Eine Architektur für grafische Benutzeroberflächen nach dem MVC-Architekturmuster und unter Berücksichtigung der SOLID-Architekturprinzipien

Stefan Berger
Beuth Hochschule für Technik Berlin
Luxemburger Str. 10
13353 Berlin
s.berger@live.com

ABSTRACT

Dieses Dokument befasst sich mit dem Model-View-Controller (MVC) Architekturmuster. Es befasst sich weiterhin mit den fünf SOLID-Architekturprinzipien Single Responsibility, Open-Closed, Liskovsches Substitionsprinzip, Interface Segregation und Dependency Inversion. Das Architekturmuster wird auf die Architekturprinzipien hin überprüft und angepasst. In Beispielimplementierungen wird die Praxistauglichkeit nach den Anpassungen untersucht.

Das Liskovsche Substitionsprinzip kann als einziges Prinzip beim Entwurf der neuen Architektur nicht beachtet werden, sondern erst wenn die abstrakten Klassen implementiert und erneut erweitert werden.

Die neue Struktur bildet ein Grundgerüst, das aus den drei Komponenten des Architekturmusters sowie aus deren Unterkomponenten besteht. Die Komponenten und Unterkomponenten sind lose gekoppelt. Das Grundgerüst ist flexibel einsetzbar, stabil und erweiterbar.

1. INTRODUCTION

Seit der Entwicklung des Architekturmusters MVC hat sich in den Programmentwürfen für Benutzerschnittstellen die Trennung von Daten und View immer mehr durchgesetzt. Dabei wird der Quelltext einer Anwendung in die drei Klassen Programmlogik (Model), Ansicht (View) und Steuerung (Controller) unterteilt. Diese Struktur macht den Quelltext für Views und Controller austauschbar und wiederverwendbar. Außerdem erleichtern die Abstrahierung und die Modularisierung die Zusammenarbeit mehrerer Entwickler. Solange die Schnittstellen unverändert bleiben, können die verschiedenen Klassen flexibel und voneinander unabhängig bearbeitet werden.

Zwei der wichtigsten Argumente der SOLID-Prinzipien sind Wiederverwendbarkeit und Flexibilität. Außerdem soll durch deren Anwendung die Lesbarkeit und die Erweiterbarkeit des Quelltexts sichergestellt werden, und Entwicklerteams sollen effizienter zusammenarbeiten können. Die SOLID-Prinzipien sind in dem Buch “Agile Software Development, Principles, Patterns, and Practices” von Robert C. Martin ausführlich beschrieben.

Wir werden den Abschnitten 2.1 bis 2.4 zuerst die Komponenten des MVC-Architekturmusters genau betrachten, anschließend in Abschnitt 3 die fünf SOLID-Architekturprinzipien. Die drei Komponenten des MVC-Architekturmusters werden wir in Abschnitt jeweils auf die Relevanz der fünf SOLID-Prinzipien untersuchen und entsprechend anpassen. In Beispielimplementierungen werden wir die Praxistauglichkeit überprüfen.

2. ARCHITEKTURMUSTER UND ARCHITEKTURPRINZIPIEN

2.1 MVC

Das Architekturmuster MVC sieht für interaktive Anwendungen die Trennung in ein Modul für die Programmlogik (Model), ein Viewmodul (View) und ein Steuerungsmodul (Controller) vor [2, S. 26-49]. Jedes dieser Module steht mit jedem anderen in einer bestimmten Relation.

2.2 Model

Im Model befindet sich die eigentliche Programmlogik. Ein typisches Szenario ist eine Interaktion des Benutzers mit der View, die durch den Controller an das Model vermittelt wird. Nach Abschluss einer Verarbeitung der Interaktion durch das Model benachrichtigt es die View über geänderte Daten. Die View aktualisiert schließlich die Darstellung der Daten. Danach oder auch währenddessen können weitere Interaktionen stattfinden. Das Model ist normalerweise einzigartig, d.h. Views und Controller kennen nur ein Model.

