Hochschule Esslingen
University of Applied Sciences

VON MVC ZU MODEL-VIEW-VIEWMODEL

Wissenschaftliche Vertiefung von Lukas Jaeckle

Studiengang Softwaretechnik und Medieninformatik

Agenda

1. Architekturmuster

2. Architekturmuster für interaktive Systeme
   - Model-View-Controller
   - Model-View-Presenter
   - Presentation Model

3. Model-View-ViewModel
   - Komponenten
   - Abläufe
   - Vergleich MVC & MVVM
   - Bewertung

- Beschreiben:
  - die **Zerlegung eines Systems** in Komponenten und
  - deren **Zusammenwirken**

- Dienen zur einfachen Klärung von komplexen Sachverhalten

- Beispiele:
  - Schichtenmodelle
  - Pipes and Filters
  - Plug-in
  - Model-View-Controller

Definition

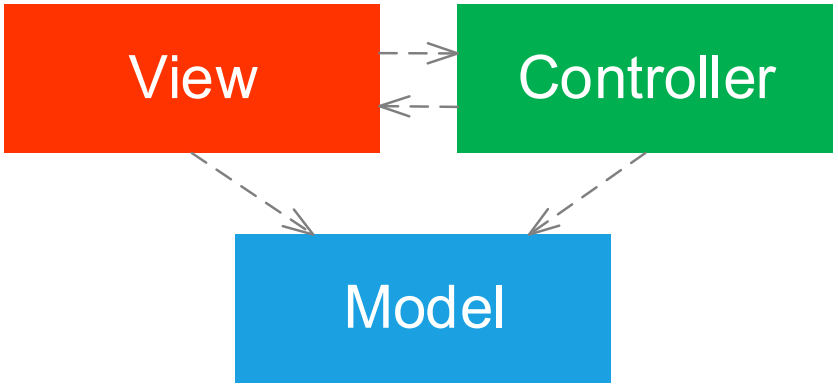
- Umsetzung von **Separation-of-Concerns** durch Trennung von:
  - Darstellung
  - Eingabeverarbeitung
  - Transiente Datenhaltung
  - Ferner: Verarbeitung von Geschäftsprozessen und Persistente Datenhaltung

- Unabhängige Entwicklung der Komponenten
- Parallelisierung der Entwicklung
- Vereinfachter Austausch des schnelllebigen MMI
- Wiederverwendbarkeit von Komponenten
- Einfachere Fehlerfindung

Angestrebte Ziele für die Muster zu interaktiven Systemen

- Trennung von:
  - Datenhaltung und Geschäftslogik (Model)
  - Darstellung (View) und
  - Benutzereingaben sowie Zustand der Darstellung (Controller)

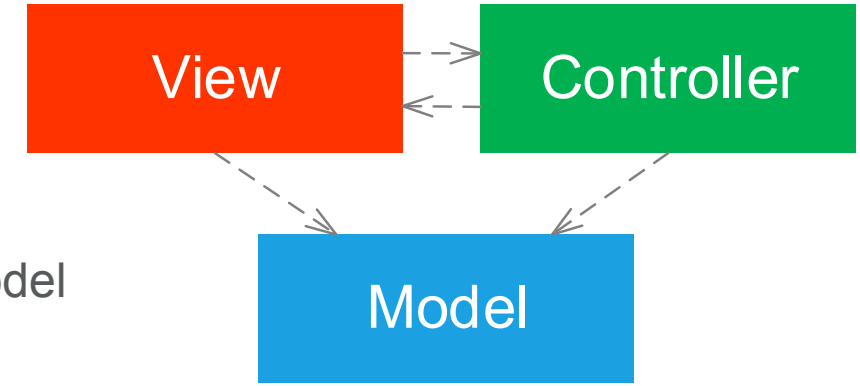
- Abhängigkeiten:
  - Die View übergibt Eingaben an den Controller
  - Der Controller steuert aufgrund der Eingaben:
    - Datenübergabe an das Model
    - Verarbeitungsprozesse des Model
  - Der Controller kann die Darstellung der View ändern
  - Die View stellt Daten des Model dar



Model-View-Controller (1979 Reenskaug)

- Heutige Bedeutung:
  - Erster Ansatz, das MMI austauschbar zu gestalten
  - Ermöglicht mehrere Darstellungen von denselben Daten
  - **Findet sich in vielen Techniken wieder**

