Evaluation des Model View Controllers

Seminararbeit

für die Vorlesung

Advanced Software-Engineering

an der Dualen Hochschule Baden-Württemberg Stuttgart

von

Maximilian Bender von Säbelkampf, Kevin Basener

Dezember 2024

Erklärung

Wir versichern hiermit, dass wir unsere Seminararbeit mit dem Thema: *Evaluation des Model View Controllers* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

Stuttgart, Dezember 2024

Kevin Basener

Maximilian Bender von Säbelkampf, Kevin Basener

Abstract

Diese Seminararbeit untersucht das Model-View-Controller (MVC) Architektur-Pattern und vergleicht es mit verwandten Architektur-Pattern wie Model-View-Presenter (MVP) und Model-View-ViewModel (MVVM). Ziel ist es, die Vor- und Nachteile dieser Architektur-Pattern hinsichtlich Modularität, Kohäsion, Kopplung und Code Overhead zu bewerten. Die Arbeit beginnt mit einer detaillierten Vorstellung der MVC-Architektur, einschließlich ihrer Eigenschaften, grafischen Darstellung und eines Codebeispiels. Anschließend werden die Bewertungsmethoden für Architektur-Pattern erläutert. Die Untersuchung zeigt, dass jedes Architektur-Pattern spezifische Stärken und Schwächen aufweist, die je nach Anwendungsszenario unterschiedlich gewichtet werden können.

Inhaltsverzeichnis

Abkürzungsverzeichnis

MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel

Abbildungsverzeichnis

1 Einleitung

In der Softwareentwicklung spielen Architektur-Pattern eine entscheidende Rolle, um die Struktur und Wartbarkeit von Anwendungen zu verbessern. Eines der bekanntesten und am häufigsten verwendeten Pattern ist das Model-View-Controller (MVC) Pattern. Dieses Architektur-Pattern teilt eine Anwendung in drei Hauptkomponenten: Model, View und Controller. Jede dieser Komponenten hat spezifische Aufgaben und Verantwortlichkeiten, was zu einer klaren Trennung der Anliegen führt und die Wartung und Erweiterung der Anwendung erleichtert.

Neben MVC gibt es weitere verwandte Architektur-Pattern wie Model-View-Presenter (MVP) und Model-View-ViewModel (MVVM), die ähnliche Ziele verfolgen, aber unterschiedliche Ansätze zur Trennung von Logik und Darstellung bieten. Diese Pattern sind besonders nützlich in komplexen Anwendungen, bei denen eine klare Strukturierung und Modularität entscheidend sind.

In dieser Arbeit werden die Eigenschaften und Vorzüge des MVC-Patterns sowie die verwandten Architektur-Pattern MVP und MVVM untersucht. Dabei wird auf die Modularität, Kohäsion, Kopplung und den Code Overhead eingegangen, um die jeweiligen Vor- und Nachteile zu ermitteln und eine objektive Analyse zu gewährleisten.

2 Vorstellung der MVC-Architektur

Das MVC-Pattern ist ein bewährtes Software-Architekturpattern, dass die Strukturierung von Anwendungen in drei Hauptkomponenten vorsieht: Model, View und Controller. Diese Trennung der Verantwortlichkeiten erleichtert die Wartung und Erweiterung der Anwendung und fördert die Wiederverwendbarkeit und Testbarkeit der einzelnen Komponenten. MVC wurde ursprünglich für die Benutzeroberflächenentwicklung in Smalltalk (Programmiersprache) entwickelt und hat sich seitdem als Standard für die Strukturierung von Webanwendungen etabliert [1].

Das MVC-Pattern ist besonders nützlich in komplexen Anwendungen, bei denen eine klare Trennung zwischen Daten, Benutzeroberfläche und Steuerungslogik erforderlich ist. Es ermöglicht Entwicklern, sich auf spezifische Aspekte der Anwendung zu konzentrieren, ohne sich um die anderen Teile kümmern zu müssen. Dies führt zu einer besseren Organisation des Codes und erleichtert die Zusammenarbeit in Teams.