2.3 View

Die Relationen der drei Module werden durch verschiedene Entwurfsmuster realisiert. Die View als Observer des Models [1, S. 293—305] wird über Änderungen am Model benachrichtigt. Hierfür registriert die View einen oder mehrere Callbacks (Methoden oder Funktionen), die immer dann aufgerufen werden, wenn sich an den anzuzeigenden Daten etwas ändert. Man sagt, dass die View das Model kennt, das Model die View aber nicht – obwohl aus programmatischer Sicht dem Model dadurch eine Schnittstelle der View bekannt gemacht wird. Mehrere Views können außerdem

untereinander in Relation stehen. Mit dem Composite-Pattern [1, S. 163—175] können Teilkomponenten der View zum Gesamtmodul zusammengefasst werden. Sowohl das Gesamtmodul als auch die Teilkomponenten können dann auf Änderungen am Model reagieren.

2.4 Controller

Benutzereingaben werden vom Controller interpretiert. In Form des Strategy-Patterns [1, S. 315—325] wird der View ein Controller zugewiesen. Eine View kann einen oder mehrere Controller besitzen. In der Regel werden aber sogenannte View-Controller-Paare erzeugt, also eine View mit einem Controller. Der Controller der View kann sich jederzeit ändern, auch zur Laufzeit. Wir betrachten später ein Beispiel, in dem gleiche Benutzereingaben aufgrund geänderter Zustände unterschiedlich interpretiert werden. Controller besitzen außerdem die Fähigkeit, andere Views aufzurufen. Ein Controller benötigt deshalb Zugriff auf entsprechende Schnittstellen der Laufzeitumgebung. Ein Controller muss außerdem das Datenmodell der Ansicht manipulieren können.

3. SOLID

3.1 Single Responsibility

Das Single-Responsibility-Prinzip besagt, dass es *für eine Klasse nur einen Grund zur Änderung geben sollte* [3, S. 95]. Nehmen wir an, dass eine Datenbankanwendung um eine Filtermöglichkeit nach betriebsspezifischen Kennzahlen – etwa dem Anteil eines Verkaufsartikels am Umsatz – erweitert werden soll. Um konkrete Werte für gefilterte Abfragen eingeben zu können, müssen der View entsprechende Steuerelemente hinzugefügt werden. Jede Änderung, die außerdem an der View vorgenommen werden muss, weist auf eine Verletzung des Single-Responsibility-Prinzips hin. Es ist zum Beispiel denkbar, dass der Ergebnistabelle im Viewmodul eine Spalte mit der neuen Kennzahl hinzugefügt werden soll. Offensichtlich müssen die Steuerelemente zur Eingabe der Abfragewerte und die Darstellung des Abfrageergebnisses in unterschiedlichen Viewklassen implementiert werden, sodass sichergestellt ist, dass jede Klasse eine einzige Verantwortlichkeit besitzt.

3.2 Open-Closed

Open-Closed sind *Klassen, Module, Funktionen etc., die für Erweiterungen offen, aber für Modifikationen verschlossen sind* [3, S. 99]. Ein Anwendungsgerüst muss flexibel erweiterbar sein, aber auch eine unveränderliche Grundfunktionalität bieten, die von Erweiterungen nicht beeinträchtigt wird.

Eine Hauptaufgabe von Controllerklassen ist es, Benachrichtigungen entgegenzunehmen und weiterzuleiten. Weil die Stabilität des bestehenden Quelltexts davon abhängt, dass der Controller diese Aufgaben zuverlässig ausführt, muss die Controllerklasse für Änderungen an ihren Funktionen verschlossen sein. Neue Controller-Implementierungen müssen zusätzlich weitere Aufgaben ausführen. Das Modul muss deshalb für solche Erweiterungen offen sein.

3.3 Liskovsches Substitutionsprinzip

Vereinfacht gesagt schreibt das Liskovsche Substitutionsprinzip vor, dass *Obertypen durch Untertypen ersetzbar sein* sollen [3, S. 111]. Angenommen, einer von mehreren Teilkomponenten des Viewmoduls soll ein Textfeld hinzugefügt werden. Die Teilkomponente ohne das zusätzliche Textfeld soll aber weiterhin zur Verfügung stehen. Es ist naheliegend, für diesen Zweck eine neue Viewklasse von der bestehenden abzuleiten. Bei Bedarf kann dann ein Objekt der Basisklasse durch ein Objekt der abgeleiteten Klasse ersetzt werden. Wird das Liskovsche Substitutionsprinzip befolgt, kann das Objekt der abgeleiteten Klasse verwendet werden, ohne das Viewmodul zu verändern. Ein wichtiger Aspekt ist dabei das neue Datenfeld, das dem Viewmodul unbekannt ist.

