Softwareentwicklung

HTL Krems

Inhaltsverzeichnis

Kapitel 1

Grundlagen

1.1 Model View ViewModel - MVVM

Das MVVM-Modell trennt die Model-Schicht (datenverarbeitende Schicht) von der View-Schicht (UI) über die sogenannte ViewModel-Schicht.
Mit der VM-Zwischenschicht stellen wir sicher, dass alle Änderungen, die an den Daten vorgenommen werden, ein Event auslösen (`PropertyChanged`). Dadurch kann die UI darauf subscriben und sich somit bei Änderungen automatisch updaten ("DataBinding").

Der Aufbau einer ViewModel-Klasse ist relativ einfach zusammengefasst:

- Die VM-Klasse implementiert das `INotifyPropertyChanged`-Interface, welches das `PropertyChanged`-Event verlangt.
- Die VM-Klasse erhält eine Instanz der Model-Klasse, in welcher die eigentliche Datenverarbeitung stattfindet.
- Alle Properties der Model-Klasse, die für die View sichtbar sein sollen, werden im ViewModel ebenfalls angelegt. Diese werden in der Langschreibweise implementiert und schleifen sowohl den Getter als auch den Setter einfach auf die Property der Model-Klasse durch. Zusätzlich wird im Setter das `PropertyChanged`-Event aufgerufen.

1.1.1 Concrete ViewModel

Beispiel: StudentVM

```
class Student
{
    public int StudentId { get; set; }
```

```
    public string Name { get; set; }
}

class StudentVM : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private Student student;
    public StudentVM(Student student)
        { this.student = student; }

    public int StudentId
    {
        get => student.StudentId;
        set
        {
            student.StudentId = value;
            if (PropertyChanged != null)
                PropertyChanged(this, new
                    PropertyChangedEventArgs("StudentId"));
        }
    }
    public string Name
    {
        get => student.Name;
        set
        {
            student.Name = value;
            if (PropertyChanged != null)
                PropertyChanged(this, new
                    PropertyChangedEventArgs("Name"));
        }
    }
}
```

- `PropertyChanged` ist jenes Event, das bei jeder Änderung aufgerufen werden soll.
- Im Konstruktor wird die `Student`-Instanz mitgegeben, in der die eigentliche Datenverarbeitung stattfindet.
- Die `get`-Methoden werden einfach auf den `Student` weitergeleitet.
- In den `set`-Methoden wird zuerst der Wert geupdated und dann das Event aufgerufen.

- `if (PropertyChanged != null)` stellt sicher, dass nicht versucht wird, das Event aufzurufen, wenn niemand subscribed hat. (Würde eine Exception verursachen.)
- Der erste Parameter des `PropertyChanged`-Events ist der `sender` - sprich jene Klasse, in der eine Property verändert wurde. Hier kann immer `this` verwendet werden.
  Der zweite Parameter ist eine EventArgs-Instanz, welche als `string` mitbekommt, welche Property verändert wurde.

1.1.2 Abstract ViewModel

Der Aufruf des Events ist relativ umständlich und immer wieder sehr ähnlich. Außerdem ist eine Gemeinsamkeit aller ViewModel-Klassen, dass sie das `INotifyPropertyChanged`-Interface implementieren.
Folglich macht es Sinn, den ViewModel-Klassen eine gemeinsame abstrakte Basisklasse zu geben.

Beispiel: Abstract ViewModel

```
abstract class AViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected void CallPropertyChanged(string property)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new
               PropertyChangedEventArgs(property));
    }
}
```

Lässt man `StudentVM` jetzt von dieser abstrakten Klasse erben, so werden die Properties etwas einfacher zu implementieren.

```
class StudentVM : AViewModel
{
    private Student student;
    public StudentVM(Student student)
        { this.student = student; }

    public int StudentId
    {
        get => student.StudentId;
        set
        {
            student.StudentId = value;
```

```
            CallPropertyChanged("StudentId");
        }
    }
    public string Name
    {
        get => student.Name;
        set
        {
            student.Name = value;
            CallPropertyChanged("Name");
        }
    }
}
```

