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.

Die Prinzipien werden einzeln für jede der drei Komonenten Model, View und Controller ausgewertet. Es sollen die Auswirkungen auf die Struktur einer MVC-Anwendung und die Anwendbarkeit bei der Entwicklung untersucht 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 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 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 in den Abschnitten 2.1 bis 2.4 zuerst die drei Komponenten des MVC-Architekturmusters und anschließend in den Abschnitten 3.1 bis 3.5 die fünf SOLID-Architekturprinzipien genau betrachten. Die drei Komponenten des MVC-Architekturmusters werden wir in den Abschnitten 4 bis 6 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 Ansichtsmodul (View) und ein Steuerungsmodul (Controller) vor [3, S. 26-49]. Jedes dieser Module steht mit jedem anderen in einer bestimmten Relation.

2.2 Model

Im Model kann ein reines Datenmodell sein [3, S. 27] oder das Verhalten der Anwendung und das Datenmodell [1, S. 2]. Ein typisches Szenario ist eine Interaktion des Benutzers mit der View, die durch den Controller an das Model vermittelt wird. Bei jeder Änderung der Daten im Model benachrichtigt es die View. 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. Model und View können jedoch mehrere verschiedene Datenobjekte austauschen [6, S. 14].

2.3 View

Eine Ansicht ist eine visuelle Repräsentation eines Models [5, S. 1]. Die Relationen der drei Module werden durch verschiedene Entwurfsmuster realisiert. Die View als Observer des Models [2, 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 [2, 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 [2, S. 315—325] wird der View ein Controller zugewiesen. Eine View kann einen oder mehrere Controller besitzen. Normalerweise werden aber sogenannte View-Controller-Paare erzeugt, also eine View mit einem Controller. Der Controller der View kann sich jederzeit ändern, auch zur Laufzeit. Controller können außerdem die Fähigkeit besitzen, andere Views aufzurufen. Der Controller benötigt dann Zugriff auf entsprechende Schnittstellen der Laufzeitumgebung. Ein Controller muss außerdem das Datenmodell der Ansicht manipulieren und diese aktualisieren können.

3. SOLID

3.1 Single Responsibility

Das Single-Responsibility-Prinzip besagt, dass es *für eine Klasse nur einen Grund zur Änderung geben sollte* [8, S. 63]. 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* [7, 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 [7, 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 sein muss.

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.

Im MVC-Architekturmuster verwenden sowohl die View als auch der Controller Schnittstellen des Models. Die View muss als Empfänger für Benachrichtigungen über Änderungen registriert werden. Der Controller muss das Model über Benutzereingaben benachrichtigen. Daraus ergeben sich zwei verschiedene Schnittstellen zum Model. Gemäß des Interface-Segretation-Prinzips sollten die View und der Controller auch zwei verschiedene Schnittstelle erhalten.

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 [7, S. 127].

Wir werden dieses Prinzip bei der Implementierung des Viewmoduls und seiner Teilkomponenten berücksichtigen.

4. SOLID VIEW

4.1 Single Responsibility im Viewmodul

Ein Viewmodul beinhaltet wenig bis keine Programmlogik. Es reagiert auf Änderungen an den Anwendungsdaten und macht diese sichtbar.

Für Benachrichtigungen über Änderungen ist das Observer-Entwurfsmuster vorgesehen. Ein Observer hat eine Verantwortlichkeit im Sinne des Single-Responsibility-Prinzips. Die Registrierung am Model und der Callback sollten sich deshalb in einer eigenen Klasse befinden.

Abbildung 1 zeigt die Klassenstruktur des Viewmoduls. Die Klasse `DateTimeObserver` erhält Benachrichtungen über Änderungen im Datenmodell der Anwendung und leitet sie an Objekte der Klasse `DateTimeView` 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 im Viewmodul

Die Klasse `DateTimeObserver` ist für Modifikationen am zugrundeliegenden Observer-Entwurfsmuster verschlossen. Um zum Beispiel mehrere Ansichten über Änderungen am Modul zu informieren ist sie Erweiterungen am Benachrichtigungsmechanismus sie dagegen offen.