3.4 Interface Segregation

Interface Segregation bedeutet, dass eine Klasse einer anderen genau die Schnittstelle zur Verfügung stellt, die für den jeweiligen Zweck vorgesehen ist. In MVC verwenden sowohl die View als auch der Controller Schnittstellen des Models. Die View muss sich als Empfänger für Benachrichtigungen über Änderungen registrieren. Der Controller muss das Model über Benutzereingaben benachrichtigen. Daraus ergeben sich zwei verschiedene Schnittstellen zum Model, von denen die View und der Controller auch jeweils nur eine kennen sollten.

3.5 Dependency Inversion

Das Dependency-Inversion-Prinzip schreibt vor, dass Module nicht von anderen Modulen, die sich niedriger in der Modulhierarchie befinden, abhängig sein sollten [3, S. 127]. Wir werden dieses Prinzip bei der Implementierung des Viewmoduls und seiner Teilkomponenten berücksichtigen.

4. SOLID VIEW

4.1 Single Responsibility

Ein Viewmodul beinhaltet wenig Programmlogik. Es reagiert auf Änderungen an den Anwendungsdaten und macht diese sichtbar. Für Benachrichtigungen über Änderungen ist das Observer-Entwurfsmuster vorgesehen. Ein Observer besitzt eine Verantwortlichkeit im Sinne des Single-Responsibility-Prinzips. Die Registrierung am Model und der Callback sollten sich deshalb in einer eigenen Klasse befinden.

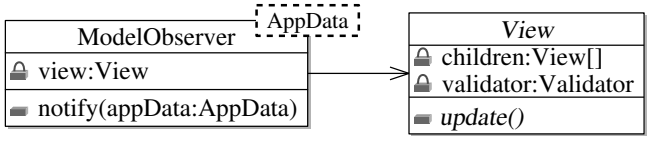

Figure 1: Klassendiagramm der View

Abbildung 1 zeigt die Klassenstruktur des Viewmoduls. Die Klasse `ModelObserver` erhält Benachrichtungen über Änderungen im Datenmodell der Anwendung und leitet sie an die Klasse `View` weiter. Die Klasse `View` ist für die Darstellung der Daten und Zustände verantwortlich. Diese Klassenstruktur befolgt das Single-Responsibility-Prinzip besser als eine einzige `View`-Klasse, in der alle diese Aspekte implementiert sind.

4.2 Open-Closed

Das Viewmodul ist für Erweiterungen an seiner Präsentationslogik offen. Die Klasse `ModelObserver` ist keiner anderen Klasse

bekannt und ist deshalb vollständig implementiert. Die Klasse View ist abstrakt. Es sind keine Änderungen am Viewmodul nötig, um neue Views zu implementieren. Die Vorgaben des Open-Closed-Prinzips sind eingehalten.

4.3 Liskovsches Substitionsprinzip

Bisher gibt es im Viewmodul keine Vererbungshierarchie. Für die Erweiterungen des Datenmodells und der Präsentationslogik werden von den bestehenden Klassen neue abgeleitet.

Nennen wir die Klasse, die wir von `View` ableiten, `FormView`. Ein Formular kann Pflichtfelder enthalten, und Formularfelder können bereits mit Werten gefüllt sein. Diese Informationen werden vom Model der Methode `update`, die die neue Klasse erbt bzw. implementiert, im Parameter `appData` übergeben. `FormView` erhält zwei weitere Methoden. Eine Methode füllt Formularfelder mit vorgegebenen Werten und die zweite markiert noch ungesicherte Änderungen. Die beiden neuen Methoden werden aus `update` heraus aufgerufen.

Wir fügen der Klasse `FormView` noch eine Erweiterung hinzu, indem wir wieder eine neue Klasse ableiten. Wir nennen die Klassen `ExtFormView`. Damit wird ein erweitertes Formular realisiert, in dem bestimmte Formularfelder abhängig von einem Radiobutton ein- oder ausgeblendet werden. `ExtFormView` erhält eine Methode, die Formularfelder ein- und ausblendet. Die Methode `update` wird noch einmal überschrieben, um die Basisklassenversion und die neue Methode aufzurufen In `AppData` wird in einem zusätzlichen Attribut die Information gespeichert, ob es versteckte Formularfelder mit ungespeicherten Änderungen gibt, beziehungsweise welchen Radiobutton-Optionen diese zugeordnet sind. Dafür kann eine Klasse `ExtAppData` abgeleitet werden. Wichtig ist aber, dass der Parameter in der Signatur von `update` in `ExtFormView` nicht verändert wird.