1.1.3 ViewModel für ganze Listen

Möchte man nun in der View auf eine ganze Liste zugreifen, so benötigt man nicht nur die ViewModel-Klasse für die einzelnen Instanzen, sondern auch für das gesamte Liste.
Der Aufbau einer solchen sieht etwas anders aus, da wir jetzt nicht mehr selbst ein Event zur Verfügung stellen, sondern die Daten als `ObservableCollection<...>` bereitstellen, welche sich automatisch darum kümmert, dass die entsprechenden Events zur richtigen Zeit geworfen werden.

Beispiel: Concrete ViewModel für ganze DbSets

```
class PersonsVM
{
    public ObservableCollection<PersonVM> People { get;
       private set; }

    public PersonsVM(List<Person> people)
    {
        this.People = new ObservableCollection<PersonVM>(
            people.Select(p => new PersonVM(p))
        );
    }
}
```

- Die Klasse muss nichts implementieren oder beerben.
- Die ViewModel-Klasse erhält jetzt eine ganze Liste an Instanzen.
- Im Konstuktor wird die einfache Liste (gleiches funktioniert mit jedem `IEnumerable<...>`) in eine `ObservableCollection<...>` konvertiert.

- Zunächst werden die `Person`-Instanzen mithilfe des `.Select(...)` aus der Punkt-Notitation des LINQ in `PersonVM`s konvertiert. Dies ist notwendig, da wenn wir ein einzelnes Element aus der Liste nehmen (z.B. um es bearbeiten zu können), dieses nach wie vor die benötigen Events zur Verfügung stehen.
  (Z.B. wird es dadurch möglich auf `.SelectedItem` eines `DataGrid`s zu binden, da nicht nur die ganze Liste observable ist, sondern auch die einzelnen Elemente die `INotifyPropertyChanged` zur Verfügung stellen.)
- Aus diesen VM-Instanzen wird dann eine `ObservableCollection<...>` erstellt.

Kapitel 2

DataBinding

In einfachen Windows Forms- bzw. WPF-Anwendungen ändern wir die angezeigten Daten, in dem eigene Befehle ausgeführt werden. Möchten wir beispielsweise den Text eines Labels in WPF updaten, so müssen wir jedes Mal die `<label>.Content`-Property gesetzt werden.
Das Problem liegt nun darin, dass - bei einer korrekten MVC-Trennung - der Controller (die Logik) in den Klassen Werte ermittelt und verändert, die View allerdings nie davon erfährt.

Die Lösung dafür ist das DataBinding. Wir haben bereits die MVVM-Klassen kennengelernt - diese bilden eine Schnittstelle, die jedes Mal ein `INotifyPropertyChanged`-Event auslöst, wenn eine Property der Logik-Klasse verändert wird. Mittels DataBinding können diese MVVM-Klassen nun mit WPF-Elementen verbunden werden. Diese subscriben automatisch sich auf das `INotifyPropertyChanged`-Event und updaten bei jeder Veränderung einer Property in der Logik die zugehörigen WPF-Elemente.

Für die folgenden Beispiele gehen wir davon aus, dass es eine Klasse `Person` mit den Properties `Name` und `Address` und die zugehörige `PersonVM` und `PersonsVM` bereits fertig implementiert zur Verfügung stehen.

2.1 DataContext

Jedes WPF-Window und -Element besitzt einen eigenen DataContext. Dieser hat an sich keine Funktion und bildet nur die Grundlage für die Bindings. Der DataContext kann sowohl im Code als auch im XAML gesetzt werden.

2.1.1 DataContext im XAML

Wird der DataContext im Code gesetzt, so muss die Klasse einen leeren Konstruktor enthalten, da WPF automatisch eine Instanz erstellen können muss.
Im folgenden Beispiel setzen wir den DataContext für ein ganzes `Window`.

