HOCHSCHULE HAMM-LIPPSTADT

Praktische Informatik

Vorlesung 04

Einführung in die WPF

Zuletzt haben wird gelernt…

- Wie man Ereignisse mit Hilfe des Beobachter Musters umsetzen konnte.
- Wie man ein Ereignis-Ziel mit Hilfe von Delegaten realisieren kann.
- Wie man Ereignisquellen mit dem Schlüsselwort event umsetzt.
- Wie Ereignisse im .Net-Framework umgesetzt werden.
- Warum die Schnittstelle INotifyPropertyChanged benutzt wird, um Änderungen zu signalisiern.

Inhalt heute

- Windows Presentation Foundation
- XAML
- Code Behind
- Routed Events
- Dependency Properties

WPF

- Die sog. **Windows Presentation Foundation (WPF)** ist ein Framework zur Erstellung von grafischen Benutzeroberflächen (GUIs).
  - Die WPF ist Teil des .Net Frameworks.
  - Entwickler und Eigentümer ist die Firma Microsoft.
  - Microsoft benutzt die WPF zur Entwicklung eigener Produkte: Visual Studio.
- Teile der WPF können mittlerweile auf unterschiedlichen Plattformen genutzt werden.
  - Mit Xamarin können z.B. Apps für iOs, Android, usw. entwickelt werden.
  - Die eigentliche Heimat ist aber das Betriebssystem Windows.
- WPF basiert auf hunderten von Klassen, die man für die Entwicklung von GUIs benötigt.
  - Fenster, Steuerelemente, ...
  - WPF ist keine freie Software (OpenSource).
  - Man kann damit aber freie Software entwickeln.

WPF vs WinForms

- **WPF** ist der Nachfolger von **Windows Forms**.
  - WinForms Anwendungen benutzen die Steuerelemente, die Windows selbst bereit stellt (Win32).
  - Dadurch ist WinForms technisch an Windows gebunden.

- WPF hingegen rendert alle Steuerelemente selber.
  - Unter Windows wird DirectX benutzt, was eine entsprechende Grafikkarte voraussetzt.
  - WPF ist vektorbasiert, die Oberflächen können also leicht in der Größe verändert werden.

- WPF hat den Vorteil, dass eigene Steuerelemente erzeugt werden können.
  - Es können runde Buttons erzeugt oder ein Eingabefeld in einem Menü platziert werden, wenn man das will.
  - Zudem wird die WPF dadurch (zumindest theoretisch) plattformunabhängig.

Avalonia

- WPF ist ein Framework, welches lediglich mit Windows funktioniert.
  - Mit dem OpenSource Projekt Avalonia existiert aber ein sehr ähnliches Projekt.
  - Dieses erlaubt die plattformunabhängige Entwicklung von GUIs für Windows, Linux und OS X.

- Die meisten Beispiele in dieser Veranstaltung sollten auch mit Avalonia funktionieren.
  - Wenn Sie möchten, können Sie auch Ihr Semesterprojekt mit Avalonia entwickeln.
  - Zentraler Anlaufpunkt für Avalonia ist das Github-Repository unter https://github.com/AvaloniaUI/Avalonia.

Maui

- .Net Maui (Multi Platform App Ui) wird WPF bald ablösen.
  - Mit Maui können GUIs mit C# für Android, iOs, Mac und Windows mit einer gemeinsamen Code-Basis entwickelt werden.
  - Das funktioniert schon für Windows, aber für den Mac noch nicht richtig.
- In der Zukunft wird diese Veranstaltung sicher auf Maui setzen.
  - Zum Zeitpunkt der Erstellung war Maui aber noch nicht ausgereift.

Entwicklungsumgebungen

- WPF kann am besten unter Windows genutzt werden.
  - Das Visual Studio (auch Community) eignet sich zur Entwicklung von WPF Anwendungen am besten.
  - Mit Blend für Visual Studio steht zudem für Designer ein eigenes Werkzeug zur Verfügung, um XAML zu erzeugen.
- Wer experimentierfreudig ist, sollte sich MAUI ansehen.
  - Funktioniert noch nicht alles, aber vieles…
- Mit dem Framework **Xamarin Forms** kann auch auf dem Mac entwickelt werden.
  - Xamarin war eine eigenständige Firma, die Microsoft im Jahr 2015 übernommen hat.
  - Auch bekannt als die Entwickler des Mono-Framework, eine plattformunabhängige Implementierung des .Net-Frameworks.
- Xamarin nutzt ebenfalls XAML, um Anwendungen für iOS und Android entwickeln zu können.
  - Die Möglichkeiten sind dort aber kleiner als mit WPF.