2.1 Eigenschaften

Das MVC-Pattern teilt eine Anwendung in drei Hauptkomponenten, die jeweils spezifische Aufgaben und Verantwortlichkeiten haben. Diese Hauptkomponenten werden im folgenden genauer erklärt [2]:

- Model: Diese Komponente repräsentiert die Daten und die Geschäftslogik der Anwendung. Sie ist verantwortlich für das Abrufen, Speichern und Verarbeiten von Daten. Das Model benachrichtigt die View über Änderungen, sodass die Benutzeroberfläche aktualisiert werden kann. Das Model enthält die Kernfunktionalität der Anwendung und ist unabhängig von der Benutzeroberfläche.
- View: Die View-Komponente stellt die Benutzeroberfläche dar. Sie zeigt die Daten des Models an und sendet Benutzereingaben an den Controller weiter. Die View ist für die Darstellung der Daten zuständig und reagiert auf Änderungen im Model. Sie ist passiv und kennt die Geschäftslogik nicht; sie zeigt nur die Daten an, die ihr vom Model bereitgestellt werden.

- Controller: Der Controller verarbeitet die Benutzereingaben, interagiert mit dem Model und aktualisiert die View entsprechend. Er fungiert als Vermittler zwischen Model und View und steuert den Datenfluss zwischen diesen Komponenten. Der Controller interpretiert die Eingaben des Benutzers und ruft die entsprechenden Methoden im Model auf, um die Daten zu ändern. Anschließend aktualisiert er die View, um die neuen Daten anzuzeigen.

Diese Struktur ermöglicht eine klare Trennung der Verantwortlichkeiten, was die Wartbarkeit und Erweiterbarkeit der Anwendung erleichtert. Entwickler können Änderungen an einer Komponente vornehmen, ohne die anderen Komponenten zu beeinflussen. Dies ist besonders nützlich in großen Projekten, bei denen mehrere Entwickler gleichzeitig an verschiedenen Teilen der Anwendung arbeiten.

2.2 Grafische Darstellung

Ein Klassendiagramm für das MVC-Pattern könnte wie folgt aussehen. Es zeigt die Beziehungen und Interaktionen zwischen Model, View und Controller.

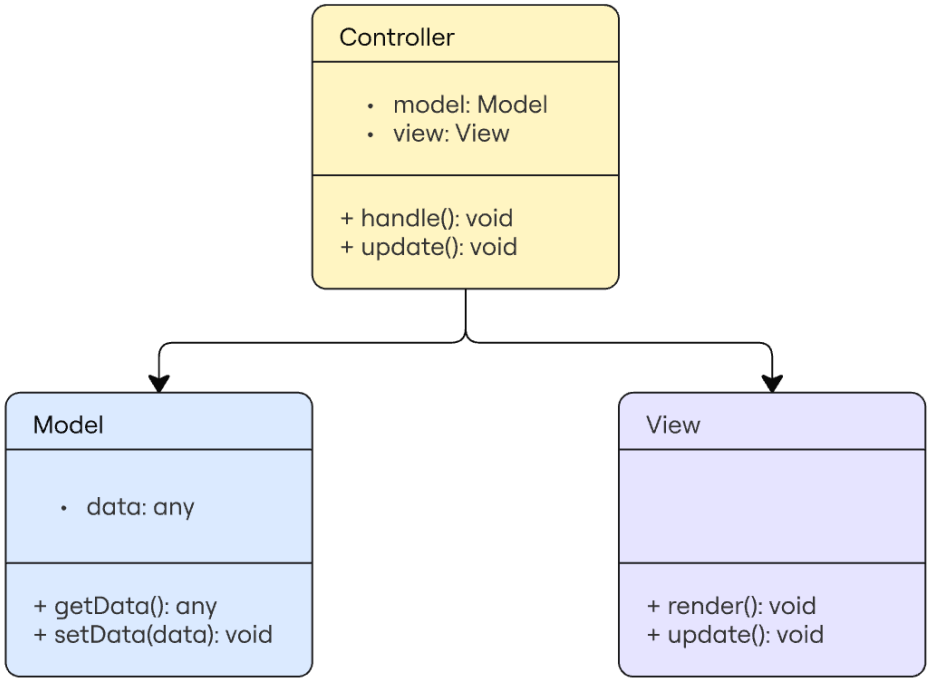

Abbildung 2.1: MVC-Pattern Klassendiagramm

Das Klassendiagramm sehen wir die folgenden drei Komponenten:

- Der Controller enthält Referenzen auf das Model und die View und hat Methoden zum Verarbeiten von Eingaben und Aktualisieren der View. Er ist das Bindeglied zwischen Model und View und sorgt dafür, dass die Benutzerinteraktionen korrekt verarbeitet werden.
- Das Model enthält die Daten und Methoden zur Datenmanipulation. Es stellt die Geschäftslogik der Anwendung dar und ist unabhängig von der Benutzeroberfläche.
- Die View enthält Methoden zur Darstellung der Daten und zur Aktualisierung der Benutzeroberfläche. Sie zeigt die Daten an, die vom Model bereitgestellt werden, und reagiert auf Änderungen im Model.

Um die Interaktionen zwischen den Komponenten zu verstehen, werden diese im folgenden anhand eines Beispiels erklärt. In diesem Beispiel interagiert ein Nutzer mit der Benutzeroberfläche. Darauf hin werden folgende Interaktionen durchgeführt.

1. Der Benutzer interagiert mit der View (z.B. durch Klicken auf einen Button).
2. Die View sendet die Benutzereingabe an den Controller.
3. Der Controller verarbeitet die Eingabe und aktualisiert das Model.
4. Das Model gibt die Änderungen für die View an den Controller weiter.
5. Der Controller gibt die veränderten Daten an die View weiter.
6. Die View rendert die aktualisierten Daten.

Diese Abfolge zeigt, wie die Komponenten zusammenarbeiten, um Benutzereingaben zu verarbeiten und die Benutzeroberfläche zu aktualisieren. Die klare Trennung der Verantwortlichkeiten erleichtert die Wartung und Erweiterung der Anwendung und fördert die Wiederverwendbarkeit und Testbarkeit der einzelnen Komponenten [1].

2.3 Code Beispiel

Im folgenden wird das MVC-Pattern anhand eines abstrakten Codebeispiels, in der Programmiersprache Python, veranschaulicht. Hierbei wird sich am vorgestellten Klassendiagramm (siehe Kapitel 1.2) orientiert.

```
# Model
class Model:
    def __init__(self):
        self.data = "Initial Data"

    def get_data(self):
        return self.data

    def set_data(self, data):
        self.data = data

# View
class View:
    def render(self, data):
        print(f"Data: {data}")

    def update(self, data):
        self.render(data)

# Controller
class Controller:
    def __init__(self, model, view):
        self.model = model
        self.view = view

    def handle(self):
        self.view.render(self.model.get_data())

    def update(self, data):
        self.model.set_data(data)
        self.view.update(self.model.get_data())

# Anwendung
if __name__ == "__main__":
    model = Model()
    view = View()
    controller = Controller(model, view)
```

```
# Initiale Anzeige
controller.handle()

# Daten aktualisieren
controller.update("Neue Daten")
```

In diesem Beispiel:

- Das `Model` speichert die Daten und stellt Methoden zum Abrufen und Aktualisieren der Daten bereit.
- Die `View` zeigt die Daten an und aktualisiert die Anzeige basierend auf den Daten, die vom Model bereitgestellt werden.
- Der `Controller` vermittelt zwischen Model und View, verarbeitet Benutzereingaben und aktualisiert das Model entsprechend.

3 Methodik zur Bewertung von Architektur-Pattern

Olsson et al. [3] haben Kriterien zur Untersuchung der MVC-Architektur in Videospielen festgelegt. Diese Kriterien können auch zur Bewertung verschiedener Architektur-Pattern verwendet werden. Die Bewertung orientiert sich hierbei an allgemeinen Kriterien, die zur Analyse der Architekturstruktur und ihrer Effektivität herangezogen werden. Im Fokus stehen die folgenden Aspekte:

- **Modularität**: Untersucht wird, wie klar die Trennung von Modell, View und Controller ist. Ein hohes Maß an Trennung deutet auf eine verbesserte Wartbarkeit hin.
- **Kohäsion**: Die Kohäsion misst, wie eng zusammenhängend die Verantwortlichkeiten der einzelnen Komponenten sind. Eine hohe Kohäsion innerhalb einer Komponente ist ein positives Zeichen für eine klare und fokussierte Aufgabenverteilung.
- **Kopplung**: Die Kopplung bewertet, wie stark die Abhängigkeiten zwischen den verschiedenen Komponenten sind. Eine niedrige Kopplung zwischen Klassen ist ein positives Zeichen für Wiederverwendbarkeit und Flexibilität.
- **Code Overhead**: Hierbei wird analysiert, inwiefern die Implementierung eines Patterns zusätzlichen Codeaufwand erzeugt und wie sich dies auf die Effizienz der Anwendung auswirkt.

Jede dieser Metriken dient dazu, die strukturellen Vorzüge sowie potenzielle Nachteile des jeweiligen Architektur-Pattern zu ermitteln und eine objektive Analyse zu gewährleisten.

4 Untersuchung des MVC-Pattern und verwandter Pattern

In dieser Untersuchung werden drei wichtige Architektur-Pattern analysiert: MVC, MVP und MVVM. Diese Pattern wurden basierend auf der Untersuchung von Aihara et al. [4] ausgewählt, da diese häufig in der Praxis verwendet werden und unterschiedliche Ansätze zur Strukturierung von Software bieten. Die Bewertung erfolgt anhand der in Kapitel 2 beschriebenen vier Kriterien: Modularität, Kohäsion, Kopplung sowie Code Overhead, um die jeweiligen Vor- und Nachteile zu ermitteln.

4.1 MVC (Model-View-Controller)

Das MVC-Pattern bietet klare Vorteile in der Modularität und Wiederverwendbarkeit, da es eine strikte Trennung zwischen Modell, View und Controller vorsieht. Diese Trennung ermöglicht es Entwicklern, die Benutzeroberfläche zu ändern, ohne dass Änderungen am zugrunde liegenden Datenmodell und der Anwendungslogik notwendig sind. Dadurch wird die Wartbarkeit des Systems erhöht. Dieser Vorteil zeigt sich besonders in großen Projekten, bei denen eine klare Strukturierung der Codebasis essenziell ist [5], [6].

Die Kopplung des MVC-Pattern zwischen Controller und View wird oft als Nachteil angesehen. Diese starke Abhängigkeit bedeutet, dass Änderungen bei einer der beiden Komponenten häufig zu Änderungen in der anderen führen. Dadurch wird die Flexibilität des Systems verringert und der Wartungsaufwand erhöht [6].

Ein weiterer Nachteil des MVC-Pattern ist der durch die Trennung entstehende Code Overhead. Dieser zeigt sich vor allem in der erhöhten Komplexität der Kommunikation zwischen den drei Komponenten. Dies kann insbesondere bei komplexen Benutzeroberflächen zu Leistungseinbußen führen. Hinzukommend kann der Entwicklungsprozess von Systemen mit mehreren Benutzeroberflächen (Views) verlangsamt und komplexer werden [5].

4.2 MVP (Model-View-Presenter)

Das MVP-Pattern fördert eine klare Trennung zwischen den verschiedenen Komponenten der Architektur. Der Presenter fungiert als Vermittler zwischen dem Model und der View. Dadurch wird eine hohe Modularität erreicht [7]. Diese Trennung ermöglicht es, dass verschiedene Views den gleichen Presenter nutzen können. Dies reduziert die Anzahl der benötigten Komponenten und erhöht die Wiederverwendbarkeit des Codes.

Die Kohäsion im MVP-Pattern ist hoch, da jede Komponente spezifische Aufgaben hat. Das Model kümmert sich um die Daten, die View um die Darstellung und der Presenter um die Vermittlung und Steuerung der Logik zwischen beiden.

Die Kopplung zwischen View und Presenter ist jedoch relativ eng, da der Presenter direkten Zugriff auf die View hat, um sie zu aktualisieren. Dies kann die Testbarkeit der einzelnen Komponenten beeinträchtigen [4]. Im Vergleich zu MVC bietet MVP eine bessere Kontrolle über die Interaktion zwischen den Komponenten, da der Presenter als alleiniger Vermittler fungiert und die View passiver ist. Dies reduziert die Komplexität der Kopplung und verbessert die Testbarkeit.

