

Einleitung: Model-View-ViewModel - MVVM

Übersicht

Das klingt ganz nach MVC (Model-View-Controller), und das nicht ohne Grund.

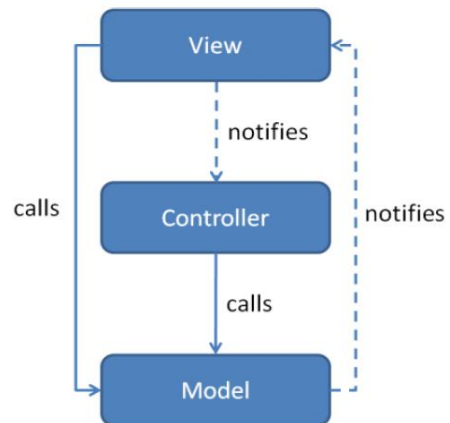
Wiederholung vom MVC-Prinzip:

Es geht zunächst um eine klare Kompetenzverteilung für folgende Aufgaben:

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

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

Die View verständigt den Controller über eingegangene Ereignisse (zB. Button-Klick). Der Controller setzt daraufhin erforderlichen Aktionen im Model. Das Model verständigt die View, wenn sich etwas an den Daten verändert hat (Observer Pattern). Anschließend werden von der View die neuen Daten entsprechend präsentiert (dazu können von der View natürlich auch Methoden vom Model aufgerufen werden).

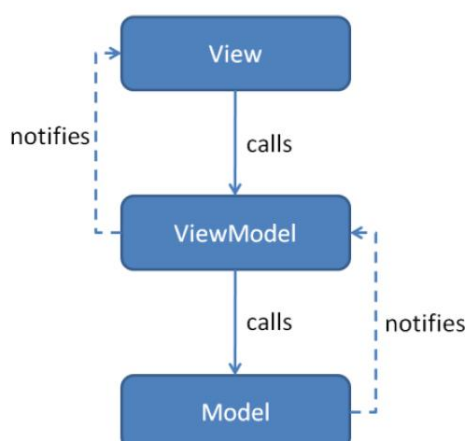


Vorteil: Die Anwendungslogik ist von den dazugehörenden Darstellungen und den Benutzerinteraktionen klar getrennt

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

Es sei angemerkt, dass es unterschiedliche Implementierungen von MVC gibt, sodass die obenstehende Darstellung nur eine Art von MVC ist. Alle Implementierung haben aber gemeinsam, dass das Model weder View noch Controller kennen muss und daher z.B. unabhängig getestet werden kann.

MVVM-Prinzip:



Beim MVVM-Prinzip ist der Nachteil der unzureichenden Schichtentrennung 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:**

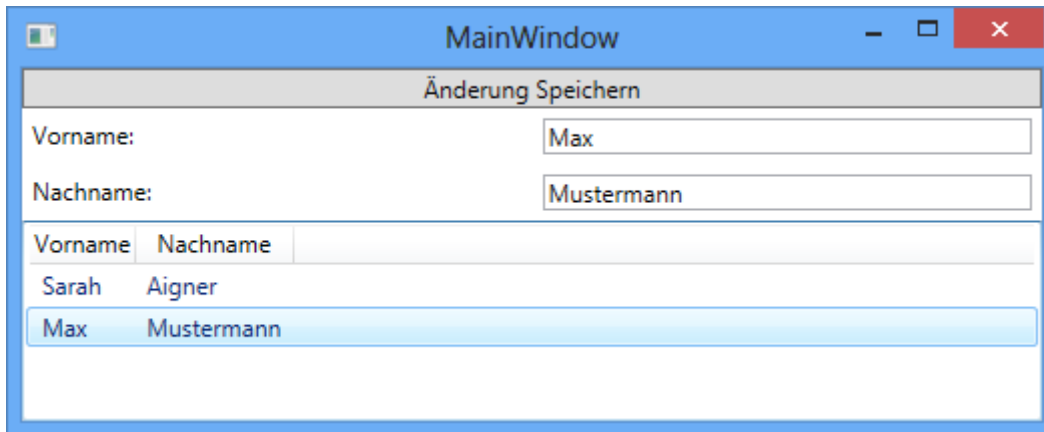
Die Kommunikation zwischen View und View-Model kann vollständig über Datenbindung ablaufen. Daher ist eine Code-Behind-Datei für die View nicht mehr

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!