Beispiel: DataContext im XAML - Window

```
<Window [...]
        xmlns:local="clr-namespace:MyNamespace"
        [...]>

    <Window.DataContext>
        <local:Person />
    </Window.DataContext>

     [...]
</Window>
```

- Der DataContext wird immer in einem `<[element name].DataContext>`-Tag definiert. (Hier: `<Window.DataContext>`)
- In diesem Tag wird die Klasse, welche die Context-Instanz bildet, definiert. (Hier: `<local:Person />`
  - `local` ist weiter oben im Dokument definiert:
    `xmlns:local="clr-namespace:MyNamespace"`
    Hiermit wird angegeben, in welchem `namespace` die Klasse liegt. In diesem Beispiel wird zunächst der `namespace`, falls bereits einer gesetzt sein sollte, gecleared (`clr-namespace`) und dann auf `MyNamespace` gesetzt.
  - Die Klasse `Person` liegt im `namespace` `MyNamespace` und kann daher mit `local:Person` ausgewählt werden.
- WPF instanziert automatisch die Klasse `Person` mit dem Default-Konstruktor und verwendet diese Instanz als DataContext.

Der DataContext kann jedoch nicht nur für das `Window` gesetzt werden, sondern auch für jedes andere WPF-Element.

Beispiel: DataContext im XAML - Label

```
<Label>
    <Label.DataContext>
        <local:Person />
    </Label.DataContext>
</Label>
```

- Statt `<Window.DataContext>` wird in diesem Beispiel dann `<Label.DataContext>` genommen.

2.1.2 DataContext im Code

Setzen wir den DataContext im Code, so ist es nicht notwendig, dass die Klasse einen leeren Konstruktor enthält, da wie hier die Instanz selbst erzeugen müssen.
Wieder kann der DataContext auf `Window` bzw. Element-Ebene definiert werden.

Beispiel: DataContext im Code

```
public MainWindow()
{
    InitializeComponent();

    //Window
    this.DataContext = new Person() { Name = "Window" };
    //Element
    lblName.DataContext = new Person() { Name = "Element" };
}
```

2.1.3 Auswertung des DataContexts

Da ein DataContext auf `Window` als auch Element-Ebene im XAML und im Code definiert werden kann, müssen wir uns überlegen, welche DataContext für die Anzeige genommen wird, wenn wir - im Extremfall - alle 4 setzen.
Grundsätzlich gilt:

- Element-Ebene: Ist der DataContext auf Element-Ebene gesetzt, so gewinnt dieser über den des `Windows`.

- Window-Ebene: Der DataContext des `Windows` wird nur genommen, wenn auf Element-Ebene keine gesetzt ist.

Außerdem gilt:

- Code: Da DataContext im Code überschreibt jenen des XAML.
- XAML: Der DataContext des XAML ist nur aktiv, wenn er nicht im Code überschrieben wurde.

Somit gilt insgesamt für die 4 Fälle folgende Hierarchie. (Der oberste DataContext, der von dieser Liste gesetzt ist, wird verwendet.)

- Element-DataContext
  - Code
  - XAML
- `Window`-DataContext
  - Code
  - XAML

2.2 PropertyBinding

Das PropertyBinding erfordert eine MVVM-Klasse und ein setzen das DataContexts. Ist das erledigt, ist die eigentliche Implementierung des Bindings sehr bequem:

Beispiel: Binden einer Property

Hierfür ersetzen wir den statischen Wert einer Eigenschaft im XAML

```
<Label Content="Max Mustermann" />
```

durch einen {`Binding`}-Ausdruck

```
<Label Content="{Binding Name}" />
```

- Hier wird nun statt des statischen Wertes "`Max Mustermann`" der `Name` der `Person` aus dem DataContext eingefügt.

2.3 ListBinding

Im einfachsten Fall binden die Liste auf ein `DataGrid`. Hier genügt es die `ItemsSource`-Property und den DataContext zu setzen.