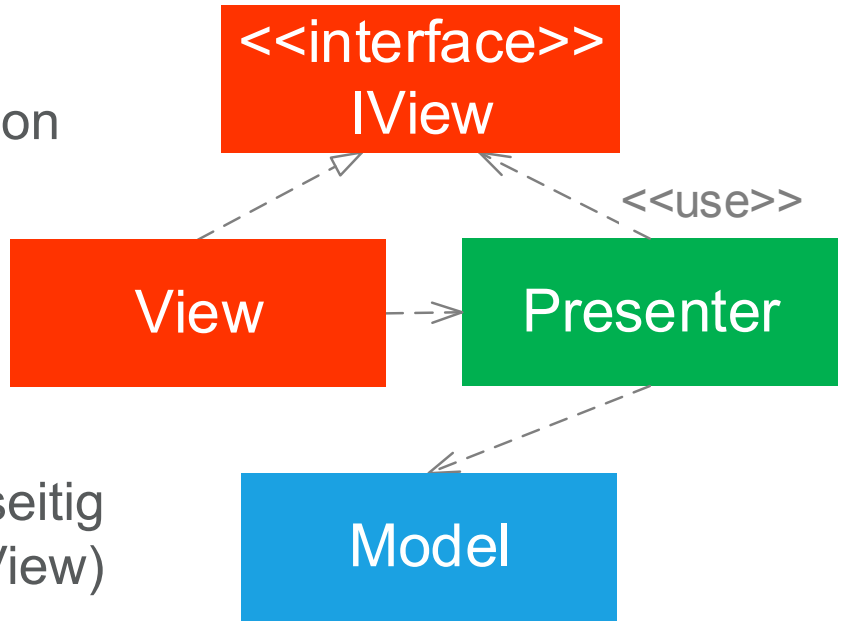
- Aber:
  - Gegenseitige Abhängigkeit zwischen View und Controller erschwert das Testen der Eingabeverarbeitung
  - Unterstützt Separation-of-Concerns nur eingeschränkt:
    - Synchronisation (Observable) & Konvertierung in der View
    - Geschäftslogik & Datenhaltung im Model



Model-View-Controller (1979 Reenskaug)

Verwandte Architekturmuster

- Presenter (ehem. Controller):
  - verliert Abhängigkeit zu **konkreter** View durch zusätzliches Interface
  - beinhaltet Eingabe-/Verarbeitungslogik (inkl. Konvertierung)
  - stellt Daten für View bereit
- **Vorteile**:
  - Testbarkeit des Presenters erhöht
  - Observable-Muster zur Synchronisation (inkl. Abhängigkeit) entfällt
  - View definiert nur die Darstellung
- **Nachteile**:
  - Zusätzliches Interface
  - Presenter und View weiterhin gegenseitig abhängig (wenn auch indirekt über IView)



MVP (90er, Def. Fowler) - Variante „Passive View“

Verwandte Architekturmuster

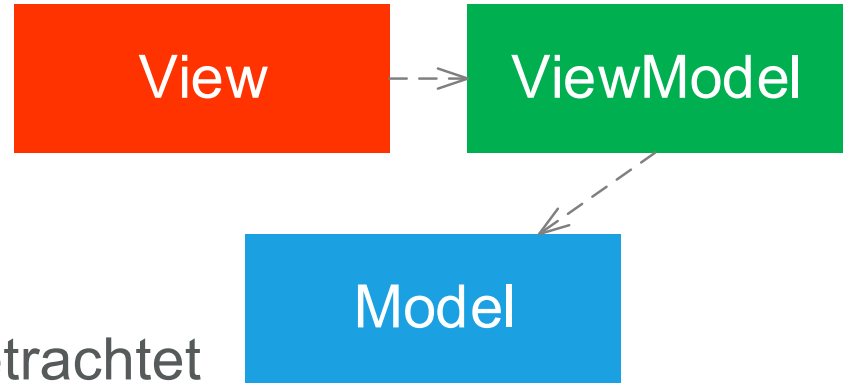
- Löst ggs. Abhängigkeit durch synchronisierte **Data Fields**
- Synchronisation erfolgt über Getter-/Setter-**Methoden**, die entsprechend **bei Eingaben** aufgerufen werden müssen.
- Ort der Synchronisation nach Bedarf:
  - Synchronisation im Presentation Model (ehem. Controller)
    - **„Schmale“ View und testfähige Synchronisation**
    - Abhängigkeit vom Presentation Model zur View bzw. IView
    - IView nur zum einfacheren Testen benötigt
  - ~~Alternative: Synchronisation in der View~~
    - ~~**IView entfällt**~~
    - ~~Abhängigkeit von der View zum Presentation Model~~
    - ~~Aber: View nicht nur Darstellung (Separation-Of-Concerns verletzt)~~
- Unschön: Boilterplate Code für Synchronisation benötigt

