HOCHSCHULE HAMM-LIPPSTADT

Praktische Informatik

Vorlesung 08

Datenbindung

Zuletzt haben wir gelernt…

- Welche Shapes in WPF zum Zeichnen benutzt werden können.
- Wie man Bilder darstellt.
- Wie mit unterschiedlichen Pinsel-Arten die Darstellung von Linien und Füllungen gestaltet werden kann.
- Wie man Shapes dynamisch im Code Behind bewegen kann.
- Wie man damit z.B. das Computerspiel Pong realisieren kann.
- Wie man auf Elemente in WPF grafische Transformationen anwendet.

Inhalt heute

- Synchronisation von Daten
- Element-Bindung
- Die Klasse Binding
- Bindungsmodus und Converter
- Datenbindung
- INotifyPropertyChanged
- MVC und MVVM

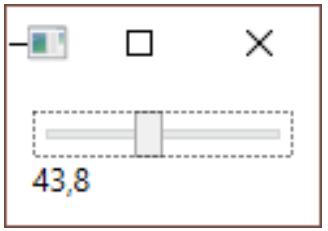
Synchronisation von Daten

- Bei der Anwendungsentwicklung steht man oft vor der Herausforderung, dass Daten von einer Quelle an einer anderen Stelle angezeigt werden müssen.
  - z.B. muss der Vorname aus einer Eigenschaft eines Objektes in ein Eingabefeld geschrieben werden und die Änderungen anschließend wieder zurückgeschrieben werden.

- Diese **Synchronisation** von Daten ist mitunter komplex.
  - Es muss sehr viel Programmcode dafür geschrieben werden.

- In moderneren Frameworks wie WPF wird diese Aufgabe durch **Datenbindung (engl. data binding)** erledigt.
  - Da Framework hilft bei der Synchronisation der Daten und stellt dafür spezielle Sprachmittel zur Verfügung.

Beispiel



- Schauen wir uns ein einfaches Beispiel an.
  - Der Wert eines `Sliders` soll unterhalb durch ein `Label` angezeigt werden.
  - Ändert sich die Position des `Sliders`, soll auch der Wert automatisch angepasst werden.

- Dies können wir schon jetzt mit Hilfe von Code Behind lösen.
  - Wir müssen auf das Ereignis `ValueChanged` des Sliders reagieren.
  - Im Event Handler können wir dann den Text des `Labels` ändern.

```
<StackPanel Margin="10">
   <Slider x:Name="slider" Minimum="0" Maximum="100" ValueChanged="slider_ValueChanged"/>
   <TextBlock x:Name="label" />
</StackPanel>
```

```
private void slider_ValueChanged(object sender, RoutedPropertyChangedEventArgs<double> e)
{
    label.Text = e.NewValue.ToString();
}
```

Datenbindung

- In WPF kann diese Aufgabe sehr viel leichter mit Hilfe von **Datenbindung (engl. Data Binding)** gelöst werden.
  - Es muss dafür kein Programmcode geschrieben werden.
  - Die Datenbindung kann auch im XAML beschrieben werden.

- Datenbindung verbinden i.A. eine Datenquelle mit eine Ziel.
  - Im Ziel werden eine oder mehrere Eigenschaften mit der Quelle verbunden.
  - Das Ziel einer Datenbindung muss immer eine **Abhängigkeitseigenschaft** sein.

- Die Datenquelle kann ein Steuerelement sein.
  - Dies nennt man dann Element-Bindung.

- Die Datenquelle kann aber auch …
  - ein einfaches C#-Objekt,
  - eine Collection,
  - eine Datenbank, XML-Datei, … sein.

Element-Bindung definieren

- Wir wollen in unserem Beispiel den Wert des `Sliders` an den Text des `Labels` binden.
  - Wir weisen der Text-Eigenschaft des Labels keinen einfachen String, sondern ein `Binding`-Objekt zu.
  - Dies geschieht über geschweifte Klammern (**markup extension**).

```
<StackPanel Margin="10">
    <Slider x:Name="slider" Minimum="0" Maximum="100"/>
    <TextBlock Text="{Binding ElementName=slider, Path=Value}"/>
</StackPanel>
```

- Den Code Behind brauchen wir dann nicht mehr.
  - Das Binding aktualisiert den Text automatisch mit dem Wert des Sliders.