Klassische GUI Programmierung

- In den meisten GUI Frameworks wird die Oberfläche ausschließlich über Anweisungen im Programmcode aufgebaut.
  - Man erzeugt bestimmte Objekte, konfiguriert diese und setzt daraus die GUI zusammen.

- Beispiel:
  - Erzeuge Fenster, erzeuge Button, lege Button im Fenster an der Position x/y ab, ...

- Diese Art der Oberflächengestaltung hat Nachteile:
  - Man kann das Design der Oberfläche und das Programmieren nicht voneinander trennen.
  - Designer können aber i.d.R. nicht programmieren.
  - Es entstehen meist recht „hässliche“ Oberflächen von Programmierern, die wenig anpassbar sind.

XAML

- Auch mit WPF kann eine Oberfläche auf diese Weise erzeugt werden.
  - Es gibt aber auch noch einen anderen, besseren Weg.

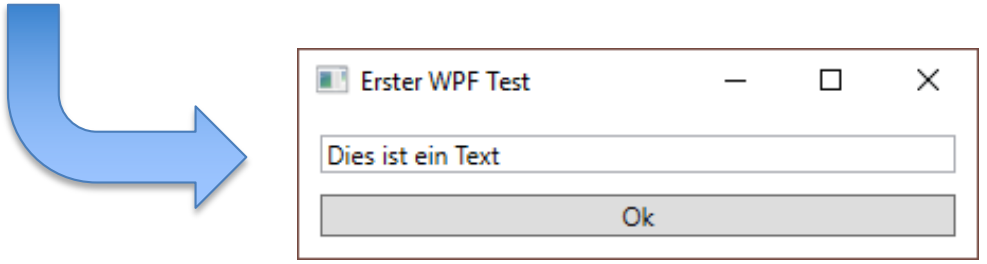
- Eine Oberfläche kann in WPF in einer speziellen Beschreibungssprache namens **XAML (Extensible Application Markup Language)** erstellt werden.
  - XAML basiert aus XML und kann mit Html verglichen werden.
  - Mit XAML beschreibt man die Struktur und das Aussehen einer Benutzeroberfläche.
  - Mit Html beschreibt man die Struktur und das Aussehen einer Web-Seite.

- Dies dient der Trennung von Aussehen und Logik einer Benutzeroberfläche.
  - Das Aussehen wird in XAML beschrieben.
  - Die Logik mit C# im Programmcode.

XAML Beispiel

```
<Window x:Class="WpfApplication2.MainWindow"
  …

  Title="Erster WPF Test" SizeToContent="WidthAndHeight">
  <StackPanel>
    <TextBox Margin="10 10 10 10" MinWidth="300">Dies ist ein Text</TextBox>
    <Button Margin="10 0 10 10">Ok</Button>
  </StackPanel>
</Window>
```



Eigenschaften von XAML

- XAML basiert auf **XML**.
  - Daher erbt es alle Eigenschaften von XML.

- XAML ist textbasiert und beschreibt die hierarchische Struktur einer Benutzeroberfläche.
  - Zur Laufzeit des Programms werden zu jedem Element im XAML-Code passende Objekte aus der WPF-Bibliothek erzeugt.
  - XAML dient dazu, die Steuerelemente zu konfigurieren, sie in der Oberfläche auszurichten etc.

- Ein XAML-Dokument muss wohlgeformt sein.
  - Es hat genau einen Wurzel-Tag → Meist das Window-Element.
  - Start-Tags müssen durch ein End-Tag geschlossen werden.

Tags

- XAML-Code besteht aus einer Vielzahl von sog. **Tags** (engl. für Auszeichnung, Etikett).
  - z.B. <Button>
  - Tags repräsentieren meist ein in der GUI sichtbares Element, z.B. ein Button.

