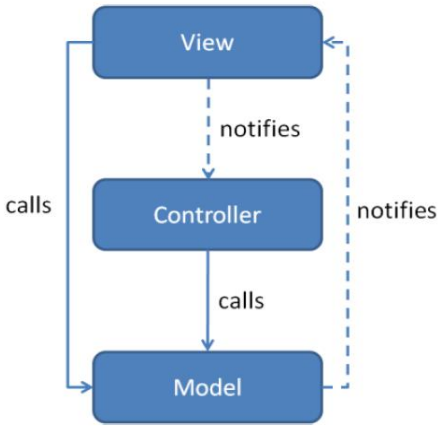
Einleitung: Model-View-ViewModel - MVVM

Übersicht

Das klingt ganz nach MVC (Model-View-Controller), und das nicht ohne Grund. Eine kleine Wiederholung vom MVC-Prinzip:

Es geht zunächst um eine klare Kompetenzverteilung für folgende Aufgaben:

- Datenansicht (View)
- Ablaufkoordination (Controller)
- Datenhaltung (Model)

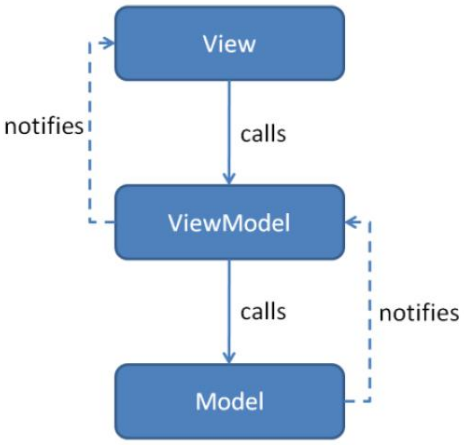


Darüber hinaus sind die drei Module als Layer (Ebene) organisiert:

Die View verständigt den Controller über eingegangene Ereignisse und reagiert mit einem Aufruf an das Model mit der gewünschten Datenänderung. Das Model verständigt die View, dass sich etwas an den Daten verändert hat. Anschließend werden von der View die neuen Daten angefordert.

Vorteil: Die Anwendungslogik ist von den dazugehörenden Darstellungen und den Benutzerinteraktionen klar getrennt

Nachteil: Es gibt keine klare Schichtentrennung, die View muss das Model und den Controller kennen.



Beim MVVM-Prinzip ist dieser Nachteil ausgeräumt und die View kennt nur das View-Model. Sie bezieht von dieser die Daten und wird verständigt, wenn sich Datenänderungen ergeben haben.

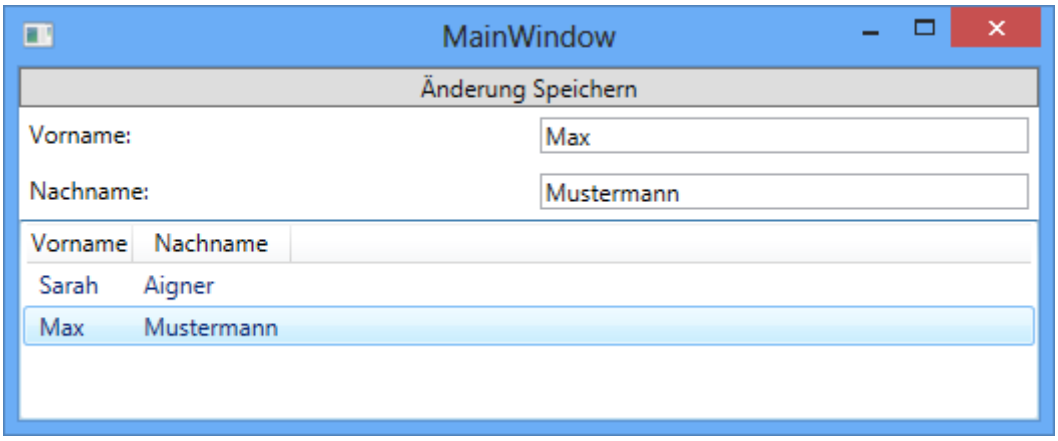
Ein weiterer **Vorteil**:

Die Kommunikation zwischen View und View-Model kann vollständig über Datenbindung ablaufen. Daher ist eine Code-Behind-Datei für die View nicht mehr vonnöten. Die View kann ausschließlich in XAML erstellt werden.