- Bei dieser Bindung müssen zwei Parameter definiert werden.
  - Der `ElementName` legt fest, aus welchem Element die Daten stammen.
  - Der `Path` legt fest, welche Eigenschaft des Elements die Daten bereitstellen.

Binding im Code Behind

- Ein solches Data Binding kann auch im Code Behind aufgebaut werden.
  - Dafür wird ein Objekt der Klasse Binding erzeugt.

- Das Ziel (hier das Label) erhält dieses Binding-Objekt zur Synchronisation der Daten:

```
Binding binding = new Binding();
binding.Path = "Value";
binding.ElementName = "slider";
label.SetBinding(Label.ContentProperty, binding);
```

Die Klasse Binding

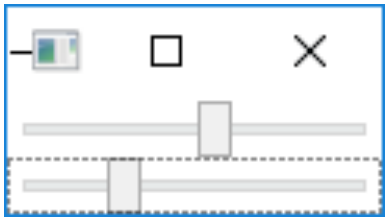
- Objekte vom Typ `Binding` dienen als Verbindung zwischen
  - Ziel-Objekten (normalerweise WPF-Elemente) und
  - Einer Datenquelle (z.B. eine Datenbank, eine XML-Datei oder ein beliebiges Objekt mit Daten).

Eigenschaften der Klasse Binding

- Die Eigenschaften der Klasse Binding ermöglichen eine Konfiguration der Datenbindung.

Eigenschaft	Bedeutung
ElementName	Gibt den Namen des Steuerelements (vom GUI) an, welches als Datenquelle dient.
Path	Path ist die Eigenschaft (Property) an welche die Daten gebunden werden (Value, Text, Content, …)
Mode	Definiert den Bindungsmodus (Aktualisierungsmodus). Der Modus kann z.B. einseitig (OneWay) oder beidseitig (TwoWay) zwischen GUI und Datenobjekt sein. Standardeinstellung ist TwoWay.
Converter	Gibt das Objekt an, welche als Converter (Übersetzer) verwendet werden soll.
Source	Legt das Objekt fest, welches als Quelle der Datenbindung dient.
UpdateSourceTrigger	Definiert, wann die Datenquelle aktualisiert werden soll (z.B. jedesmal wenn sich die die Daten ändern).

Bindungsmodus

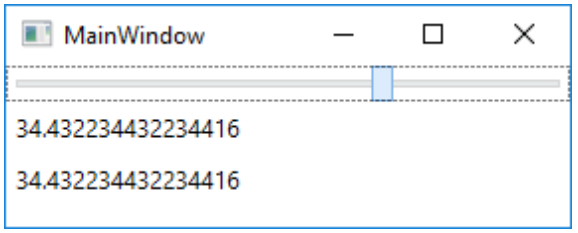


- Mit Hilfe der Eigenschaft Mode kann der Bindungsmodus festgelegt werden.
  - Als Beispiel sehen wir uns zwei Slider an, die miteinander synchronisiert werden sollen:

```
<StackPanel>
    <Slider x:Name="slider" />
    <Slider Value="{Binding ElementName=slider, Path=Value, Mode=OneWay}"/>
</StackPanel>
```

- Ohne die Mode-Angabe würde die Veränderung eines Sliders sofort auch den Wert des anderen Sliders verändern.
  - Die Angabe OneWay sorgt nun dafür, dass nur Änderungen an der Datenquelle auf das Ziel wirken.
  - Die Gegenrichtung wird nicht synchronisiert.

Converter



- Mit Hilfe der Eigenschaft `Converter` können wir ein zusätzliches Objekt definieren, welches den Wert der Quelle vor der Datenbindung transformiert.
  - Das Converter-Objekt muss dazu die Schnittstelle `IValueConverter` implementieren.