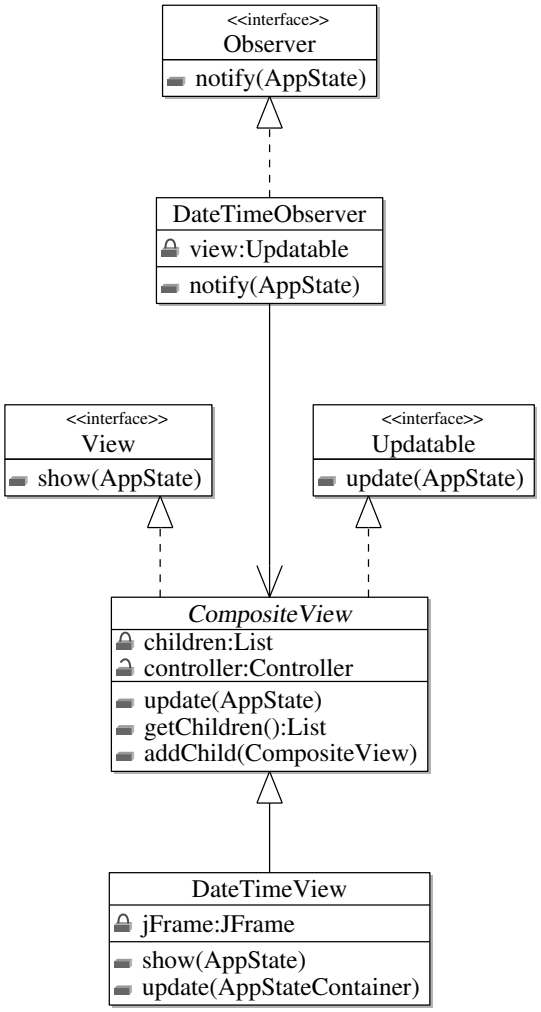

Figure 1: Klassendiagramm des Viewmoduls

Die Klasse `DateTimeView` ist für Erweiterungen an ihrer Präsentationslogik offen. Die Methoden `show` und `update` können zu diesem Zweck überschrieben werden. Die Implementierung des Composite-Entwurfsmusters der abstrakten Basisklasse `CompositeView` ist für Modifikationen verschlossen.

Es sind keine Änderungen am Ansichtsmodul nötig, um neue Ansichten zu implementieren. Die Vorgaben des Open-Closed-Prinzips sind eingehalten.

4.3 Liskovsches Substitionsprinzip im Viewmodul

Das Viewmodul des Grundgerüstes besitzt keine Vererbungshierarchie. Für die Erweiterungen des Datenmodells und der Präsentationslogik werden von den bestehenden Klassen neue abgeleitet.

Nennen wir eine Klasse zur Verwaltung unserer Ansichten, welche die Interfaces `View` und `Updatable` implementiert, `CompositeView`. Eine Ansicht besteht aus verschiedenen Elementen, die alle bei Aktualisierungen der Daten benachrichtigt werden müssen. Das Programm übergibt Informationen darüber, welche Daten dargestellt werden sollen, an die Methode `update` des Interfaces `Updatable` im Parameter `appState`. `CompositeView` erhält zwei weitere Methoden `addChild` und `getChildren`, mit denen Unteransichten organisiert werden können.

Wir fügen der Klasse `CompositeView` eine Erweiterung hinzu, indem wir eine neue Klasse ableiten. Wir nennen die Klasse `DateTimeView`. Mit ihr werden das Datum und die Uhrzeit dargestellt, indem jeweils eine Unteransicht hinzugefügt wird. Für die Unteransichten werden die Klassen `DateView` und `TimeView` von `CompositeView` abgeleitet. Die Methode `update` in `DateTimeView` bekommt im Parameter `appState` die Bestandteile des Datums und der Uhrzeit als Ganzzahlen übergeben. Die Methode wird überschrieben, um die Basisklassenversion aufzurufen, in der die Unteransichten ebenfalls aktualisiert werden. In `appState` sind in einer generischen Map die Daten der verschiedenen Ansichten, also Datum und Uhrzeit, separat abgelegt. Dadurch behält der Parameter aller `update`-Methoden der verschiedenen Ansichtsklassen denselben Typ.