Presentation Model (2004 Fowler)

Model-View-ViewModel

- Historie:
  - 2005 mit Windows Presentation Foundation (WPF) entstanden
  - Erstmalig von John Gossman vorgestellt
  - Basiert auf dem Muster Presentation Model

- Aufteilung der Verantwortlichkeiten:
  - Transiente Datenhaltung (Model)
  - Darstellung (View)
  - Eingabelogik und Datenaufbereitung (ViewModel)
- Verarbeitungslogik wird getrennt betrachtet
- Trick: Synchronisation mithilfe .NET-Event-Mechanismus
  -> daher View nur Darstellung



Komponenten

```
<Window x:Class="HW.MainWindowView“ …>
  <TextBox Value="{Binding Text}" />
</Window>
```

```
public class MainWindowViewModel : INotifyPropertyChanged {
  public event PropertyChangedEventHandler PropertyChanged;
  private string _text = "Hello World!";
  public string Text {
    get{ return _text; }
    set{ _text = value; PropertyChanged?.Invoke(…); }
  }
}
```

Code-Beispiel – Data Bindung

```
<Window x:Class="HW.MainWindowView“ …>
  <TextBox Value="{Binding Text}" />
</Window>
```

```
[ImplementPropertyChanged]
public class MainWindowViewModel {
  public string Text { get; set; } = "Hello World!";
}
```

Code-Beispiel – Aspektorientierte Programmierung

- **Model**:
  - Zuständig für transiente Datenhaltung
  - Realisiert durch Entity-Klassen

- **View**:
  - Bestimmt Anordnung von Kontrollelementen (z. B. TextBox)
  - Bestimmt Animation von Kontrollelementen (z. B. Ausgrauen)
  - Bestimmt Darstellung der Daten
  - Bindet Kontrollelemente an Datenfelder bzw. an die Eingabeverarbeitungslogik des ViewModel
  - Aktualisiert Anzeige bei Änderungen

