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

Stefan Berger

23. April 2018

Inhaltsverzeichnis

1 Einleitung **2**

2 Architekturmuster und -prinzipien **2**

2.1 MVC 2

2.2 SOLID 3

3 Logik und Datenmodell der Ansicht **4**

3.1 Single Responsibility 4

3.2 Open-Closed 5

3.3 Liskovsches Substitutionsprinzip 5

3.4 Interface Segregation 6

3.5 Dependency Inversion 6

4 Gültigkeitsprüfung für Eingaben **6**

4.1 Interpretation und Weiterleitung der Benutzereingaben 6

Glossar **I**

Zusammenfassung

1 Einleitung

Seit der Entwicklung des Architekturmusters Model-View-Controller (MVC) hat sich in den Programmentwürfen für Benutzerschnittstellen die Trennung von Daten und Ansicht 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 bearbeitet werden, ohne dass sich die Programmierer gegenseitig behindern.

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.

2 Architekturmuster und -prinzipien

2.1 MVC

Das Programmierparadigma MVC [KraPo88, S. 26–49] sieht für interaktive Anwendungen die Trennung in ein Modul für die Programmlogik (Model), ein Ansichtsmodul (View) und ein Steuerungsmodul (Controller) vor. Jedes dieser Module steht mit jedem anderen in einer bestimmten Relation.

Das Model-View-Controller-Entwurfsmuster basiert auf diesem Programmierparadigma. Die Relationen der drei Module werden durch verschiedene Entwurfsmuster realisiert. Die View als Observer [GoF94, S. 293–305] des Models wird über Änderungen am Model benachrichtigt. Benutzereingaben im Ansichtsmodul werden vom Controller interpretiert. In Form des Strategy-Patterns [GoF94, S. 315–325] wird der View ein Controllermodul zugewiesen. Die Zuweisung kann sich zur Laufzeit ändern, wenn gleiche Benutzereingaben aufgrund geänderter Zustände unterschiedlich interpretiert werden müssen. Der Controller benachrichtigt das Model gegebenenfalls über Benutzereingaben, sodass diese entsprechend verarbeitet werden können. Einer View können also mehrere Controller zugewiesen werden und umgekehrt. Das Model ist normalerweise einzigartig. Views und Controller kennen also jeweils nur ein Model.

Mehrere Views können außerdem untereinander in Relation stehen. Mit dem Composite-Pattern [GoF94, S. 163–175] können Teilkomponenten der Ansicht zum Ansichtsmodul zusammengefasst werden. Sowohl das Ansichtsmodul als auch die Teilkomponenten können dann auf Änderungen am Model reagieren.

2.2 SOLID

Die SOLID-Prinzipien sind

- Single Responsibility
- Open-Closed
- Liskovsches Substitutionsprinzip (Ersetzungsprinzip)
- Interface Segregation
- Dependency Inversion

Das Single-Responsibility-Prinzip besagt, dass es *für eine Klasse nur einen Grund zur Änderung* geben sollte [Marti13, 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 Ansichtsmodul 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, wenn sichergestellt werden soll, dass jede Klasse eine einzige Verantwortlichkeit besitzt.

Open-Closed sind *Klassen, Module, Funktionen etc., die für Erweiterungen offen, aber für Modifikationen verschlossen sind* [Marti13, S. 99]. Eine Hauptaufgabe von Controllerklassen ist es, Benachrichtigungen entgegenzunehmen und weiterzuleiten. Im einfachsten Fall sollen Benutzereingaben nur an das Anwendungsmodul übergeben werden. Ein komplizierteres Beispiel ist das Austauschen der View auf Anweisung des Anwenders. Weil das Zusammenspiel mit den anderen Klassen davon abhängt, dass der Controller diese Aufgabe wahrnimmt, muss die Controllerklasse für Änderungen an den jeweiligen Schnittstellen verschlossen sein. Im Gegensatz dazu müssen verschiedene Controller-Implementierungen dieselbe Aufgabe unterschiedlich ausführen. Die Klasse muss deshalb für solche Erweiterungen offen sein.

Vereinfacht gesagt schreibt das Liskovsche Substitutionsprinzip vor, dass *Obertypen durch Untertypen ersetzbar* sein sollen [Marti13, S. 111]. Angenommen, einer von mehreren Teilkomponenten des Ansichtsmoduls 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 Ansichtsklasse von der bestehenden abzuleiten. Bei Bedarf kann dann ein Objekt der bestehenden Klasse durch ein Objekt der neuen Klasse ersetzt werden. Wird das Liskovsche Substitutionsprinzip befolgt, kann das Objekt der neuen Klasse verwendet werden, ohne das Ansichtsmodul zu verändern. Ein wichtiger Aspekt ist dabei das neue Datenfeld, das dem Ansichtsmodul unbekannt ist.

Interface Segregation bedeutet, dass eine Klasse dem Programmierer genau die Schnittstelle zur Verfügung stellt, die für den jeweiligen Zweck vorgesehen ist. In MVC verwenden die View und der Controller jeweils die Schnittstelle 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 besitzen sollten.