Objekte der Klasse `ModelObserver` funktionieren zusammen mit Objekten der Klasse `FormView`, die direkt von der abstrakten Basisklasse abgeleitet ist, genauso wie mit Objekten der Klasse `ExtFormView`, die von der konkreten Implementierung erneut abgeleitet wurde. Die Forderung des Liskovschen Substitutionsprinzips ist erfüllt.

4.4 Interface Segregation im Viewmodul

Ein `ModelObserver` registriert sich im Model als Observer, um vom Model über Änderungen an den anzuzeigenden Daten benachrichtigt zu werden. Die Klasse `View` besitzt ein Attribut vom Typ `Validator`, das im Kapitel über Controller vorgestellt wird. Es ist die Schnittstelle des Controllers, mit der Benutzereingaben an den Controller weitergegeben werden. Die klar definierten Schnittstellen für jede Beziehung zwischen den Komponenenten erfüllt die Forderung des Interface-Segregation-Prinzips.

4.5 Dependency Inversion im Viewmodul

Die Klassen, die direkt oder indirekt von `View` abgeleitet sind, stellen die "High Level Modules" des Viewmoduls im Sinne des Dependency-Inversion-Prinzips dar. Das Prinzip besagt, dass diese Klassen nicht von Implementierungsdetails in den anderen Klassen abhängig sein dürfen. Dadurch soll verhindert werden, dass Änderungen an Implementierungsdetails die Geschäftslogik insgesamt beeinträchtigen. Wie schon beim Open-Closed-Prinzip ermöglicht Abstraktion, diese Anforderung zu implementieren. Keins der Attribute der Klasse `View` ist von einem konkreten Typ, und die Klasse besitzt keine konkreten Methoden. Die Implementierungsdetails befinden sich in abgeleiteten Klassen, ohne die ungewünschte Abhängigkeit zu erzeugen.

5. SOLID CONTROLLER

5.1 Single Responsibility

Der Controller in einer MVC-Architektur hat zwei Aufgaben. Er interpretiert Benutzereingaben, überprüft zum Beispiel Pflichtfelder eines Formulars, und er benachrichtigt gegebenenfalls andere Module über relevante Benutzereingaben.

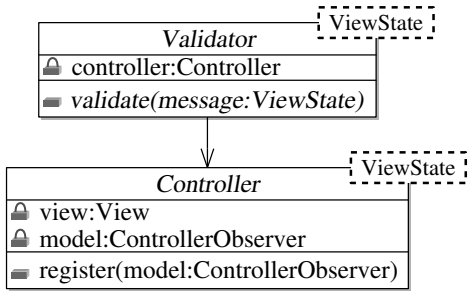

Figure 2: Klassendiagramm des Controllers

Das Controllermodul besteht aus einer Datenstruktur namens `ViewState` für die verschiedenen Zustände der Ansicht, der Klasse `Validator` für die Prüfung der Benutzereingaben und der Klasse `Controller`, wie in Abbildung 2 zu sehen ist. Die Aufgabe der Klasse `Controller` ist es, andere Komponenten über Benutzereingaben zu benachrichtigen. Das Single-Responsibility-Prinzip ist mit dieser Klassenstruktur eingehalten.

5.2 Open-Closed im Controllermodul

Im Controllermodul befindet sich die Klasse `Validator`. Sie nimmt Benachrichtigungen über Benutzereingaben entgegen und gibt unmittelbare Rückmeldungen. Sie besitzt ein Attribut vom Typ `Controller` und ist für Änderungen an diesem Attribut verschlossen. Die Methode `validate`, in der die Regeln für die Gültigkeit von Benutzereingaben definiert werden, ist abstrakt. Sie ist also für Änderungen offen.