Objekte der Klasse `DateTimeObserver` funktionieren zusammen mit Objekten der Klasse `DateTimeView`, die direkt von der abstrakten Basisklasse abgeleitet ist, genauso wie mit Objekten der Klassen `DateView` und `TimeView`, die von der konkreten Implementierung erneut abgeleitet wurden. Die Forderung des Liskovschen Substitutionsprinzips ist erfüllt.

4.4 Interface Segregation im Viewmodul

Das Interface `View` stellt die Methode `show` zur Verfügung, um die Ansicht initial darzustellen. Ein Objekt der Klasse `DateTimeObserver` erhält eine Referenz auf die Hauptansicht in Form eines Attributes vom Typ `Updatable`. Die Klasse `CompositeView` implementiert die einzige Methode `update` dieses Interfaces und die Methode `show`. Der Implementierung des Interfaces `Clock`, welche die eigentliche Arbeit verrichtet, ist zu diesem Zweck ebenfalls nur die Methode `update` des Interfaces `Observer` bekannt. 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 das Interface `View` implementieren, 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 erfüllen. Alle Attribute der Klasse `CompositeView` und der von `CompositeView` abgeleiteten Klassen sind vom Typ eines Interfaces. 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-Anwendung hat zwei Aufgaben. Er interpretiert Benutzereingaben und er benachrichtigt andere Module über relevante Benutzereingaben. Hierfür hält er gegebenenfalls Zustandsdaten der Ansicht vor.

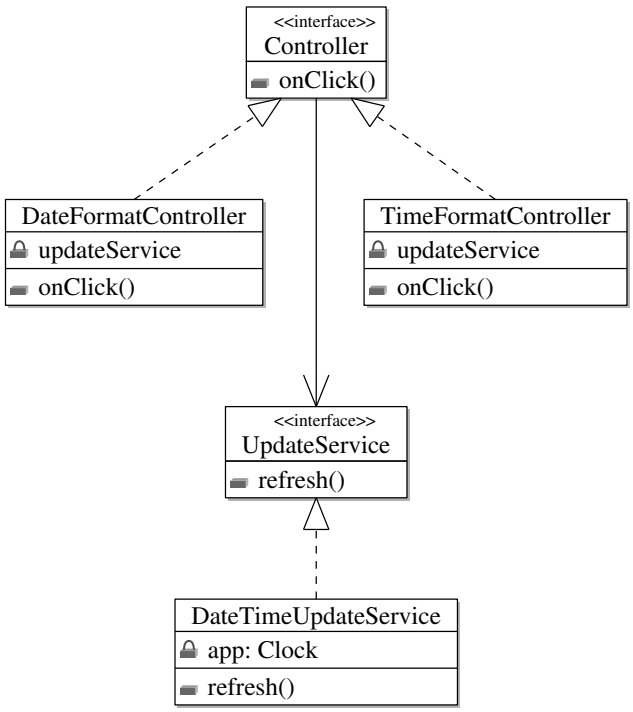

Figure 2: Klassendiagramm des Controllermoduls

Im Beispiel der Ansichtsklasse `DateTimeView` besitzen die beiden Unteransichten `DateView` und `TimeView` jeweils einen Controller. Jeder der beiden Controller besitzt ein Attribut, in dem Benutzereinstellungen zum Format der Datums- oder Zeitanzeige vorgehalten werden und ein Attribut vom Typ `UpdateService`, mit dem die Neuberechnung der anzuzeigenden Daten im Fall einer entsprechenden Änderung der Einstellungen durch den Benutzer angestoßen werden kann.

Dafür ist eine Methode `getMouseEventHandler` im Interface `Controller` festgelegt, die einen Event-Handler für die jeweilige Ansichtsklasse erzeugt und zurückgibt. Die Klassen `DateController` und `TimeController` implementieren diese Methode ihren Anforderungen entsprechend.