Damit die Datenbindung korrekt ablaufen kann, muss das View-Model entsprechende Properties für die View zur Verfügung stellen, welche die Daten für die Anzeige aufbereiten bzw. die Kommandos entgegennehmen. Diese Properties können leicht mit Unit-Tests auf Korrektheit geprüft werden.

Einleitendes Beispiel

Die Benutzerverwaltung des Activity Report Beispiels soll mittels MVVM realisiert werden.



Unten wird eine ListView mit allen Mitarbeitern angezeigt. Im oberen Bereich sind Eingabefelder, dort kann der jeweils unten angewählte Mitarbeiter bearbeitet werden. Wenn auf den Knopf „Änderungen Speichern“ gedrückt wird, so soll die Änderung in die Datenbank persistiert werden!

Schritt 1: Model

Das Model (in Form der Projekte Model und Repository) ist bereits in der Vorlage vorgegeben. Die Daten werden wiederum aus einer Datenbank bezogen (CodeFirst).

Schritt 2: ViewModel

Wir erstellen ein eigenes Projekt ActivityReport.ViewModel.

Schritt 2a: Erstellen eines Basis-View-Models (BaseViewModel)

Da ein View-Model immer seine View über Änderungen verständigt, lässt man jedes View-Model die Schnittstelle `INotifyPropertyChanged` implementieren. Eine abstrakte Basisklasse schafft hier Erleichterung:

```
public class BaseViewModel: INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanges(string propertyName)
    {
        if (PropertyChanged!=null)
        {
            PropertyChanged(this,new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Schritt 2b: Erstellen einer Basisklasse zur Kommandobearbeitung (RelayCommand)

Das View-Model wird von der View über Command-Binding verständigt, ob ein Event abzuarbeiten ist. Kommandos die über Binding mit XAML Komponenten verknüpft werden sollen, müssen die Schnittstelle ICommand implementieren.

ICommand definiert zwei Methoden und ein Event:

Methode CanExecute: Liefert true oder false, je nachdem ob ein Kommando gerade ausgeführt werden kann oder nicht. Wird das Kommando mit einem Button verknüpft, so wird der Button automatisch ausgegraut, wenn CanExecute „false“ liefert! Damit der Button eine Änderung des Zustandes mitbekommt, abonniert dieser das Event CanExecuteChange.

Methode Execute: Diese Methode wird aufgerufen, wenn das Kommando ausgeführt werden soll (z.B. der Button der mit dem Command verknüpft ist wird gedrückt).

Zur bequemen Erstellung dieser Kommandos ist eine Hilfsklasse günstig, welche ICommand implementiert. Damit wir unterschiedliche Kommandos mit dieser Basisklasse implementieren können, werden für die Inhalte von CanExecute und Execute Delegates verwendet. Dadurch kann der auszuführende Code an die Basisklasse übergeben werden (im Konstruktor).

HINWEIS: Die Referenz auf PresentationCore muss dem Projekt hinzugefügt werden, damit die Klasse CommandManager gefunden wird!

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Input;

namespace ActivityReport.ViewModel
{
    5 references
    class RelayCommand : ICommand
    {
        readonly Action<object> _execute;
        readonly Predicate<object> _canExecute;

        //Konstruktoren
        0 references
        public RelayCommand(Action<object> execute):this(execute,null)
        { }

        2 references
        public RelayCommand(Action<object> execute, Predicate<object> canExecute)
        {
            if (execute==null)
            {
                throw new ArgumentNullException("Execute darf nicht null sein!");
            }
            this._execute = execute;
            this._canExecute = canExecute;
        }

        public event EventHandler CanExecuteChanged
        {
            add  { CommandManager.RequerySuggested += value; }
            remove { CommandManager.RequerySuggested -= value; }
        }

        0 references
        public bool CanExecute(object parameter)
        {
            if (_canExecute == null)
                return true;
            else
                return _canExecute(parameter);
        }

        0 references
        public void Execute(object parameter)
        {
            _execute(parameter);
        }

        //Factory Methoden
        1 reference
        public static ICommand CreateCommand(ref ICommand cmd, Action<object> execute,
            Predicate<object> canExecute = null)
        {
            if (cmd==null)
            {
                cmd = new RelayCommand(execute, canExecute);
            }
            return cmd;
        }
    }
}
```