- Wir wollen diese Möglichkeit nutzen, um unseren Umrechner von Celsius nach Fahrenheit noch einmal neu zu gestalten.
  - Hierzu erstellen wir zunächst eine Anwendung mit einem Slider und zwei Labeln, die wir an den Slider binden:

```
<StackPanel>
    <Slider x:Name="slider" Minimum="-100" Maximum="100"/>
    <Label Content="{Binding ElementName=slider, Path=Value" />
    <Label Content="{Binding ElementName=slider, Path=Value" />
</StackPanel>
```

IValueConverter

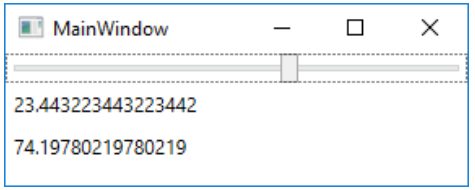
- Wir erstellen nun in unserem Projekt eine neue Klasse `CelsiusToFahrenheitConverter`.
  - Diese implementiert die Schnittstelle `IValueConverter`.

- Die beiden Methoden `Convert` und `ConvertBack` dienen der Transformation der Werte.
  - Hier müssen wir die Formeln für die Umrechnung einsetzen:

```
public class CelsiusToFahrenheitConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return System.Convert.ToDouble(value) * 1.8 + 32;
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return (System.Convert.ToDouble(value) - 32) / 1.8;
    }
}
```

Nutzung des Converters



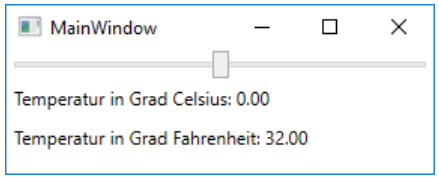
- Im XAML können wir den neuen Converter leider nicht sofort benutzen.
  - Es muss erst ein Objekt der Klasse erzeugt werden.
  - Ein solches Objekt kann für das Fenster als globale Ressource abgelegt werden.

```
<Window.Resources>
    <local:CelsiusToFahrenheitConverter x:Key="converter" />
</Window.Resources>
```

- Erst jetzt kann der Converter auch in der Datenbindung benutzt werden.
  - Dabei wird über den vergebenen Schlüssel auf die vorher erstellte Ressource zurückgegriffen:

```
<StackPanel>
    <Slider x:Name="slider" Minimum="-100" Maximum="100"/>
    <Label Content="{Binding ElementName=slider, Path=Value}" />
    <Label Content="{Binding ElementName=slider, Path=Value,
     Converter={StaticResource converter}}" />
</StackPanel>
```

Text im Label formatieren

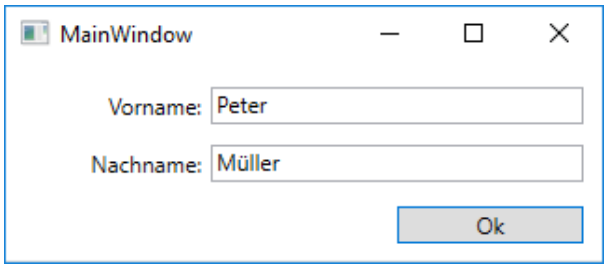


- Die Umwandlung funktioniert nun schon perfekt.
  - Allerdings sind wir mit der Formatierung der Daten in den Labels nicht ganz zufrieden.
  - Die Daten sollten besser nur 2 Nachkommastellen aufweisen.

- Mit Hilfe der Eigenschaft `ContentStringFormat` können wir beim Label die Ausgabe formatieren.

```
<StackPanel>
    <Slider x:Name="slider" Minimum="-100" Maximum="100"/>
    <Label Content="{Binding ElementName=slider, Path=Value}"
       ContentStringFormat="Temperatur in Grad Celsius: {0:F2}" />
    <Label Content="{Binding ElementName=slider, Path=Value,
       Converter={StaticResource converter}}"
       ContentStringFormat="Temperatur in Grad Fahrenheit {0:F2}"/>
</StackPanel>
```

Datenbindung



- Die letzten Beispiele haben Bindungen zwischen Elementen in einer GUI aufgebaut.
  - Häufig will man aber die Interaktionselemente einer GUI mit Daten aus dem Code Behind synchronisieren.
  - Auch das ist leicht möglich.

- Als Beispiel erstellen wir einen Dialog für die Bearbeitung von Vorname und Nachname eines Benutzers.
  - Dabei sollen die Textboxen automatisch mit einem Objekt aus dem Code Behind synchronisiert werden.