Die Klasse `Controller` besitzt zwei unveränderliche Attribute, und zwar die Schnittstellen zur View und zum Model. Sie besitzt außerdem eine Methode zum Registrieren eines `ControllerObserver`s, der Schnittstelle des Models. Diese Methode ist nach Möglichkeit als nicht überschreibbar zu deklarieren, um die Forderungen des Open-Closed-Prinzips zu erfüllen.

5.3 Liskovsches Substitutionsprinzip im Controllermodul

Es gibt im Controllermodul wie auch im Viewmodul zunächst keine Vererbungshierarchie. Erweitern wir die abstrakten Basisklassen, sind die Forderungen des Liskovschen Substitutionsprinzips einzuhalten.

Damit Benutzereingaben auf ihre Gültigkeit überprüft werden können, muss eine konkrete Implementierung der Klasse Validator geschrieben werden. Validator ist eine generische Klasse, deren Implementierungen denselben oder einen von

`ViewState` abgeleiteten Typparameter haben müssen. In der Vererbungsstruktur muss für die Implementierungen und Überschreibungen einer Implementierung die Invarianz oder Kovarianz des Rückgabetyps von `validate` beachtet werden.

In der Schnittstelle der Klasse `Controller` sind keine Übergabe- oder Rückgabeparameter deklariert. Das Liskovsche Substitutionsprinzip ist eingehalten, solange die Schnittstelle nicht verändert wird.

5.4 Interface Segregation im Controllermodul

Das Controllermodul ist nur der View bekannt. Sie erhält Zugriff auf die Klasse `Validator`, um Benutzereingaben zu senden und unmittelbar Rückmeldungen zu erhalten. Die Klasse Controller erhält selbst eine Schnittstelle der View. Views sind je nach Art der Anwendung sehr unterschiedlich strukturiert. Der Controller benötigt mindestens die Methode `update`, um die View über Änderungen am `ViewState` zu benachrichten. Dieser einfache Anwendungsfall kann über das Observer-Entwurfsmuster realisiert werden.

5.5 Dependency Inversion im Controllermodul

Wie in Abschnitt 3.5 angedeutet, ist Dependency Inversion in MVC-Architekturen vor allem im Viewmodul zu beachten. Das Controllermodul definiert zwar auch eine Abhängigkeit der Klasse `Validator` von `Controller`. Weil `Controller` eine abstrakte Klasse ist, wird die Dependency Inversion dadurch aber bereits sichergestellt.

6. SOLID MODEL

6.1 Single Responsibility

Das Model im MVC-Architekturmuster ist noch etwas abstrakter als die anderen Komponenten. View und Controller dienen in MVC immer demselben Zweck, während die Funktion des Models von der Art der Anwendung abhängt. Zum Beispiel verwaltet eine Notizbuch-Anwendung eine Sammlung von Notizen, und hält eine oder mehrere Notizen im Arbeitsspeicher vor. Eine Anwendung für Online-Banking berücksichtigt dagegen vor allem Sicherheitsaspekte und kann erfolglose Transaktionen rückabwickeln.

In MVC ist die Verantwortlichkeit des Models, die View über Änderungen der Daten, die dem Benutzer angezeigt werden sollen, zu informieren.

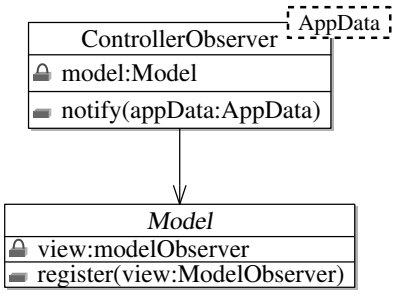

Figure 3: Klassendiagramm des Models

Das Klassendiagramm des Modells in Abbildung 3 sieht dem der View sehr ähnlich. Ein `ControllerObserver` wird über Änderungen informiert, die der Controller an das Model weiterleitet. Änderungen an den Daten, die dem Benutzer angezeigt werden sollen, werden an die View gesendet. Diese Verantwortlichkeit befindet sich in der Klasse Model.

6.2 Open-Closed im Model

Die Nachrichtenübermittlung vom Controller an das Model und vom Model an die View sind essentielle Bestandteile des Models. Sie sind verschlossen für Veränderungen. Denkbare Erweiterungen am Model im Sinne des MVC-Architekturmusters sind mehrere Ansichten oder weitere Observer für Datenquellen, die keine Benutzeroberflächen sind. Dieses Dokument behandelt diese Fälle nicht; das Open-Closed-Prinzip ist wegen der abstrakten Natur des Begriffs Model nur bedingt anwendbar.