Das Dependency-Inversion-Prinzip schreibt vor, dass Module nicht von anderen Modulen, die sich niedriger in der Modulhierarchie befinden, abhängig sein sollten [Marti13, S. 127]. Wir werden dieses Prinzip beim Design des Ansichtsmoduls und seiner Teilkomponenten berücksichtigen.

3 Logik und Datenmodell der Ansicht

3.1 Single Responsibility

Ein Ansichtsmodul 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.

Interaktive Benutzeroberflächen mit Steuerelementen, die selbst von Benutzereingaben abhängig sind, besitzen eigene Zustände. Es würde keinen Sinn machen, diese Zustände im Datenmodell der Anwendungslogik zu speichern. Deshalb erhält unsere View außerdem ein eigenes Datenmodell und eine Klasse für die Implementierung aller dynamischer Aspekte der View. 1 zeigt die Klassenstruktur

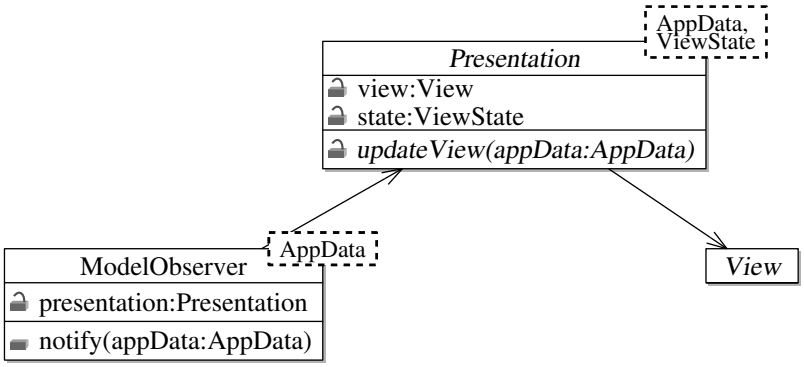

Abbildung 1: Ansichtsmodul mit Präsentationslogik

des Ansichtsmoduls. Die Klasse `ModelObserver` behandelt die Benachrichtungen über Änderungen im Datenmodell der Anwendung, indem sie ein Attribut der Klasse `PresentationState` synchronisiert. In `PresentationState` werden außerdem die Zustände von Steuerelementen gespeichert. In der Klasse `Presentation` befinden sich Methoden, mit denen die Ansicht abhängig von den Zustandsattributen dynamisch aktualisiert werden kann. Zum Beispiel könnte das Datum der letzten Aktualisierung angezeigt werden. 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.

3.2 Open-Closed

Das Ansichtsmodul ist für Erweiterungen an seinem Datenmodell und der Präsentationslogik offen. Wir werden später Zustandsattribute hinzufügen, um ungespeicherte Benutzereingaben hervorzuheben. Bis auf eine Ausnahme sind alle Klassen des Ansichtsmoduls abstrakt. Die Klasse `ModelObserver` ist keiner anderen Klasse bekannt und ist deshalb vollständig implementiert. Es sind keine Änderungen am Ansichtsmodul nötig, um neue Views zu implementieren. Die Vorgaben des Open-Closed-Prinzips sind eingehalten.

3.3 Liskovsches Substitutionsprinzip

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

Nennen wir die Klasse, die wir von `Presentation` ableiten, `Form`. Ein Formular kann Pflichtfelder enthalten, und Formularfelder können bereits mit einem Wert gefüllt sein. Diese Informationen werden vom Model geliefert und sind über das Attribut `state`, das die neue Klasse erbt, verfügbar. `Form` erhält zwei neue Methoden. Eine Methode füllt Formularfelder mit Standardwerten und die zweite markiert noch ungespeicherte Änderungen. Die geerbte abstrakte Methode `updateView` wird implementiert, um die beiden neuen Methoden aufzurufen. Die von `PresentationState` abgeleitete Klasse nennen wir `FormState`. Sie erhält für jedes Formularfeld ein Flag, das eine Änderung an dem Wert des jeweiligen Felds anzeigt. Von der Klasse `View` leiten wir ebenfalls eine Klasse ab, sie bekommt den Namen `FormView` und Attribute für die Formularfelder. Bei jeder Änderung wird im Objekt der Klasse `FormState` das Änderungs-Flag des jeweiligen Formularfelds gesetzt und geänderte Formularfelder werden hervorgehoben.