Schritt 2c: Erstellen eines View-Models für einen Entity-Typ

Das ist bei diesem Beispiel der Employee-Typ. Dabei sind zwei Aufgaben zu erledigen:

1. Felder/Daten der View:

   Zuerst werden alle darzustellenden Daten, welche an die View gebunden werden, in Form von Properties ab. Bei einer Datenänderung wird das `PropertyChanged`-Event ausgelöst. Zusätzlich muss die Auflistung `der` Mitarbeiter (Employees) eine ObservableCollection sein. Der Grund dafür ist, dass die geänderte Auflistung in einer Sammelanzeige (z.B. GridView) automatisch aktualisiert wird.

```
private string _firstName;     //Eingabefeld Vorname
private string _lastName;      //Eingabefeld Familienname
private Employee _selectedEmp; //Aktuell ausgewählter Mitarbeiter

private ObservableCollection<Employee> _employees; //Liste aller Mitarbeiter

1 reference
public string FirstName
{
    get  { return _firstName; }
    set
    {
            _firstName = value;
            OnPropertyChanged("FirstName");
    }
}
1 reference
public string LastName
{
    get  { return _lastName; }
    set
    {
            _lastName = value;
            OnPropertyChanged("LastName");
    }
}

public Employee SelectedEmp
{
    get { return _selectedEmp; }
    set
    {
        _selectedEmp = value;
        FirstName = _selectedEmp?.FirstName;
        LastName = _selectedEmp?.LastName;
        OnPropertyChanged("SelectedEmp");
    }
}
1 reference
public ObservableCollection<Employee> Employees
{
    get  { return _employees; }
    set
    {
        _employees = value;
        OnPropertyChanged("Employees");
    }
}
```

2. Mitarbeiterliste im Konstruktor laden:

```
1 reference
public EmployeeViewModel()
{
    LoadEmployees();
}

1 reference
private void LoadEmployees()
{
    using (UnitOfWork uow = new UnitOfWork())
    {
        var res = uow.EmployeeRepository.Get(
            orderBy: coll => coll.OrderBy(emp => emp.LastName)).ToList();
        Employees = new ObservableCollection<Employee>(res);
    }
}
```

3. Commands: Als nächstes wird für jede Aktion eine entsprechende *get-Property*, welches ein korrekt initialisiertes `Command`-Objekt zurückgibt, erstellt.

```
private ICommand _cmdSaveChanges;

0 references
public ICommand CmdSaveChanges
{
    get
    {
        return RelayCommand.CreateCommand(ref _cmdSaveChanges,
            p =>
            {
                using (UnitOfWork uow = new UnitOfWork())
                {
                    _selectedEmp.FirstName = _firstName;
                    _selectedEmp.LastName = _lastName;
                    uow.EmployeeRepository.Update(_selectedEmp);
                    uow.Save();
                    LoadEmployees();
                }
            },
        p =>
        {
            return _selectedEmp != null;
        });
    }
    set
    {
        _cmdSaveChanges = value;
    }
}
```

Damit ist die Funktionalität der verknüpften View festgelegt.

Diese Funktionalitäten könnte jetzt mit Unit-Tests überprüft werden, obwohl noch kein UI existiert.

Schritt 3: Erstellen der View

Zunächst erstellen wir ein neues WPF Projekt. (Die Connection Strings im App.config übernehmen wir dabei sicherheitshalber aus der App.config des Repository-Projektes!)