Datenmodell

- Wir erstellen zunächst eine einfache Klasse Person.
  - Dies stellt die beiden Eigenschaften Vorname und Nachname bereit:

```
public class Person
{
    private string vorname;
    private string nachname;

    public string Vorname
    {
        get { return vorname; }
        set { vorname = value; }
    }

    public string Nachname
    {
        get { return nachname; }
        set { nachname = value; }
    }
}
```

An die Eigenschaften Vorname und Nachname wollen wir später die Textboxen unseres Formulars binden.

Bereitstellung der Daten

- Damit die Datenbindung hegestellt werden kann, muss das Objekt zwingend über eine **Eigenschaftsmethode** zur Verfügung gestellt werden:

```
public partial class MainWindow : Window
{
    private Person person;

    public Person CurrentPerson
    {
        get { return person; }
    }

    public MainWindow()
    {
        person = new Person() { Vorname = "Peter", Nachname = "Müller" };
        InitializeComponent();
    }
}
```

Achtung: Die Daten müssen bereitstehen, bevor die Methode InitializeComponent aufgerufen wird.

XAML des Benutzerdialogs

- Das Formular in XAML ist sehr ähnlich zu unserem Beispiel Benutzeranmeldung der Vorlesung zum Thema Layouts.
  - Daher verzichten wir an dieser Stelle auf die Definition des Layouts.

- Damit die Datenbindung funktioniert, müssen wir dem Window-Element mit Hilfe von x:Name einen Namen geben.
  - Diesen benutzen wir dann in der Datenbindung.

```
<TextBox Text="{Binding ElementName=window, Path=CurrentPerson.Vorname}" />
<TextBox Text="{Binding ElementName=window, Path=CurrentPerson.Nachname}"/>
```

- Die Daten aus dem Objekt werden nun wie gewünscht in die Textboxen synchronisiert.
  - Dies werden wir später noch verifizieren.

DataContext

- Jede Klasse, die vom FrameworkElement ableitet, besitzt die Eigenschaft **`DataContext`**.
  - Auch die Klasse `Window` besitzt also diese Eigenschaft.
  - Der DataContext kann auf ein Objekt gesetzt, so dass sich eine Datenbindung darauf beziehen kann.

- Im Code Behind können wir daher unser Objekt an diese Eigenschaft der Window-Klasse setzen.
  - Dadurch verschlankt sich der Quellcode deutlich.

```
public partial class MainWindow : Window
{
    private Person person = new Person() { Vorname = "Peter", Nachname = "Müller" };

    public MainWindow()
    {
        DataContext = person;
        InitializeComponent();
    }
}
```

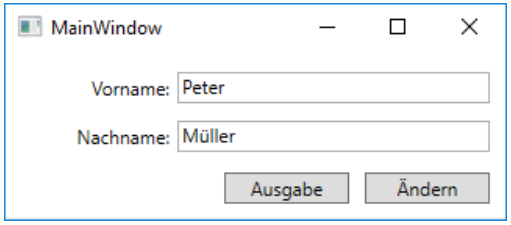
Datenbindung anpassen

- Durch den `DataContext` kann auch die Definition des Binding im XAML verschlankt werden.

```
<TextBox Text="{Binding Path=Vorname}" />
<TextBox Text="{Binding Path=Nachname}"/>
```

- Die Definition des `ElementName` kann entfallen.
  - Entsprechend muss auch das Window-Element nicht mehr benannt werden.

- Zudem verweist der Path nun auf die Eigenschaft direkt .
  - Der DataContext bezieht sich bereits auf das Objekt vom Typ Person.



Synchronisation verifizieren

- Wir wollen nun noch sicherstellen, dass die Synchronisation der Daten in beide Richtungen auch richtig funktioniert.
  - Dazu erstellen wir zwei Buttons im XAML und verbinden diese mit Event Handlern im Code Behind.

- Der erste Button „Ausgabe“ soll die Daten aus dem Objekt in einer MessageBox anzeigen.
  - Dies hilft, um zu prüfen, ob die Änderungen aus dem Formular auch im Objekt ankommen.