Beispiel: ListBinding - DataGrid

```
<DataGrid ItemsSource="{Binding People}">
    <DataGrid.DataContext>
        <local:PersonsVM />
    </DataGrid.DataContext>
</DataGrid>
```

- Das `DataGrid` zeigt automatisch alle zur Verfügung stehenden Properties in Tabellenform an.
- Der DataContext kann natürlich auch im `Window` oder im Code gesetzt werden.

Deutlich mehr Möglichkeiten bietet jedoch die `ListBox`. Hier werden die Items nicht fix in Tabellenform angezeigt, sondern es kann das aussehen der Items eigens konfiguriert werden.

Beispiel: ListBinding - ListBox

```
<ListBox ItemsSource="{Binding People}">
    <ListBox.DataContext>
        <local:PersonsVM />
    </ListBox.DataContext>
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <Label Content="{Binding Name}" />
                <Label Content="{Binding Address}" />
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

- Grundsätzlich erfolgt die Konfiguration genau gleich wie beim `DataGrid`.
- Zusätzlich wird dann ein `ItemTemplate` definiert, welches festlegt, wie jede `Person` angezeigt werden soll.
  - Für unsere Zwecke reicht der Tag `<DataTemplate>` im `ItemTemplate` vollkommen aus; andere Möglichkeiten werden hier nicht behandelt.

- Im DataTemplate können wir dann die Elemente, die zum Anzeigen eines Items verwendet werden, definieren. Möchte man mehr als ein Element anzeigen lassen, so muss man diese in einem Container zusammenfasse (hier: `<StackPanel>`; auch möglich: `<Grid>`, `<Canvas>`, etc.).

2.4 Commands

Commands bieten uns nun die Möglichkeit, nicht nur Properties aus den Logik-Klassen an WPF zu binden, sondern auch Methoden-Aufrufe.
Hierfür ist es notwendig, eine Instanz vom Typ `ICommand` zur Verfügung zu stellen, welche den Aufruf an die Logik weiterleitet.

Für die folgenden Beispiele wird folgende Ergänzung für die `PersonsVM` angenommen:

```
public void AddPerson(Person p)
{
    People.Add(new PersonVM(p));
}
public void RemovePerson(Person p)
{
    People.Add(new PersonVM(p));
}
```

2.4.1 ICommand

Die primitive Herangehensweise wäre, für jeden Methodenaufruf eine eigene ICommand-Klasse zu programmieren. Hierfür lässt man ganz einfach eine Klasse das `ICommand`-Interface implementieren.

Beispiel: ICommand-Klasse

```
class PersonAddCommand : ICommand
{
    private PersonsVM personsVM;
    public PersonAddCommand(PersonsVM personsVM)
        { this.personsVM = personsVM; }

    public event EventHandler CanExecuteChanged;
    public bool CanExecute(object parameter)
    {
        return true;
    }

    public void Execute(object parameter)
    {
        personsVM.AddPerson(new Person());
    }
}
```

- Die Klasse erhält eine Instanz der `PersonsVM`. In diesem Fall programmieren wir also eine Command-Klasse für die gesamte Liste; das gleiche kann natürlich auch mit Instanzen der einzelnen VM-Klasse geschehen.
- Die Methode `CanExecute(...)` liefert dem Aufrufenden die Information, ob die `Execute(...)`-Methode überhaupt aufgerufen werden darf. In unserem Fall gibt es keinen Grund, dass dies nicht der Fall sein sollte, folglich geben wir immer `true` zurück.
- Das `event` `CanExecuteChanged` muss immer dann aufgerufen werden, wenn sich der Wert der `CanExecute(...)` ändert.
  Da dies bei uns nie der Fall ist, wird das `event` nie aufgerufen.
- Die Methode `Execute(...)` wird aufgerufen, wenn der Command ausgeführt werden soll. Als Parameter erhält die Methode ein `object`, welches beim Aufruf beliebig gesetzt werden kann. Beispielsweise könnte man einen `string` übergeben, welcher den Namen enthält:

```
public void Execute(object parameter)
{
    string personInfo = (string)parameter;
    personsVM.AddPerson(new Person()
        { Name = personInfo }
    );
}
```

vgl. Design Patterns: Command Pattern

Der Command kann nun in der `PersonsVM` zur Verfügung gestellt werden:

```
public ICommand PersonAddCommand
{
    get
    {
        return new PersonAddCommand(this);
    }
}
```

2.4.2 RelayCommand

Die Lösung, für jeden Methodenaufruf eine eigene Methode zu implementieren führt zu Unmengen an redundantem Code. Deshalb schreiben wir uns dafür einmal die `RelayCommand`-Klasse, welche alle Command-Klassen ersetzen wird.

