Das Architekturmuster Model-View-ViewModel

Lukas Jaeckle*, **Joachim Goll**, **Manfred Dausmann**

Fakultät Informationstechnik der Hochschule Esslingen - University of Applied Sciences

Wintersemester 2015/2016

Model-View-ViewModel (Kurzform: MVVM) ist ein Architekturmuster für interaktive Systeme. Als solches dient es Entwicklern als Vorlage, um grafische Oberflächen standardisiert und strukturiert zu implementieren. MVVM kann als eine Weiterentwicklung bewährter Architekturmuster wie das Model-View-Controller-Muster (Kurzform: MVC) betrachtet werden.

MVVM **entstand 2005** durch die Entwicklung des Windows Presentation Foundation Frameworks (Kurzform: WPF) und dem .NET Framework 3.0. Darüber hinaus ist MVVM fester Bestandteil von Silverlight, den neuen Universal Apps, aber auch von gegenwärtig beliebten Webframeworks wie KnockoutJS oder AngularJS. Ferner werden auch existierende Frameworks wie JavaFX durch MVVM erweitert (mvvmFX).

Wie ist MVVM aufgebaut? Wo liegen die Vorteile, aber auch, welcher Voraussetzungen bedarf es für MVVM? Was macht MVVM anders als das weitverbreitete MVC? Diese Fragen sollen nachfolgend beantwortet werden.

Aufbau von MVVM

Interaktive Systeme müssen mehrere Aufgaben erfüllen. Die Offensichtlichste ist die **Darstellung** einer Benutzeroberfläche. Benutzereingaben, die über diese Oberfläche getätigt werden, müssen des Weiteren entsprechend entgegengenommen und verarbeitet werden (**Eingabeverarbeitung**[1]). Neben diesen interaktiven Aufgaben muss eine Anbindung an das weitere System hergestellt werden. Dieses besteht zum einen aus einer **Datenhaltung** und zum anderen aus den systemweit gültigen **Verarbeitungsprozessen**[2]. Insbesondere bei verteilten Systemen werden die Verarbeitungsprozesse oftmals getrennt betrachtet, sodass die interaktiven Aufgaben auf Darstellung, Eingabeverarbeitung und einer reduzierten Datenhaltung für die Darstellung beschränkt werden können. Diese Aufgaben werden in MVVM auf klar abgegrenzte Komponenten verteilt (siehe Abbildung 1).

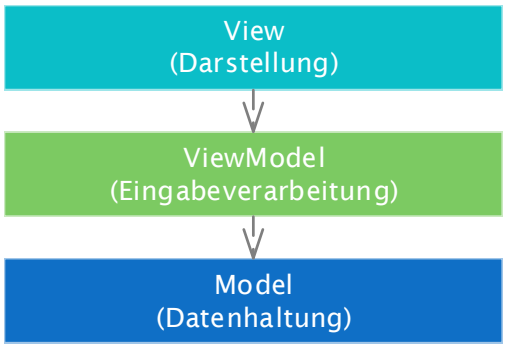

Abbildung 1: Komponenten und Abhängigkeiten

In der **View** wird die Benutzeroberfläche eines Systems festgelegt. In WPF wird dazu die deklarative Sprache XAML verwendet. Im XML-Format kann dadurch u. a. die Anordnung von Kontrollelementen spezifiziert werden. Dazu sind keine Programmierkenntnisse vonnöten, wodurch **Oberflächendesigner leichter** in die Entwicklung **eingebunden** werden können.

Das **ViewModel**, gerne auch als Model der View bezeichnet, beinhaltet Datenfelder[3] und die Eingabeverarbeitung[4]. Die Datenfelder des ViewModel können von der View an deren Kontrollelemente gebunden werden, sodass z. B. eine TextBox (**View**) die **Daten aus** einem Datenfeld des **ViewModel** anzeigt bzw. Benutzereingaben direkt in ein Datenfeld des ViewModel übernehmen kann. Ähnlich verhält es sich bei der Eingabeverarbeitung. So kann z. B. ein Button der View an eine Eingabeverarbeitung des ViewModel gebunden werden, sodass ein Klick automatisch die Eingabeverarbeitung ausführt. Soll dies temporär nicht möglich sein, kann das **ViewModel** den Zustand der Eingabeverarbeitung verändern und **die View** darüber **informieren**, sodass die View ihre Kontrollelemente entsprechend des Zustandes de- bzw. aktiviert.