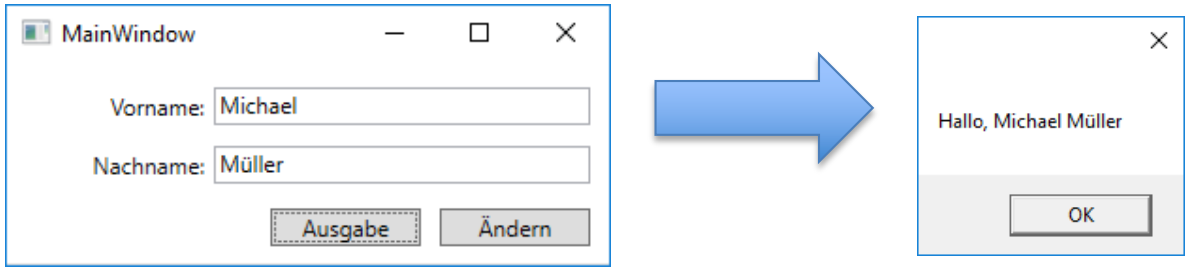
- Der zweite Button „Ändern“ soll die Daten im Code Behind verändern.
  - Dadurch sollte auch der Inhalt der Textboxen automatisch angepasst werden.

Ausgabe der Daten

- Im Event Handler des Ausgeben-Buttons erstellen wir eine MessageBox mit dem aktuellen Inhalt des Person-Objektes.

```
private void Ausgabe(object sender, RoutedEventArgs e)
{
    var s = String.Format("Hallo, {0} {1}", person.Vorname, person.Nachname);
    MessageBox.Show(s);
}
```

- Ändern wir den Inhalt der Textboxen und klicken dann den Button, sehen wir, dass die Daten offenbar auch im Person-Objekt geändert wurden.
  - Diese Richtung der Synchronisation funktioniert also.



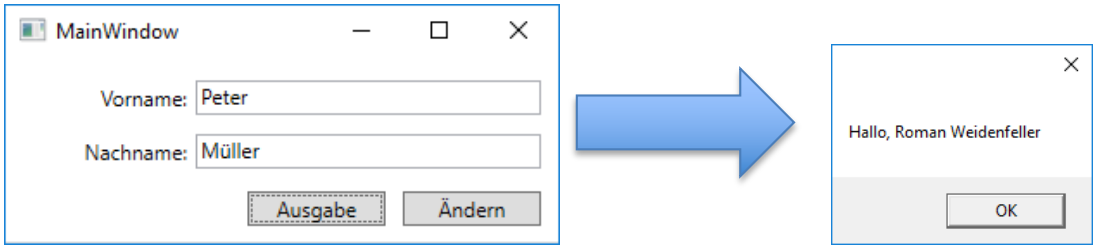
Ändern der Daten

- Im Event Handler des zweiten Buttons wollen wir die Daten des Objektes ändern.
  - Dadurch sollte sich auch der Inhalt der Textboxen ändern.

```
private void Aendern(object sender, RoutedEventArgs e)
{
    person.Vorname = "Roman";
    person.Nachname = "Weidenfeller";
}
```

- Klicken wir nun den Ändern-Button, passiert nichts.
  - Klicken wir anschließend den Ausgeben-Button, sehen wir die geänderten Daten.

- Die Änderungen sind offenbar nicht an die Textboxen weitergeleitet worden.
  - Warum ist das so?



INotifyPropertyChanged

- Objekte der Klasse Person sind normale, sog. CLR-Objekte.
  - Die Eigenschaften dieser Objekte haben keine eingebaute Möglichkeit, Beobachter über Änderungen an den Daten zu informieren.

- Dies kann aber nachgerüstet werden.
  - Dazu muss die Klasse Person die Schnittstelle **`INotifyPropertyChanged`** implementieren.
  - Die Klasse bekommt dadurch den Event **`PropertyChanged`**.

- Dieser Event muss genau dann geworfen werden, wenn sich Daten in dem Objekt ändern.
  - Dies geschieht in dem Set-Teil der Eigenschaftsmethoden.

Hilfsmethode

- Es bietet sich an, eine Hilfs-Methode **`NotifyPropertyChanged`** einzuführen.
  - Diese übernimmt das Werfen des Events.

```
private void NotifyPropertyChanged([CallerMemberName] String propertyName = "")
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
```

- Beim Werfen des Events muss der Name der betroffenen Eigenschaft übergeben werden.
  - Dies wird über den Parameter propertyName der neuen Methode erledigt.
  - Der Parameter muss nicht übergeben werden, sondern wird automatisch auf den Namen des Aufrufers gesetzt.