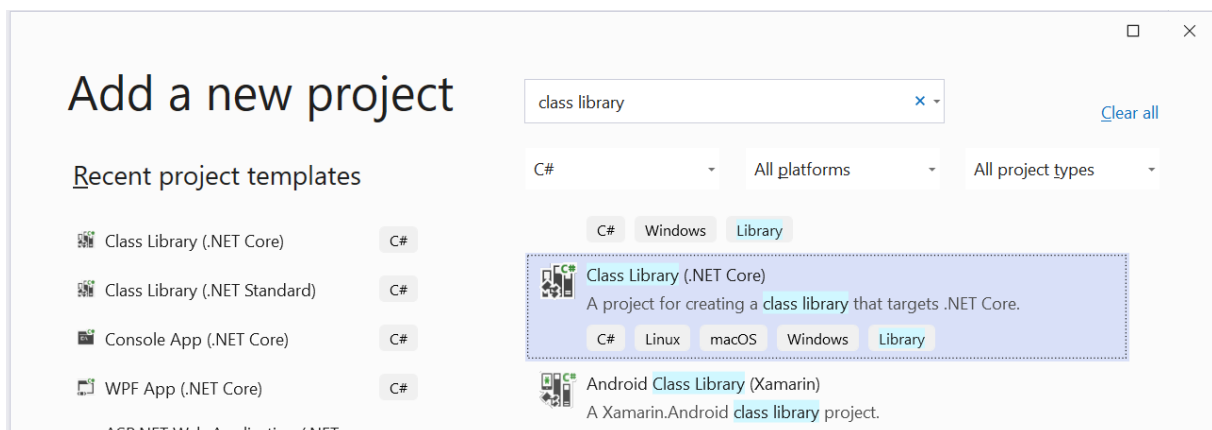
Schritt 1: Model

Der Data-Layer (in Form der Projekte Core und Persistence) ist bereits in der Vorlage vorgegeben. Die Daten werden wiederum aus einer Datenbank bezogen (CodeFirst).

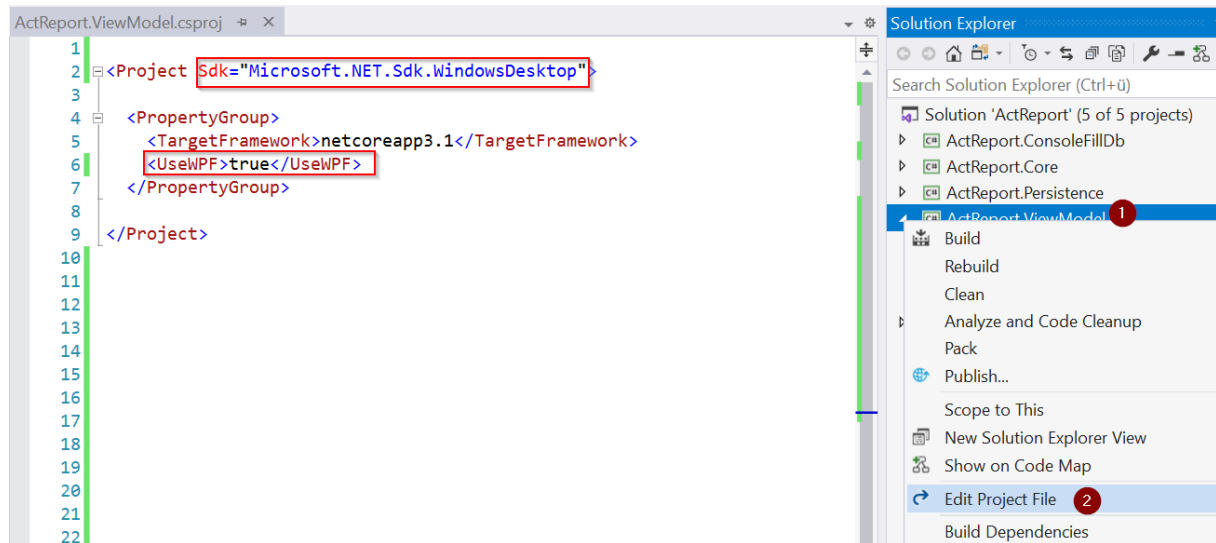
Hinweis : Der Data-Layer wurde mit dem Target Framework .NET Core implementiert.

Schritt 2: ViewModel

Wir ergänzen der Solution eine neue Class-Library ActReport.ViewModel.



Um sämtliche WPF-Funktionalität verwenden zu können, müssen in der ActReport.ViewModel.csproj folgende Änderungen vorgenommen werden:



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 abstract class BaseViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(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 CanExecuteChanged.

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!

```
public class RelayCommand : ICommand
{
    private readonly Action<object> _execute;
    private readonly Predicate<object> _canExecute;

    public RelayCommand(Action<object> execute, Predicate<object> canExecute)
    {
        _execute = execute;
        _canExecute = canExecute;
    }

    public bool CanExecute(object parameter)
        => _canExecute == null || _canExecute(parameter);

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

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

Schritt 2c: Erstellen eines View-Models für einen Entity-Typ („EmployeeViewModel.cs“)

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 abgebildet. 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.