Ein Nachteil von MVP ist der zusätzliche Code Overhead, der durch die Interaktion zwischen Presenter und View entsteht. Der Presenter verwaltet die Benutzerinteraktionen und aktualisiert die View. Dies kann zu einem höheren Aufwand in der Codebasis führen. Dennoch überwiegen die Vorteile in Bezug auf Testbarkeit und Wartbarkeit oft diesen zusätzlichen Aufwand [7].

4.3 MVVM (Model-View-ViewModel)

Das MVVM-Architekturpattern zeichnet sich durch eine klare Trennung von View- und Geschäftslogik aus. MVVM bietet eine hohe Modularität, da die View und die Geschäftslogik (Model) durch das ViewModel klar getrennt sind. Diese Trennung ermöglicht es Entwicklern, die Benutzeroberfläche unabhängig von der Geschäftslogik zu entwickeln und zu testen. Die Möglichkeit der automatischen Synchronisierung zwischen View und ViewModel führt zu einer besseren Wartbarkeit und erleichtert die Wiederverwendung von Komponenten [8].

Die Kohäsion in MVVM ist hoch, da jede Komponente (Model, View, ViewModel) klar definierte Verantwortlichkeiten hat. Das Model kapselt die Geschäftslogik und Daten, die View ist für die Darstellung und Benutzerinteraktion zuständig, und das ViewModel dient als Bindeglied zwischen den beiden. Diese klare Trennung der Verantwortlichkeiten führt

zu einer besseren Strukturierung des Codes und erleichtert die Wartung und Erweiterung der Anwendung [8].

Ein Nachteil von MVVM ist die potenziell hohe Kopplung zwischen View und ViewModel. Obwohl das ViewModel die Geschäftslogik von der View trennt, kann die extensive Nutzung von Datenbindungen (Bindings) zu einer engen Kopplung führen. Diese enge Kopplung kann die Flexibilität und Testbarkeit der Anwendung beeinträchtigen, da Änderungen in der View auch Anpassungen im ViewModel erfordern können [8].

MVVM kann zu einem höheren Code Overhead führen, da zusätzliche Klassen und Bindungen erforderlich sind, um die Trennung von View und Geschäftslogik zu gewährleisten. Insbesondere die Implementierung und Verwaltung von Bindungen zwischen View und ViewModel kann komplex und zeitaufwendig sein. Zudem kann die extensive Nutzung von Observer-Synchronisation zu Leistungseinbußen führen, insbesondere bei großen und komplexen Anwendungen [8].

4.4 Vergleich der vorgestellten Architektur-Pattern

Die Modell-View-Architekturpattern MVC, MVP und MVVM sind alle Teil der Familie der MV*-Architekturpattern, die sich auf die Trennung von Anliegen konzentrieren [4]. Diese Pattern sind entstanden, um die Integration von Benutzeroberflächen mit der Anwendungsdomäne zu verbessern, insbesondere als grafische Benutzeroberflächen in den 1980er Jahren populär wurden. MVC wurde 1988 dokumentiert und stellte die Grundlage für die Entwicklung weiterer Pattern dar [4].

Alle MV*-Pattern bestehen aus drei Hauptkomponenten: Modell (M), View (V) und einer dritten, variablen Komponente, die beschreibt, wie M und V miteinander kommunizieren [4]. Im MVC-Pattern übernimmt der Controller die Rolle der Verbindung zwischen dem Modell und der Ansicht, indem er Benutzereingaben verarbeitet und die Darstellung aktualisiert [4]. Im Gegensatz dazu trennt MVP die Logik weiter, indem der Presenter die Interaktion zwischen View und Model steuert, was zu einer höheren Modularität führt [4]. In MVVM wird diese Trennung weiter verfeinert, indem das ViewModel als Bindeglied zwischen der View und dem Modell fungiert und eine automatische Synchronisierung ermöglicht [4].