Durch die Aufteilung der zwei Aufgaben auf die zwei verschiedenen Interfaces `Controller` und `UpdateService` kann es für Änderungen an deren Implementierungen immer nur einen einzigen Grund geben. Das Single-Responsibility-Prinzip also eingehalten.

5.2 Open-Closed im Controllermodul

Im Controllermodul befindet sich zwei Implementierungen des Interfaces `Controller`. Sie erstellen Event-Handler für die Ansichtsklassen. Sie besitzen jeweils ein Attribut vom Typ `UpdateService` und sind für Änderungen an diesem Attribut verschlossen.

Die Methode `getMouseEventHandler`, in der das Verhalten bei Klicks des Benutzers definiert wird, ist für Änderungen offen.

Die Klasse `DateTimeUpdateService` besitzt zwei unveränderliche Attribute, und zwar die Schnittstellen zur Ansicht und zur Hauptanwendung. Diese Methode ist als nicht überschreibbar deklariert, 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 Implementierungen der Interfaces, sind die Forderungen des Liskovschen Substitutionsprinzips einzuhalten.

Um dem Anwender die Möglichkeit zu bieten, Datum und Uhrzeit per Drag-and-Drop im Anwendungsfenster zu verschieben, kann eine Implementierung des Interfaces `Controller` erstellt werden, die statt einem `MouseListener` einen `MouseInputListener` zurückgibt. Eine Spezialisierung des Typs der Interface-Methode ist mit dem Liskovschen Substitutionsprinzip vereinbar, eine Basisklasse dürfte hier nicht verwendet werden. In der Parameterliste ist es umgekehrt, dort dürften die Typen nur in allgemeinere geändert werden.

In der Schnittstelle der Klasse `Controller` sind keine Übergabeparameter deklariert. Das Liskovsche Substitutionsprinzip ist eingehalten, wenn die Schnittstelle wie beschrieben oder gar nicht verändert wird.

5.4 Interface Segregation im Controllermodul

Das Controllermodul ist außer der Hauptanwendung nur dem Ansichtsmodul bekannt. Die Ansicht nutzt die Methode `getMouseEventHandler` des Interfaces `Controller`, um auf Benutzereingaben zu reagieren.

Die beiden Implementierungen des Interfaces `Controller` besitzen als Attribut einen `UpdateService`, der für Nachrichten an die Hauptanwendung über Änderungen durch den Benutzer verantwortlich ist.

Die Klasse `UpdateService` erhält eine Schnittstelle der Hauptansicht vom Typ `Clock`, denn sie benötigt nur eine Methode, um die Anwendung zu veranlassen, den `appState` neu zu berechnen. Durch diese Trennung der Verantwortlichkeiten sind die Anforderungen des Interface-Segregation-Prinzips auch im Controllermodul erfüllt.

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 `Controller` von `UpdateService`. Weil `UpdateService` 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 unbestimmte Anzahl Notizen im Arbeitsspeicher vor. Eine Anwendung für Online-Banking berücksichtigt dagegen vor allem Sicherheitsaspekte und kann erfolglose Transaktionen rückabwickeln.

Das Model ist ein generischer Container für die Daten, die in der Ansicht dargestellt werden sollen. Dadurch ist es unter anderem möglich, der Ansicht die verschiedenartigen Datenstrukturen der verschiedenen Unteransichten in einem einzigen Aufruf zu übermitteln.