```
public class RelayCommand : ICommand
{
    private Func<object, bool> canExecute;
    private Action<object> execute;

    public event EventHandler CanExecuteChanged;

    public RelayCommand(Action<object> execute,
        Func<object, bool> canExecute)
    {
        this.canExecute = canExecute;
        this.execute = execute;
    }

    public bool CanExecute(object parameter)
        => canExecute(parameter);

    public void Execute(object parameter)
        => execute(parameter);
}
```

➤ Die Details der `RelayCommand` sind für den Test nicht relevant, da wir entweder die konkrete Implementierung schreiben müssen oder die `RelayCommand`-Klasse gegeben bekommen.

Nun können wir die selbe Funktionalität in der `PersonsVM` zur Verfügung stellen wie mit der `ICommand`-Klasse, jedoch ohne für jeden Methodenaufruf eine eigene Klasse erstellen zu müssen:

```
public RelayCommand PersonAddCommand
{
    get
    {
        return new RelayCommand(
            o => AddPerson(new Person()), //Execute
            o => true //CanExecute
        );
    }
}
public RelayCommand PersonRemoveCommand
{
    get
    {
        return new RelayCommand(
            o => RemovePerson(new Person()), //Execute
            o => true //CanExecute
        );
    }
}
```

- Die `RelayCommand`-Klasse erwartet als ersten Parameter die Implementierung der `Execute(...)`-Methode, als zweiten Parameter die Implementierung der `CanExecute(...)`-Methode.

- Die Implementierung der `CanExecute(...)`- und der `Execute(...)`-Methoden(parameter) kann natürlich wieder beliebig einfach oder komplex sein.
  Beispielsweise kann wieder ein Parameter übergeben werden:

```
public RelayCommand PersonAddCommand
{
    get
    {
        return new RelayCommand(
            o =>
            {
                string personInfo = (string)o;
                this.AddPerson(new Person() {
                    Name = personInfo
                });
            },
            o => true
        );
    }
}
```

2.4.3 Binding eines Commands

Grundsätzlich kann ein Command des DataContexts wieder genauso bequem gebindet werden wie eine Property:

Beispiel: CommandBinding einfach

```
<Button Command="{Binding PersonAddCommand}" />
```

- Wird der `Button` geklickt, so wird der `PersonAddCommand` der DataContext-Klasse ausgeführt.

- In diesem Fall wird der `CanExecute(...)`- und der `Execute(...)`-Methode `null` als Parameter mitgegeben.

Hinweis: *Wenn man den DataContext in zwei Elementen setzt, dann wird für jedes der beiden Elemente eine eigene, unabhängige Instanz erstellt. Möchte man in mehreren Elementen auf den selben DataContext zugreifen, so macht es Sinn, den DataContext auf* `Window`*-Ebene zu setzen.*

Möchte man nun einen Parameter übergeben, so kann man dies über die `CommandParameter`-Property machen:

Beispiel: CommandBinding Parameter

```
<TextBox Name="txtName" />
<Button
  Command="{Binding PersonAddCommand}"
  CommandParameter="{Binding ElementName=txtName, Path=Text}"
/>
```

- Der `CommandParameter` wird der `CanExecute(...)`- und der `Execute(...)`-Methode als Parameter übergeben.
- Der `CommandParameter` ist mittels ElementBinding gebunden.
  - `ElementName`: Name des WPF-Elements, auf das gebunden wird.
  - `Path`: Name der Property des WPF-Elements, die angezeigt werden soll.