5 Zusammenfassung

Die Untersuchung der Architektur-Pattern MVC, MVP und MVVM zeigt, dass jedes dieser Pattern spezifische Vor- und Nachteile bietet, die je nach Anwendungsfall unterschiedlich gewichtet werden können. Das MVC-Pattern zeichnet sich durch eine klare Trennung der Verantwortlichkeiten aus, was die Wartbarkeit und Erweiterbarkeit der Anwendung erleichtert. Allerdings kann die starke Kopplung zwischen Controller und View zu Herausforderungen bei der Flexibilität und Testbarkeit führen.

Das MVP-Pattern bietet eine verbesserte Kontrolle über die Interaktion zwischen den Komponenten und fördert eine hohe Modularität und Kohäsion. Die enge Kopplung zwischen View und Presenter kann jedoch die Testbarkeit beeinträchtigen, und der zusätzliche Code Overhead kann den Entwicklungsaufwand erhöhen.

Das MVVM-Pattern ermöglicht eine automatische Synchronisierung zwischen View und ViewModel, was die Wartbarkeit und Wiederverwendbarkeit der Komponenten verbessert. Die potenziell hohe Kopplung zwischen View und ViewModel sowie der zusätzliche Code Overhead durch die Implementierung von Bindungen können jedoch zu Herausforderungen führen.

Insgesamt bieten alle drei Architektur-Pattern wertvolle Ansätze zur Strukturierung von Softwareanwendungen. Die Wahl des geeigneten Pattern hängt von den spezifischen Anforderungen und Zielen des Projekts ab. Durch die Analyse der Modularität, Kohäsion, Kopplung und des Code Overheads können Entwickler fundierte Entscheidungen treffen, um die bestmögliche Architektur für ihre Anwendungen zu wählen.

Literatur

[1] N. Delessy-Gassant und E. B. Fernandez, „The Secure MVC pattern,“ in *10th Latin American and Caribbean Conference for Engineering and Technology*, LACCEI International Symposium on Software Architecture and Patterns (LACCEI-ISAP-MiniPLoP'2012), Panama City, Panama, 2012.

[2] W. Cui, L. Huang, L. Liang und J. Li, „A Development Framework of PHP Based on MVC Design Patterns - FDF Framework,“ in *2009 Fourth International Conference on Computer Sciences and Convergence Information Technology*, Date Added to IEEE Xplore: 31 December 2009, Seoul, Korea (South): IEEE, Nov. 2009, S. 1077–1082. DOI: 10.1109/ICCIT.2009.130.

[3] T. Olsson, D. Toll, A. Wingkvist und M. Ericsson, „Evolution and Evaluation of the Model-View-Controller Architecture in Games,“ in *Proceedings of the 2015 IEEE/ACM 4th International Workshop on Games and Software Engineering*, IEEE, 2015, S. 8–14. DOI: 10.1109/GAS.2015.10. Adresse: https://doi.org/10.1109/GAS.2015.10.

[4] D. S. Aihara, „A Journey Through the Land of Model-View Design Patterns,“ Master of Science Thesis, Diss., Universidad Politécnica de Madrid und Blekinge Institute of Technology, 2012. Adresse: /mnt/data/a_journey_through_the_land_of_model_view_design_pattern.pdf.

[5] Anonymous, „Understanding MVC: A Study in Web Application Development,“ *EUDL (European Union Digital Library)*, 2020. Adresse: https://eudl.eu/doi/10.4108/eai.20-10-2020.2305151.

[6] Anonymous, „Demystifying MVC Architecture,“ *ResearchGate*, 2019. Adresse: https://www.researchgate.net/publication/334555489_Demystifying_MVC_Architecture.

[7] M. R. J. Qureshi und F. Sabir, „A COMPARISON OF MODEL VIEW CONTROLLER AND MODEL VIEW PRESENTER,“ *Journal of Computing and Information Technology*, Jg. XX, Nr. YY, ZZ–ZZ, 2024.

[8] G. Arcos-Medina, J. Menéndez und J. Vallejo, „Comparative Study of Performance and Productivity of MVC and MVVM design patterns,“ in *Simposio Iberoamericano en Programación Informática (Ibero-American Symposium on Computer Programming)*, Knowledge E, 2018, S. 241–252. DOI: 10.18502/keg.v1i2.1498. Adresse: https://doi.org/10.18502/keg.v1i2.1498.