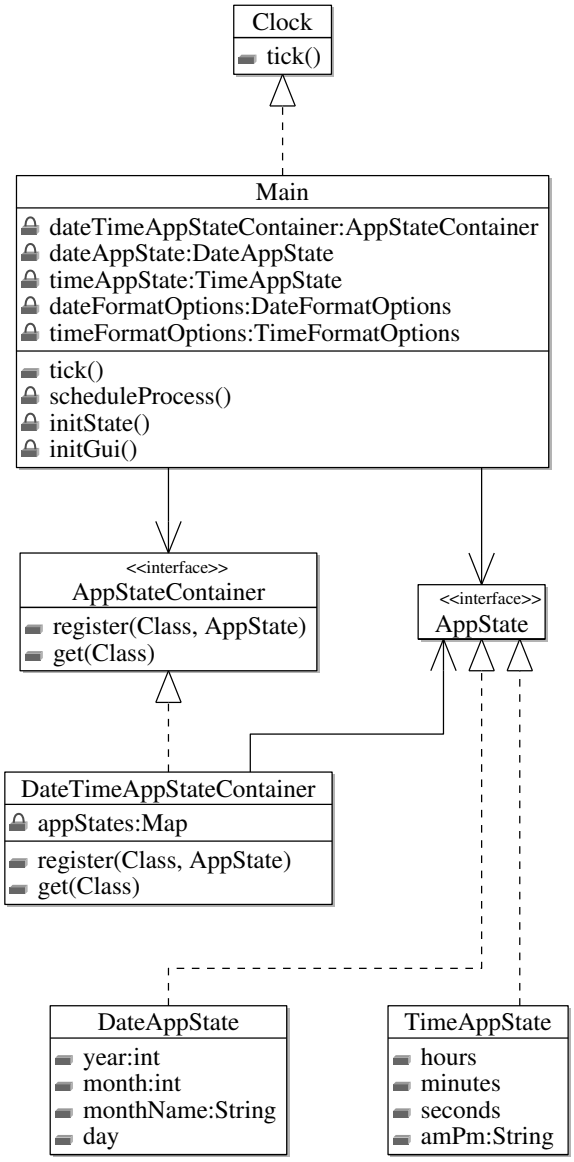

Figure 3: Klassendiagramm des Models

Das Model im MVC-Architekturmuster besitzt bereits die einzige Verantwortlichkeit, der Ansicht aktuelle Daten bereitzustellen. Wird diese Verantwortlichkeit mittels Observer-Entwurfsmuster realisiert, ist das Single-Responsibility-Prinzip eingehalten.

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. Die sekündliche Aktualisierung der Daten ist ebenfalls für Modifikationen verschlossen. Dasselbbe gilt für die Benachrichtigungen des Observers im Ansichtsmodul.

Die Berechnung der Daten ist für Erweiterungen offen. Denkbare Erweiterungen sind die Berechnung der Weltzeit und länderspezifische Datums- und Zeitformate. Das Open-Closed-Prinzip ist in der DateTime-Anwendung eingehalten.

6.3 Liskovsches Substitutionsprinzip im Model

Das Model der DateTime-Anwendung ist in einer flachen Klassenhierarchie gehalten. Das Substitutionsprinzip wird deshalb für das Model auch nicht verletzt.

Theoretisch denkbar wäre eine Klasse, die beispielsweise statt nur einer Uhrzeit die Weltzeit an die Ansicht übermittelt. Eine Spezialisierung dieser Klasse könnte die Zugriffsmethoden überschreiben, die in der Basisklasse für ein Objekt einer Zeitzonenkonfigurationsklasse, bestehend aus Land und Stadt, die Uhrzeit in einem Objekt einer ebenfalls hierfür erstellten Klasse zurückgibt. In der Überschreibung könnten zusätzlich die jährlichen Zeitumstellungen berücksichtigt werden. Um das Liskovsche Substitutionsprinzip einyuhalten dürfte hierfür der Parameter der Zugriffsmethode nicht in einen spezielleren Typ geändert werden. Der Rückgabetyp dürfte nicht in einen allgemeineren Typ geändert werden.

6.4 Interface Segregation im Model

Das Model ist dem Controllermodul in Form des Interfaces `Clock` bekannt. Das Interface bietet dem Controller durch die Methode `tick` die Möglichkeit, die berechneten Daten zu aktualisieren.

Das Model kennt nur den Observer des Ansichtsmoduls. Das Interface `Observer` stellt die Methode `notify` bereit, mit welcher die Ansicht über geänderte Daten benachrichtigt wird.