Setter anpassen

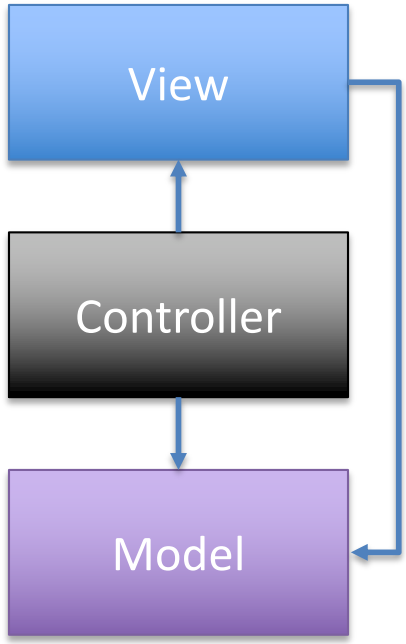
- In den Settern der Eigenschaftsmethoden muss nun noch die Hilfsmethode aufgerufen werden.

```
public string Vorname
{
   get { return vorname; }
   set
   {
      vorname = value;
      NotifyPropertyChanged();
   }
}
```

- Die gebundenen Interaktionselemente im Formular werden nun automatisch aktualisieren, wenn Daten geändert werden.
  - Die Synchronisation funktioniert damit in beide Richtungen wie gewünscht.

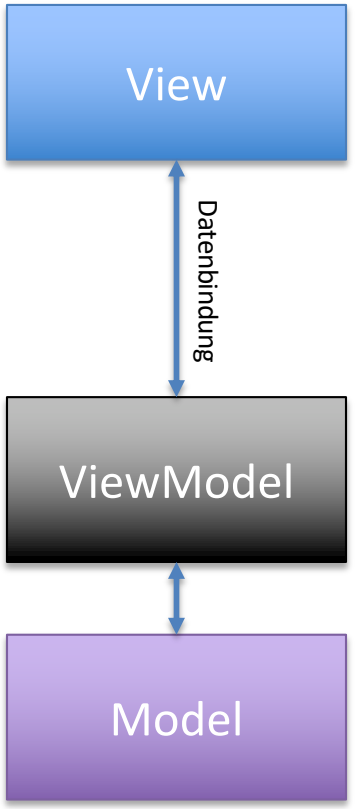
Model-View-Controller

- Anwendungen bestehen häufig aus mehreren Teilen mit unterschiedlichen Aufgaben.
  - Die unterschiedlichen Teile interagieren miteinander.
  - Muster helfen dabei, gutes Design zu erhalten und die Teile möglichst unabhängig voneinander zu konstruieren.
- Das **Model-View-Controller Prinzip (MVC)** zerlegt Anwendungen in genau drei Schichten.
  - Die Benutzeroberfläche wird als **View** bezeichnet.
  - Das Datenmodell (z.B. unsere Klasse Person) ist das **Model**.
  - Programmcode, der auf Eingaben des Nutzers reagiert und Daten zwischen Model und View synchronisiert, wird als **Controller** bezeichnet.
- Die Controller beinhalten in dieser Aufteilung meist recht viel und komplizierten Programmcode.
  - Besonders dann, wenn gleich mehrere Views mit dem Datenmodell synchronisiert werden müssen.



Model-View-ViewModel

- Da wir in WPF über die Datenbindung verfügen, müssen wir kaum/keine Controller erstellen.
  - Im schlimmsten Fall passen Datenmodell und Oberfläche allerdings nicht gut zusammen.
- Dann müssen wir in einer weiteren Schicht zwischen Datenmodell und Oberfläche vermitteln.
  - Eine solche Schicht wird als **ViewModel** bezeichnet.
- In WPF wird entsprechend das sog. **Model-View-ViewModel Prinzip** eingesetzt.



Wir haben heute gelernt…

- Das man häufig Daten zwischen Teilen von Anwendungen synchronisieren muss.
- Wie in WPF mit Element-Bindung synchronisiert werden kann.
- Welche Eigenschaften die Klasse Binding besitzt.
- Welche Bindungsmodi existieren und wie Converter verwendet werden können.
- Wie man Datenbindung einsetzt.
- Warum in Datenmodellen, die angebunden werden sollen die Schnittstelle INotifyPropertyChanged benötigt wird.
- Was der Unterschied zwischen MVC und MVVM ist.