6.3 Liskovsches Substitutionsprinzip im Model

Das Model im MVC-Architekturmuster ist nicht austauschbar. Das Substitutionsprinzip ist deshalb für das Model nicht relevant.

6.4 Interface Segregation im Model

Das Model stellt eine Schnittstelle zur Verfügung, durch die es vom Controller über Benutzereingaben informiert wird. Wie in den anderen beiden Komponenten wird die Beziehung durch das Observer-Entwurfsmuster realisiert. Die Methode notify erhält als Argument eine Datenstruktur, die das Model als Nachricht verarbeiten kann. Das Model besitzt selbst genau die Schnittstelle der View, durch die es diese über Änderungen informiert.

6.5 Dependency Inversion im Model

Die Klasse `ControllerObserver` besitzt nur eine Abhängigkeit von der abstrakten Klasse Model. Das Prinzip der Dependency Inversion ist dadurch bereits eingehalten.

7. FAZIT

Das Einhalten der SOLID-Prinzipien beim Entwurf einer MVC-Architektur führt zu kompakten Klassendiagrammen der einzelnen Komponenten. Außerdem entkoppelt es die Komponenten und sogar die Unterkomponenten, wie etwa den Validator des Controllers, voneinander. Die Austauschbarkeit der Kompenenten wird dadurch erhöht, und die Abhängigkeiten werden auf ein Minimum reduziert.

Das Single-Responsibility-Prinzip hat den größten Einfluss auf die MVC-Architektur. Diagramme des MVC-Architekturmusters zeigen normalerweise die drei Komponenten Model, View und Controller und nennen keine weiteren Klassen. Durch die Beschränkung einer Klasse auf genau eine Verantwortlichkeit erweitert sich das Controllermodul auf zwei Unterkomponenten, nämlich auf den Validator und den Controller. Die Gesamtarchitektur besteht nun aus sechs statt vorher drei (Unter-)Komponenten.

Das Open-Closed-Prinzip zwingt uns, die neue Architektur als Gesamtheit zu betrachten. Es ist ein architektonisches Grundgerüst, das erst zu einer Anwendung erweitert werden muss. Für das Grundgerüst war zu beachten, dass bestimmte Eigenschaften einer MVC-Anwendung immer vorhanden sein müssen. Andere Eigenschaften sind nur für bestimmte Anforderungen zu implementieren. Die Architektur kann um diese anderen Eigenschaften erweitert werden, ohne die Komponenten zu verändern.

Das Grundgerüst besitzt eine einfache Struktur und kann an sich nicht das Liskovsche Substitutionsprinzip verletzen. Das Prinzip kann aber in MVC-Anwendungen berücksichtigt werden, die das Grundgerüst nutzen.

Interface Segregation ist das zweite Prinzip, das erheblichen Einfluss auf die Struktur der neuen MVC-Architektur

hat. Weil sich die Komponenten alle untereinander Nachrichten senden, muss für jede Beziehung zwischen den Komponenten eine entsprechende Schnittstelle vorhanden sein. Die Schnittstellen werden in zwei Fällen als Observer realisiert und in einem Fall als Facade.

Dependency Inversion wird in allen drei Komponenten berücksichtigt. Es werden Abstraktionen konstruiert, wodurch Abhängigkeiten von Implementierungen vermieden werden.

8. REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[2] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object Oriented Programming*, August/November 88, 1988.

[3] R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices: Pearson New International Edition*. Pearson, 2013.

8.1 Glossary

SOLID Akronym für die Design-Prinzipien Single Responsibility, Open-Closed, Liskovsches Substitutionsprinzip, Interface Segregation und Dependency Inversion

Model Das 'M' in MVC; das Anwendungsobjekt dessen Daten in der Ansicht dargestellt werden und das durch Benutzereingaben manipuliert wird

View Das 'V' in MVC; das Ansichtsobjekt, das Daten der Anwendung darstellt und Steuerelemente zur Verfügung stellt

Controller Das 'C' in MVC; das Steuerungsobjekt, das Benutzereingaben validiert und das Verhalten des Ansichtsobjektes steuert