Die Ansicht kennt das Datenmodell. Es besteht aus reinen Datenobjekten für Datum und Uhrzeit und einer Map zur Organisation der Daten nach Unteransicht. Mit dieser Trennung der Schnittstellen nach den jeweiligen Anforderungen der Komponenten wird das Interface-Segregation-Prinzip berücksichtigt.

6.5 Dependency Inversion im Model

Das Model ist das Modul der DateTime-MVC-Anwendung mit den meisten Abhängikeiten. Es erstellt die Ansichten und die zugehörigen Controller, erlaubt dem Controller das Aktualisieren der Daten und stellt der View das Datenmodell in einem generischen Container bereit.

Für alle diese Abhängigkeiten zum und vom Model wurden Interfaces genutzt. Das Dependency-Inversion-Prinzip ist dadurch eingehalten.

7. EXPERIMENT

Um das Grundgerüst auf seine Praxistauglichkeit zu testen wurde die DateTime-Anwendung entwickelt. Sie zerlegt die Systemzeit in ganzzahlige Bestandteile, um sie als Datenmodell den Ansichten zur Verfügung zu stellen.

Im untenstehenden Listing ist zu erkennen, dass eine Datenmodellklasse für den Datumsteil und eine für den Uhrzeitteil existiert. Die beiden Klassen entsprechen den beiden Unteransichten im Ansichtsmodul.

Die Unteransichten besitzen jeweils einen Controller. Bei Klick auf eine der Ansichten werden die Formatoptionen geändert, deren Klassen ebenfalls im untenstehenden Listing verwendet werden. Die Objekte für die Formatoptionen werden im Controller referenziert. Dort werden die Eigenschaften in den Ereignisbehandlungsroutinen geändert und im Model wird die Neubberechnung des Datums und der Uhrzeit veranlasst. Schließlich wird der Observer im Ansichtsmodul über die Änderungen informiert.

```
1  public void tick() {
2    Calendar calendar =
3      GregorianCalendar.getInstance();
4
5    dateAppState.year =
6      calendar.get(Calendar.YEAR);
7    dateAppState.month =
8      calendar.get(Calendar.MONTH);
9    dateAppState.day =
10     calendar.get(Calendar.DATE);
11   if (dateFormatOptions.longMonth) {
12     dateAppState.month = -1;
13     dateAppState.monthName =
14       Month.of(calendar.get(Calendar.MONTH))
15         .toString();
16   } else {
17     dateAppState.month =
18       calendar.get(Calendar.MONTH);
19     dateAppState.monthName = "";
20   }
21
22   timeAppState.seconds =
23     calendar.get(Calendar.SECOND);
24   timeAppState.minutes =
25     calendar.get(Calendar.MINUTE);
26   if (timeFormatOptions.twentyFourHours) {
27     timeAppState.amPm = "";
28
29     if (calendar.get(Calendar.AM_PM) ==
30       Calendar.PM) {
31         timeAppState.hours =
32         calendar.get(Calendar.HOUR) + 12;
33     }
34   } else {
35     timeAppState.hours =
36       calendar.get(Calendar.HOUR);
37     timeAppState.amPm =
38       calendar.get(Calendar.AM_PM) ==
39         Calendar.AM ? "am" : "pm";
40   }
41
42   appStateContainer.registerClass(
43     DateAppState.class, dateAppState);
44   appStateContainer.registerClass(
45     TimeAppState.class, timeAppState);
46
47   observer.notify(appStateContainer);
48 }
```

Listing 1: Das Model der Anwendung

8. FAZIT

Das Einhalten der SOLID-Prinzipien beim Entwurf einer MVC-Architektur führt zu komplexen Klassendiagrammen der einzelnen Komponenten. Dafür entkoppelt es die Komponenten und auch die Unterkomponenten und ermöglicht eine einfache Erweiterung der verschiedenen Module.

Durch Abstraktionen wird der Controller beispielsweise vom Service zum Aktualsieren der Modeldaten entkoppelt. Die Austauschbarkeit der Kompenenten wird dadurch erhöht, und die Abhängigkeiten werden auf ein Minimum reduziert [4, S. 2].