- Welche Tags erlaubt sind, wird in XAML-Dateien im Stammelement (z.B. Window) durch die Namespace-Zuordnungen (xmlns) definiert.

```
<Window x:Class="WpfApp1.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:WpfApp1"
    Title="MainWindow" Height="350" Width="525">
    …
</Window>
```

Tags

- XML/XAML-Tags treten meist paarweise als Start-/Endtag auf.
  - `<Button></Button>`
- Ein Start-Tag wird von den größer/kleiner-Zeichen umschlossen: <Tag>
  - Ein End-Tag wird durch den zusätzlichen Querstrich / angezeigt: </Tag>
  - Jeder Start-Tag muss durch einen End-Tag geschlossen werden.
- Es existiert auch ein sog. Empty-Tag, der ohne eine End-Tag gültig ist.
  - `<Button />`
- Tag-Paare umschließen einfachen Text oder weitere andere Tags.
  - `<StackPanel><Button>Ok</Button></StackPanel>`
- Dadurch entsteht eine Hierarchie/ein Baum von Elementen.
  - In XAML kann ein Button auch ein Textfeld beinhalten usw.

Attribute

- Ein XML/XAML-Attribut ist ein Merkmal bzw. Eigenschaft.
  - Attribute dürfen nur Start-Tags oder Empty-Tags hinzugefügt werden.
  - `<Button Content="Ok" />`
- In XAML sind auch die erlaubten Attribute bereits festgelegt.
  - Sie verändern z.B. das Aussehen oder die Position von Elementen in der Oberfläche.
- Ein Attribut weist einem Schlüssel einen Wert zu.
  - Der Wert des Attributes ist in Anführungszeichen zu setzen.
- Beispiele:
  - `<Button Margin="10" Background="Red">Ok</Button>`
  - `<TextBox Margin="10" BorderBrush="Green">Text</TextBox>`

Werte von Attributen

- Zur Laufzeit versucht die WPF den Wert eines Attributs in etwas zu übersetzen, was die zugehörige Klasse auch versteht.
  - Im vorigen Beispiel haben wir z.B. gesehen, wie die Hintergrundfarbe von manchen Elementen definiert werden kann:
  - `Background="Red"`

- Der Wert „Red“ wird zur Laufzeit automatisch in ein Objekt vom Typ „Color“ umgewandelt.
  - Solche Übersetzungen übernehmen Klassen vom Typ `IValueConverter`.
  - Solche Klassen können bei Bedarf auch selbst erstellt werden.

XAML und Code Behind

- Ein WPF-Projekt besteht nicht nur aus XAML-Dateien.
  - XAML-Dateien haben die Dateiendung .xaml.
  - Zu jeder XAML-Datei existiert eine *gleichnamige* Datei mit der Endung .cs.
  - Die Klasse in dieser Datei ist der sog. **Code-Behind**, der auf C# basiert.

- Die XAML-Datei gibt an, auf welche Code-Behind-Datei sie sich bezieht.
  - Die Eigenschaft „x:Class“ gibt den Namen der zugehörigen Code-Behind-Klasse an.
  - Dieses Paar von Dateien wird automatisch für jedes WPF-Fenster generiert.

```
<Window x:Class="WpfApp1.MainWindow“
  …
  Title="MainWindow" Height="350" Width="525">
    …
</Window>
```

Aufbau der Code-Behind-Klasse

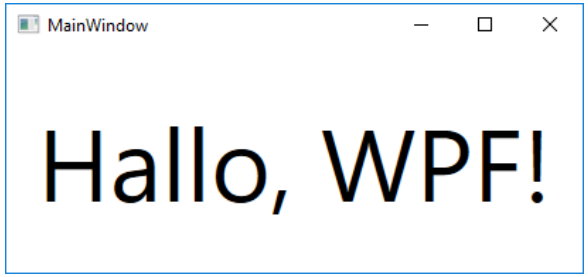
- Die Code-Behind-Klasse ist eine sog. partielle Klasse.
  - Dies ist eine Klasse, die aus mehreren Teilen zusammengesetzt ist.
- Der Rest der Klasse entsteht erst zur Laufzeit durch die Übersetzung der XAML-Datei in Programmcode.
  - Dieser Teil wird durch die Methode `InitializeComponent` aufgerufen.

```
public partial class MainWindow : Window
{
   public MainWindow()
   {
      InitializeComponent();
   }
}
```