```

public class EmployeeViewModel : BaseViewModel
{
    private string _firstName; // Eingabefeld Vorname
    private string _lastName; // Eingabefeld Nachname
    private Employee _selectedEmployee; // Aktuell ausgewählter Mitarbeiter
    private ObservableCollection<Employee> _employees; // Liste aller Mitarbeiter

    public string FirstName
    {
        get => _firstName;
        set
        {
            _firstName = value;
            OnPropertyChanged(nameof(FirstName));
        }
    }

    public string LastName
    {
        get => _lastName;
        set
        {
            _lastName = value;
            OnPropertyChanged(nameof(LastName));
        }
    }

    public Employee SelectedEmployee
    {
        get => _selectedEmployee;
        set
        {
            _selectedEmployee = value;
            FirstName = _selectedEmployee?.FirstName;
            LastName = _selectedEmployee?.LastName;
            OnPropertyChanged(nameof(SelectedEmployee));
        }
    }

    public ObservableCollection<Employee> Employees
    {
        get => _employees;
        set
        {
            _employees = value;
            OnPropertyChanged(nameof(Employees));
        }
    }
}

```

2. Mitarbeiterliste im Konstruktor laden:

```

public EmployeeViewModel()
{
    LoadEmployees();
}

private void LoadEmployees()
{
    using(UnitOfWork uow = new UnitOfWork())
    {
        var employees = uow.EmployeeRepository
            .Get(
                orderBy:
                    coll => coll.OrderBy(emp => emp.LastName))
            .ToList();

        Employees = new ObservableCollection<Employee>(employees);
    }
}

```

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

```
// Commands
private ICommand _cmdSaveChanges;
public ICommand CmdSaveChanges
{
    get
    {
        if (_cmdSaveChanges == null)
        {
            _cmdSaveChanges = new RelayCommand(
                execute: _ =>
                {
                    using IUnitOfWork uow = new UnitOfWork();
                    _selectedEmployee.FirstName = _firstName;
                    _selectedEmployee.LastName = _lastName;
                    uow.EmployeeRepository.Update(_selectedEmployee);
                    uow.Save();

                    LoadEmployees();
                },
                canExecute: _ => _selectedEmployee != null);
        }

        return _cmdSaveChanges;
    }
    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 3a: Erstellen eines WPF Projekts

Zunächst erstellen wir ein neues WPF Projekt („ActReport.UI“). (Den Connection String aus der Datei „appsettings.json“ übernehmen wir dabei aus dem „ConsoleFillDb“-Projektes!)

Hinweis: Folgende Packages müssen auch im WPF Projekt installiert werden, damit das Zusammenspiel mit Entity Framework funktioniert:

- ▲  NuGet
 - ▶  Microsoft.EntityFrameworkCore (2.0.0)
 - ▶  Microsoft.EntityFrameworkCore.SqlServer (2.0.0)
 - ▶  Microsoft.EntityFrameworkCore.Tools (2.0.0)
 - ▶  Microsoft.Extensions.Configuration.Json (2.0.0)

Schritt 3b: Erstellen der View

View:

```
<Window x:Class="ActReport.UI.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:ActReport.UI"
  mc:Ignorable="d"
  Title="ActivityReport" Height="350" Width="525">
  <!--
DataContext wird in CodeBehind gesetzt (um Laden der DB im Designmode zu unterbinden!)
  <Window.DataContext>
    <vm:EmployeeViewModel></vm:EmployeeViewModel>
  </Window.DataContext>
  -->
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"></RowDefinition>
      <RowDefinition Height="Auto"></RowDefinition>
      <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <Button Height="30" Command="{Binding CmdSaveChanges}">Änderung speichern</Button>
    <UniformGrid Grid.Row="1" Rows="2" Columns="2">
      <TextBlock>Vorname:</TextBlock>
      <TextBox Text="{Binding FirstName}"></TextBox>
      <TextBlock>Nachname:</TextBlock>
      <TextBox Text="{Binding LastName}"></TextBox>
    </UniformGrid>
    <ListView SelectedItem="{Binding SelectedEmployee}" Grid.Row="2" ItemsSource="{Binding Employees}">
      <ListView.View>
        <GridView>
          <GridViewColumn Width="120" Header="Vorname" DisplayMemberBinding="{Binding FirstName}"></GridViewColumn>
          <GridViewColumn Width="120" Header="Nachname" DisplayMemberBinding="{Binding LastName}"></GridViewColumn>
        </GridView>
      </ListView.View>
    </ListView>
  </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:

1. Erweitere das Beispiel so, dass der Button zum Speichern der Änderung nur dann aktiv ist, wenn mindestens drei Zeichen für den Nachnamen eingegeben sind (Hinweis: <https://www.wpf-tutorial.com/data-binding/the-update-source-trigger-property/>)!
2. Führe die Möglichkeit ein, über einen eigenen Button (unterhalb der Liste) einen neuen Mitarbeiter eingeben zu können (mit entsprechender Speicherfunktionalität!)
3. Ergänze ein Filterfeld, sodass nur jene Mitarbeiter angezeigt werden, dessen Familienname mit den eingegebenen Zeichen beginnen (.StartsWith)