*Diese Arbeit wurde durchgeführt bei der Firma IT-Designers, Esslingen-Zell

1 Presentation Logic
2 Business Logic
3 In WPF/C# durch Properties realisiert.
4 In WPF/C# durch Commands realisiert.

Wie das **Model** in MVC ist das Model in der ursprünglichen Definition des MVVM-Musters verantwortlich für die Geschäftslogik und die Datenhaltung [1]. In neueren Interpretationen besteht das Model nur aus **Entity-Objekten**. Die Geschäftslogik wird dabei in einer separaten Komponente implementiert. Letzteres bildet die Grundlage für eine gute Umsetzung von Separation-Of-Concerns.

MVVM in WPF

Wie geschildert, liegt der große Vorteil von MVVM in der **bidirektionalen Synchronisation** zwischen View und ViewModel. Diese Synchronisation muss jedoch durch das Framework unterstützt werden. WPF nutzt dafür das **Data Binding** und die **Änderungsbenachrichtigung** des .NET Frameworks. Beispielhaft ist dies nachfolgend dargestellt.

```
<Window x:Class="HelloWorld.View.HelloWorldV" ...>
  <Grid>  <TextBox Text="{Binding TextBox}" />  </Grid> </Window>

public class HelloWorldVm : INotifyPropertyChanged {
  public event PropertyChangedEventHandler PropertyChanged;
  protected virtual void OnPropertyChanged([CallerMemberName] string propertyName=""){
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
  }

  private string _textBox = "Hello World";
  public string TextBox {  get { return _textBox; }
                           set { _textBox = value; OnPropertyChanged(); }  }
}
```

Codebeispiel 1: Data Binding WPF

In der View wird im XML-Format angegeben, welches Kontrollelement welche Properties bzw. Commands des ViewModel nutzen soll (Data Binding). Obwohl sich die View dadurch abhängig von dem ViewModel macht, kann das Layout bzw. die **View nahezu unabhängig vom ViewModel entwickelt** werden.

Das ViewModel muss die benötigten Properties und Commands bereitstellen und eine Änderungsbenachrichtigung implementieren. WPF bietet diesbezüglich das Interface `INotifyPropertyChanged` an. Dieses Interface erfordert die Implementierung eines Events, welches bei Wertänderungen ausgelöst werden muss. Dies ist in dem Codebeispiel 1 angedeutet.

Der Vorteil dieser Lösung ist, dass das ViewModel unabhängig von der View bleibt. Dadurch kann das **ViewModel sehr gut getestet** werden. Zudem bietet das Eventhandling den Vorteil, dass jeweils nur **einzelne Elemente synchronisiert** werden. Der Overhead bleibt also gering.

MVVM im Vergleich zu MVC

Worin liegt folglich der Unterschied zu MVC[5]? In MVC holt sich die View über das Observable-Muster direkt von dem Model die Daten ab. Dadurch besitzt MVC eine zusätzliche **Abhängigkeit zwischen** der **View und** dem **Model**. MVVM verschiebt die Datenbereitstellung für die View in das ViewModel, wodurch diese Abhängigkeit **entfällt**.

Der zweite Unterschied liegt darin, dass das **ViewModel keine Abhängigkeit zur View** besitzt. In MVC besitzt der Controller eine Referenz auf die View, damit der Controller den Zustand von Kontrollelementen unmittelbar setzen kann. Dies erschwert das Erstellen von Tests allerdings deutlich. MVVM hingegen, ermöglicht durch die bidirektionale Synchronisation[6] zwischen der View und dem ViewModel, dass der Zustand der View im ViewModel verwaltet werden kann, ohne eine Referenz auf die View zu benötigen. Durch diese **lose Kopplung** ist die Testbarkeit verbessert.

Bewertung

MVVM in Kombination mit WPF bietet **Entwicklern und Oberflächendesignern** eine gute Ausgangsbasis, um interaktive Systeme **klar und strukturiert** zu entwickeln. Durch **wenige Abhängigkeiten** wird zudem eine **gute Wartbarkeit** der Komponenten ermöglicht.

[1] J. Gossman (08. Okt 2005). Introduction to Model/View/ViewModel pattern for building WPF apps: http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx, Zugriff am 7. Aug 2015.

[2] J. Goll (2014). Architektur- und Entwurfsmuster der Softwaretechnik, 2. Auflage, Springer Vieweg, pp. 377-390.

[5] Verglichen wird mit der Active-Model-Variante im Pull-Betrieb von MVC [2]

[6] Änderungsbenachrichtigung