- In der Code-Behind-Klasse kann auf alle Elemente des XAML zugegriffen werden.
  - Häufig wird hier auf Ereignisse von Elementen (z.B. Mouseclick) reagiert.
  - Entsprechend müssen dann dort passende Ereignis-Handler angelegt werden.

XAML vs. Code-Behind

- Viele Wege führen nach Rom.
  - Man kann häufig die selben Dinge in XAML oder in der Code-Behind-Klasse mit Hilfe von Programmcode erledigen.
  - Es ist aber meist wesentlich einfacher, die GUI in XAML aufzubauen.



```xaml
<Grid x:Name="Grid">
  <TextBlock
      VerticalAlignment="Center"
      HorizontalAlignment="Center"
      FontSize="72">Hallo, WPF!</TextBlock>
</Grid>
```

```code
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();

        TextBlock b = new TextBlock();
        b.Text = "Hallo, WPF!";
        b.VerticalAlignment = VerticalAlignment.Center;
        b.HorizontalAlignment = HorizontalAlignment.Center;
        b.FontSize = 72;
        Grid.Children.Add(b);
    }
}
```

Weitere Besonderheiten der WPF

- WPF nutzt einige Besonderheiten, die wir uns zunächst ansehen müssen.

- Die sog. **Routed Events** bezeichnen bestimmte Ereignisse in WPF.
  - Dadurch wird auch bei ineinander geschachtelten Elementen dafür gesorgt, dass Ereignisse an die richtigen Abnehmer weitergeleitet werden.

- Die sog. **Abhängigkeitseigenschaften (engl. dependency properties)** ermöglichen erweiterte Scenarios, um WPF-Elemente mit Eigenschaftswerten auszustatten.
  - Dadurch können Eigenschaftswerte vererbt, durch Styles, Animationen oder durch Datenbindung erzeugt werden.

Events in WPF

- Nehmen wir an, wir hätten einen Button in unserer Oberfläche.
  - Die zugehörige Klasse bietet den Event „Click“ an.
  - An diesen Event können wir in XAML einen passenden Event-Handler anhängen, der im Code-Behind erwartet wird.
  - Das Visual Studio hilft uns beim generieren des Event-Handlers durch zweimaliges Drücken der Tab-Taste.

```
<Button Click="Button_Click">Ok</Button>
```