HINWEIS: EntityFramework muss per NuGet dem Projekt hinzugefügt werden! Um Versionskonflikte zu vermeiden. NuGet Package Manger auf Solution ausführen und dort auf InstalledPackages wechseln. Bei EntityFramework dann auf Mangage drücken und ViewModel Projekt hinzufügen!

```
<Window x:Class="ActivityReport.Wpf.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:vm="clr-namespace:ActivityReport.ViewModel;assembly=ActivityReport.ViewModel"
        Title="MainWindow" Height="350" Width="525">
 <!-- DataContext wird in CodeBehind gesetzt (um Laden der Datenbank im Designmode zu unterbinden!)
    <Window.DataContext>
        <vm:EmployeeViewModel></vm:EmployeeViewModel>
    </Window.DataContext> -->
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <ListView Grid.Row="1" ItemsSource="{Binding Employees}">
            <ListView.SelectedItem>
                <Binding Path="SelectedEmp"> </Binding>
            </ListView.SelectedItem>
            <ListView.View>
                <GridView>
                    <GridViewColumn Header="Vorname" DisplayMemberBinding="{Binding FirstName}"/>
                    <GridViewColumn Header="Nachname" DisplayMemberBinding="{Binding LastName}"/>
                </GridView>
            </ListView.View>
        </ListView>
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition></RowDefinition>
                <RowDefinition></RowDefinition>
                <RowDefinition></RowDefinition>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition></ColumnDefinition>
                <ColumnDefinition></ColumnDefinition>
            </Grid.ColumnDefinitions>
            <Button Grid.Row="0" Grid.ColumnSpan="2"
                    Command="{Binding CmdSaveChanges}">Änderung Speichern</Button>
            <TextBlock Margin="5,5,5,5" Grid.Row="1" Grid.Column="0">Vorname:</TextBlock>
            <TextBlock Margin="5,5,5,5" Grid.Row="2" Grid.Column="0">Nachname:</TextBlock>
            <TextBox Margin="5,5,5,5" Grid.Row="1" Grid.Column="1"
                     Text="{Binding Employee.FirstName, UpdateSourceTrigger=LostFocus}"></TextBox>
            <TextBox Margin="5,5,5,5" Grid.Row="2" Grid.Column="1"
                     Text="{Binding Employee.LastName, UpdateSourceTrigger=LostFocus}"></TextBox>
        </Grid>
    </Grid>
</Window>
```

Die Code-Behind-Datei bleibt (fast) völlig leer, die Verbindung zum View-Model wird ausschließlich über Datenbindung im XAML-Code hergestellt.

Die Verbindung zum ViewModel wird über den Datenkontext gesetzt. Dies ist das einzige das derzeit noch in der CodeBehind-Datei gesetzt wird.

Der Grund dafür ist, dass wenn der Kontext (wie oben auskommentiert) direkt im Xaml gesetzt wird, bereits im Designmodus (vor der Ausführung des Programms) eine Instanz des ViewModels erstellt wird. Da der Konstruktor jedoch bereits eine Datenbankverbindung aufbaut, ist dies nicht erwünscht. (Bei der Verwendung mehrerer Windows wird der Datenkontext dann ohnehin über einen Controller gesetzt, dazu später mehr)

Code-Behind-Datei:

```
/// <summary>
/// Interaction logic for MainWindow.xaml
/// </summary>
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        this.DataContext = new EmployeeViewModel();
    }
}
```

Erweiterung:

Erweitere das Beispiel so, dass der Button zum Speichern der Änderung nur dann aktiv ist, wenn mindestens drei Zeichen für den Nachnamen eingegeben sind!

Führe die Möglichkeit ein, über einen eigenen Button (unterhalt der Liste) einen neuen Mitarbeiter eingeben zu können (mit entsprechender Speicherfunktionalität!)

Ergänze ein Filterfeld, sodaß nur jene Mitarbeiter angezeigt werden, dessen Familienname mit den eingegebenen Zeichen beginnen (.StartsWith)