Model & View

- **ViewModel** hat „Vermittlerrolle“ zwischen View und Model

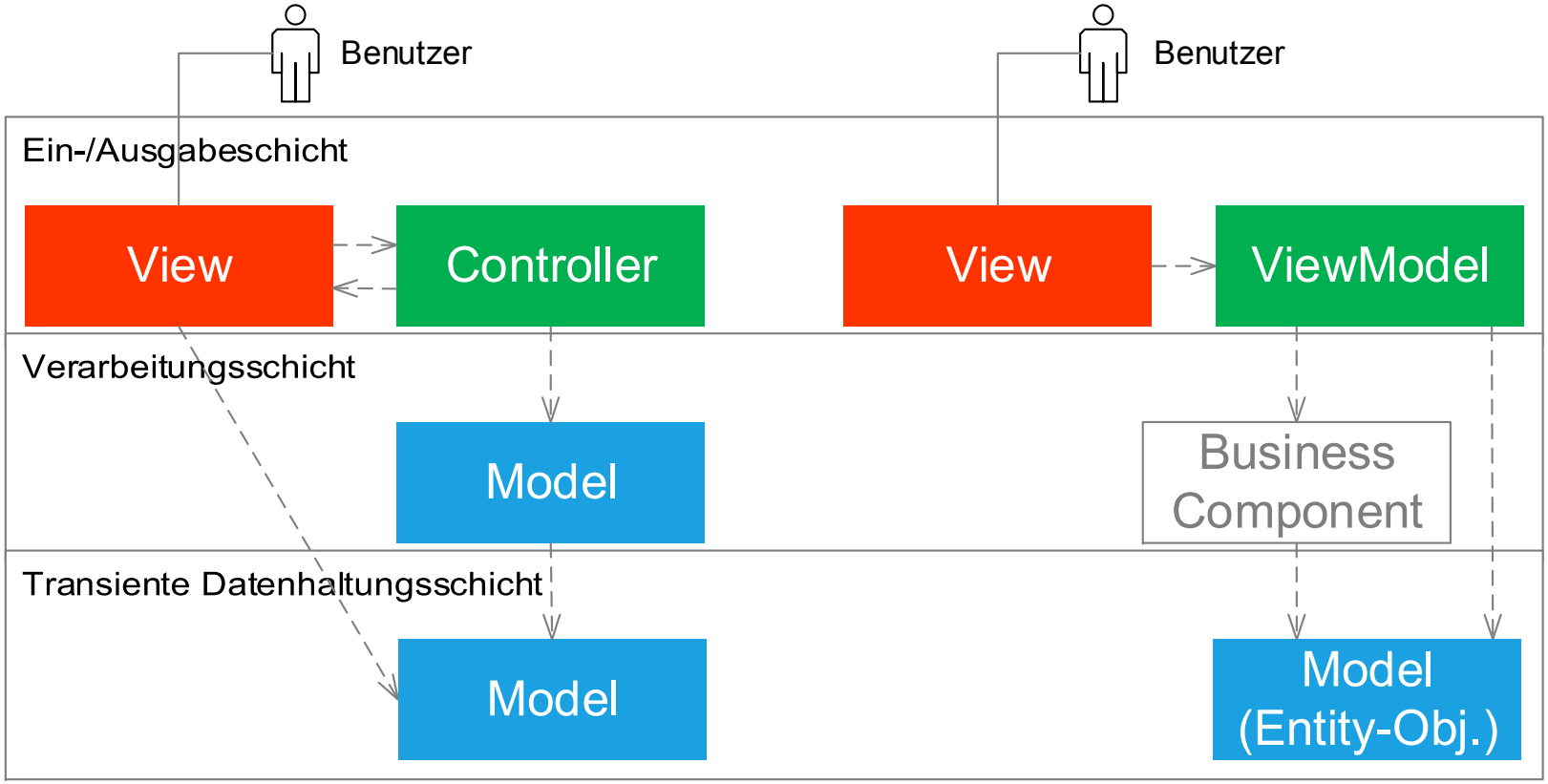
- ViewModel stellt für die View bereit:
  - Datenfelder (Properties)
  - Eingabeverarbeitungslogik (Commands)
    - z. B. Verarbeitung von Formulardaten nach Schaltflächendruck
  - Zustand der Kontrollelemente
    - z. B. Schaltflächen aktiviert/deaktiviert

- ViewModel nutzt vom Model die transiente Datenhaltung

- ViewModel ruft ferner die Verarbeitungslogik der Business Component (ehem. Model) auf

ViewModel

- Weniger Abhängigkeiten
- Eindeutigere Trennung der Concerns (SoC)



Vergleich MVC & MVVM

- MVVM nativ nur bei:
  - XAML-Anwendungen (.NET Framework)
    - WPF,
    - Silverlight und
    - Universal Apps
  - Web-Frameworks wie
    - AngularJS und
    - KnockoutJS

- Aber es existieren auch Portierungen:
  - mvvmFX für JavaFX
  - Data Binding Library für Android Entwickler (derzeit Beta)

Einsatzgebiete

- Basiert auf der einfachen Synchronisation der Datenfelder, weshalb die Unterstützung durch das eingesetzte Framework gegeben sein muss

- Data Binding erfolgt zur Laufzeit. Etwaige Laufzeitfehler unter Umständen schwer zu finden

- Für kleine Projekte möglicherweise Overhead

Nachteile

- Klare Aufgabentrennung (Komponenten)
- Ermöglicht Aufgabenverteilung bzw. Parallelisierung der Entwicklung
- Geringe Abhängigkeiten
- Gute Erweiterbarkeit
- Einfaches Finden von Fehlern
- Effizientes Testen von Model und ViewModel
- Komponenten wiederverwendbar
- Einfacher Austausch des schnelllebigen MMI möglich

Vorteile

VIELEN DANK FÜR IHRE AUFMERKSAMKEIT