- Im Code-Behind können wir nun beliebigen Quellcode hinterlegen:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Der Button wurde geklickt!");
}
```

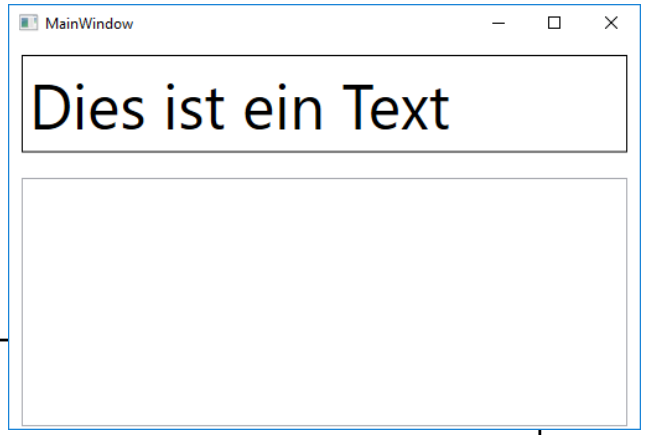
Routed Events

- Diese Art von Events unterscheiden sich nicht von denen, die wir zuletzt kennen gelernt haben.
  - In WPF werden diese als **direct events** bezeichnet.
  - Das Event wird auf dem Button ausgelöst und dort auch verarbeitet.

- Mit Hilfe von XAML können Elemente in der Benutzeroberfläche aber auch ineinander geschachtelt werden.
  - z.B. soll ein Button das Click-Ereignis erhalten, obwohl ein in dem Button angeordnetes Textfeld angeklickt wurde.

- Dazu existieren zwei weitere Arten von Events:
  - **Tunneling-Events** werden vom Wurzelelement bis zum auslösenden Element weitergereicht (Top-Down).
  - **Bubbling-Events** werden in der Gegenrichtung vom auslösenden Element bis zum Wurzelelement weitergereicht (Bottom-Up).

Beispielanwendung



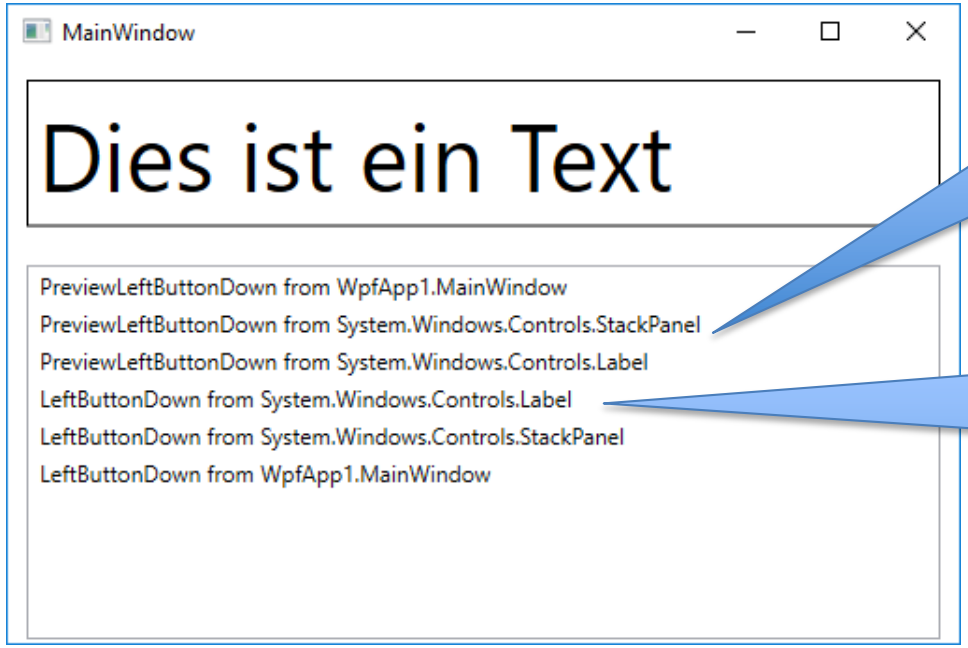
- Sehen wir uns den folgenden XAML-Code an:

```
<Window x:Class="WpfApp1.MainWindow">
   <StackPanel>
      <Label FontSize="50">Dies ist ein Text</Label>
      <ListBox x:Name="listbox" Margin="10" Height="200“ />
   </StackPanel>
</Window>
```

- Ein Label (Bezeichner) ist in ein StackPanel eingebettet.
  - Dieses ist wiederum Teil des Window-Elements.
- Alle drei Elemente können auf Ereignisse reagieren:
  - Bubbling-Events: z.B. MouseLeftButtonDown
  - Tunneling-Events: z.B. PreviewMouseLeftButtonDown
- Wir melden entsprechende Ereignis-Handler an alle Elemente an und erzeugen im zugehörigen Event-Handler eine Nachricht für die ListBox.
  - Wie sehen die Nachrichten aus?

Ergebnis

- Beim Klick auf das Label sieht das Ergebnis sieht wie folgt aus:



Die Tunneling-Events werden vom Window-Element bis hin zum Label aufgerufen.

Die Bubbling-Events steigen wieder vom Label bis zum Window auf.

Abhängigkeitseigenschaften

- **Abhängigkeitseigenschaften** (**engl. dependent properties**) sind in WPF eine Spezialform normaler Eigenschaften von Objekten, wie wir sie schon kennen.
  - Für die WPF sind die normalen Eigenschaften von Objekten mitunter nicht ausreichend.
  - Die WPF erlaubt an vielen Stellen, dass Eigenschaftswerte durch externe Quellen bestimmt werden und dass Eigenschaften Meta-Daten mitbringen.

- Beispiele:
  - Der Designer im Visual Studio weiß, welchen Wertebereich eine Eigenschaft annehmen darf.
  - Die Hintergrundfarbe aller Buttons wird durch einen Style zentral definiert.
  - Ein Button „erbt“ seine Größe durch die Größe seines Fensters.
  - Durch Datenbindung wird der Inhalt einer Textbox durch ein Datenmodell bestimmt.
  - Die Position eines Elements wird durch eine Animation dynamisch bestimmt.

- Diese Anforderungen können durch normale, sog. CLR-Properties nicht erfüllt werden.

CLR-Properties

- Denken wir zunächst zurück an unsere Klasse Bruch:

```
class Bruch
{
    private int zaehler;
    private int nenner;

    public int Zaehler
    {
        get { return zaehler; }
        set { zaehler = value; }
    }

    public int Nenner
    {
        get { return nenner; }
        set
        {
            if (nenner == 0)
                throw new ArgumentException("Der Nenner darf nicht 0 sein!");

            nenner = value;
        }
    }
}
```

Jedes Objekt der Klasse besitzt einen eigenen Satz von privaten zaehler und nenner Variablen.

Der Zugriff von Außen wird über Eigenschaftsmethoden realisiert. Solche Eigenschaften werden als CLR-Properties bezeichnet.

Klasse Bruch anpassen

- Wir wollen unsere Bruch Klasse nun so anpassen, dass für Zähler und Nenner die **dependency proeprties** der WPF verwendet werden.
  - Dazu muss die Klasse von der Klasse DependencyObject erben.

- Zudem werden die beiden Objektvariablen `zaehler` und `nenner` ersetzt.
  - Es werden zwei Klassenvariablen vom Typ DependencyProperty angelegt.
  - Diese können mit einigen Meta-Daten ausgestattet werden.

- Dadurch sorgen wir dafür, dass diese Eigenschaften auch in der WPF gut benutzbar sind.
  - Datenbindung, …

Geänderte Klasse Bruch

```
class Bruch : DependencyObject
{
    public static readonly DependencyProperty ZaehlerProperty =
       DependencyProperty.Register("Zaehler", typeof(Int32), typeof(Bruch));

    public static readonly DependencyProperty NennerProperty =
       DependencyProperty.Register("Nenner", typeof(Int32), typeof(Bruch));

    public Bruch(int zaehler, int nenner)
    {
        Zaehler = zaehler;
        Nenner = nenner;
    }

    public int Zaehler
    {
        get { return (int)GetValue(ZaehlerProperty); }
        set { SetValue(ZaehlerProperty, value); }
    }

    public int Nenner
    {
        get { return (int)GetValue(NennerProperty); }
        set { SetValue(NennerProperty, value); }
    }
}
```

Es werden zwei dependency properties registriert.

Die Werte werden nun in die dependency properties geschrieben und von da gelesen.

Abhängigkeitseigenschaften benutzen

- Die angepasste Klasse Bruch kann nun weiter so genutzt werden, wie zuvor:

```
Bruch b = new Bruch();
b.Zaehler = 1;
b.Nenner = 3;
```

- Die Eigenschaften lassen sich aber auch über die geerbten Methoden GetValue und SetValue nutzen:

```
b.SetValue(Bruch.ZaehlerProperty, 2);
int wert = (int)b.GetValue(Bruch.NennerProperty);
```

Fazit zu den Abhängigkeitseigenschaften

- Abhängigkeitseigenschaften werden nur selten selbst definiert.
  - So wie wir das soeben mit der Klasse Bruch gemacht haben.
- Sie werden aber sehr oft benutzt.
  - Fast alle WPF-Elemente nutzen dependency properties.
- Daher ist es sinnvoll zu verstehen, wie diese funktionieren.

Wir haben heute gelernt…

- Was die Windows Presentation Foundation genau ist.
- Wie man mit Hilfe von XAML Benutzeroberflächen definiert.
- Was der sog. Code Behind ist.
- Was Routed Events sind.
- Was Dependency Properties sind.