Wir fügen den drei Klassen `Form`, `FormState` und `FormView` jeweils noch eine Erweiterung hinzu, indem wir von jeder der drei Klassen wieder eine neue Klasse ableiten. Wir nennen die Klassen `ExtForm, ExtFormState` und `ExtFormView`. Mit ihnen wird ein erweitertes Formular realisiert, in dem bestimmte Formularfelder abhängig von einem Radiobutton ein- oder ausgeblendet werden. Das Attribut `appData` erhält Optionen für den Radiobutton und für jedes Formularfeld eine Zuordnung zu einer Radiobutton-Option. `ExtForm` erhält eine Methode, die Formularfelder ein- und ausblendet. Die Methode `updateView` wird noch einmal überschrieben, um die Basisklassenversion und die neue Methode aufzurufen. In `ExtFormState` wird in einem zusätzlichen Attribut die Information gespeichert, ob es versteckte Formularfelder mit ungespeicherten Änderungen gibt, beziehungsweise welchen Radiobutton-Optionen diese zugeordnet sind. In `ExtFormView` werden diese Radiobutton-Optionen hervorgehoben.

Objekte der Klassen `ModelObserver` und `Form` funktionieren zusammen mit Objekten der Klassen, die direkt von den abstrakten Basisklassen abgeleitet sind, genauso wie mit Objekten der Ext-Klassen, die von den konkreten Implementierungen erneut abgeleitet wurden. Die Forderung des Liskovschen Substitutionsprinzips ist erfüllt.

3.4 Interface Segregation

Ein Konstruktorparameter für die Klasse `ModelObserver` ist nötig, damit das Interface-Segregation-Prinzip eingehalten ist. Der Parameter gibt an, über welche Änderungen der Observer benachrichtigt werden soll. Je nach Zielsytem und verwendeter Programmiersprache kann der Parametertyp zum Beispiel eine Klasse, eine Enum oder eine URL sein.

3.5 Dependency Inversion

Weil sie die Geschäftslogik der Ansicht enthalten, stellen die Klassen, die direkt oder indirekt von `Presentation` abgeleitet sind, die «High Level Modules» 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 in ihrer Gesamtheit beeinträchtigen. Wie schon beim Open-Closed-Prinzip ermöglicht Abstraktion, diese Anforderung zu implementieren. Keins der Attribute der Klasse `Presentation` ist von einem konkreten Typ, und die Klasse besitzt keine konkreten Methoden. Die Implementierungsdetails befinden sich in abgeleiteten Klassen, ohne die ungwünschte Abhängigkeit zu erzeugen.

4 Gültigkeitsprüfung für Eingaben

4.1 Interpretation und Weiterleitung der Benutzereingaben

Der Controller in einer MVC-Architektur hat zwei verschiedene Aufgaben. Er interpretiert Benutzereingaben, überprüft zum Beispiel Pflichtfelder eines Formulars, und benachrichtigt gegebenenfalls das Model und die View über relevante Benutzereingaben.

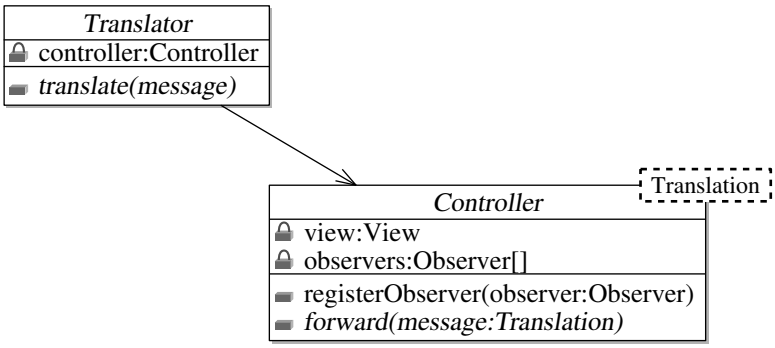

Abbildung 2: Controllermodul mit Eingabeüberprüfung

Das Controllermodul besteht aus einer Observer-Implementierung für die View, einer Klasse für die Prüfung der Benutzereingaben, einem DTO für das Prüfungsergebnis und der Controllerklasse, wie in Abbildung 2 zu sehen ist. Die Aufgabe der Klasse `Controller` ist es, das Model über Benutzereingaben zu benachrichtigen. Das Single-Responsibility-Prinzip ist mit dieser Klassenstruktur eingehalten.

Literaturverzeichnis

[GoF94] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*.

[KraPo88] Glenn E. Krasner and Steven T. Pope. “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80”. In: *Journal of Object Oriented Programming* August/November 88 ().

[Marti13] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices: Pearson New International Edition*. Pearson, 2013.

https://glossar.hs-augsburg.de/Model-View-Controller-Paradigma (status- oder fehlermodell)

Glossar

Controller Das ’C’ in MVC; das Steuerungsobjekt, das Benutzereingaben validiert und das Verhalten des Ansichtsobjektes steuert 2

Model Das ’M’ in MVC; das Anwendungsobjekt dessen Daten in der Ansicht dargestellt werden und das durch Benutzereingaben manipuliert wird 2

SOLID Akronym für die Design-Prinzipien Single Responsibility, Open-Closed, Liskovsches Substitutionsprinzip, Interface Segregation und Dependency Inversion 2

View Das ’V’ in MVC; das Ansichtsobjekt, das Daten der Anwendung darstellt und Steuerelemente zur Verfügung stellt 2