack(object value, /*More stuff*/)
    {
        return Convert(value, /*more stufmore stufff*/);
    }
}

<Window xmlns:converter="clr-namespace:...()">
  <Window.Resources>
    <converter:ReverseConverter x:Key="RConverter" />
  </Window.Resources>
  <TextBox Text="{Binding
                  Path=FirstName,
                  Mode=TwoWay,
                  Converter={StaticResource RConverter}}" />
</Window>
```

Source 24 - Multi Binding

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

```
    </MultiBinding>
  </TextBlock.Text>
</TextBlock>
```

Source 25 - Elements as Source

```
<Label Content="{Binding RelativeSource={RelativeSource FindAncestor,
    AncestorType=Window}, Path=Title}" />
<TextBox Name="MyText" Text="Hallo MGE" />
<TextBox Text="{Binding ElementName=MyText, Path=Text}" />
```

Source 26 - INotifyPropertyChangedo

```
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);
    }
}
```

Source 27 - Collection Data binding

```
public partial class MainWindow : Window
{
    private ObservableCollection<User> users;
    public MainWindow()
    {
        InitializeComponent();
        users = new ObservableCollection<User>();
        this.DataContext = users;
    }
}
// Normal User Model class (POCO)

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

Source 28 - ViewModel

```
// User is a POCO
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}";
}

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

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

```
</Window>
```

Source 29 - Bind action to ViewModel

```
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}"
            CommandParameter="{Binding Selecteduser}"/> <!-- Example
                how to pass arguments -->
  </StackPanel>
</Window>
```

Source 30 - Relay Command

```
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();
    }
}
```

Source 31 - Threading

```
Task.Run(() =>
{
  // Run in background
});
```

Source 32 - Dispatching

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