Das Model wurde durch die neue Architektur besonders stark umgestaltet. Im Einsprungspunkt in der Hauptklasse werden zunächst alle MVC-Module initialisiert. Die Controller werden an die Ansichten und die Ansichten an das Model gekoppelt, und das Datenmodell wird auf die Übertragung an die Ansicht vorbereitet.

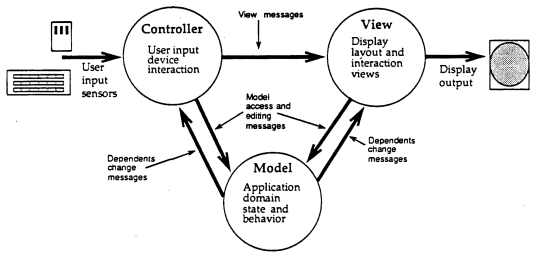

Figure 4: Typisches MVC-Diagramm [3, S. 27]

In der Ansicht ist nun der Observer für die Modeldaten von dem Konstrukt des Composite-Patterns für die Haupt- und Unteransichten völlig losgelöst. Die Ansichtsklassen implementieren zwei verschiedene Interfaces, um dem Observer und dem Composite-Pattern jeweils die Richtige Schnittstelle anzubieten.

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 (siehe Abbildung 4).

Durch die Beschränkung einer Klasse auf genau eine Verantwortlichkeit erweitert sich das Model auf mindestens zwei Klassen für Nachrichten des Controllers und das Datenmodell. Aufgrund der anderen SOLID-Prinzipien sind es sogar drei Interfaces.

Die Ansicht besteht wegen des Single-Responsibility-Prinzips mindestens aus einer Observerklasse für Modeldaten und einer Klasse für das Composite-Pattern. Wie auch beim Model sind es in Wirklichkeit drei Interfaces.

Das Controllermodul besteht nun aus zwei Unterkomponenten, nämlich aus dem Controller für die Ereignisbehandlung der Ansicht und dem UpdateService für die Kommunikation mit dem Model.

Die Gesamtarchitektur besteht nun aus sechs statt vorher drei (Unter-)Komponenten.

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

Die DateTime-Anwendung erstellt keine Views ohne Controller und tauscht den Controller einer View auch nie aus. Diese Eigenschaften können jedoch implementiert werden, ohne das Grundgerüst zu verändern.

Das Composite-Pattern ist fest in der Ansicht verankert. Das Datenmodell ist flexibel genug, um jeder Unteransicht eine eigene Klasse für Daten zu ermöglichen, und diese Eigenschaft ist ebenfalls verschlossen für Modifikationen.

Wir haben das Liskovsche Substitutionsprinzip beim Entwurf des Datenmodells berücksichtigt. Es ist ein generischer Container für alle denkbaren Datenmodelle und muss deshalb nicht spezialisiert zu werden. Dadurch kann das Model in jeder abgeleiteten Viewklasse unverändert als Übergabeparameter verwendet werden.

Das Grundgerüst besitzt ansonsten nur eine flache Hierarchie, die aus Interfaces und wenigen Klassen besteht. Diese kann das Liskovsche Substitutionsprinzip nicht verletzen. Das Prinzip kann aber in Anwendungen berücksichtigt werden, die das Grundgerüst nutzen und die Interfaces implementieren.

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.

Das Model definiert eine Schnittstelle für Nachrichten des Controllers. Der Container für die Modeldaten ermöglicht es, die Ansicht über Änderungen an einem oder mehreren konkreten Datenmodellen zu benachrichtigen.

Im Ansichtsmodul ist der Observer der Modeldaten angesiedelt. Ihm ist das Interface bekannt, das es ermöglicht eine Ansicht zu aktualisieren.

Jede Ansicht im Composite-Pattern kann Kindelemente als Unteransichten besitzen. Dafür ist in der abstrakten Klasse eine Methode zum Hinzufügen von Ansichten vorhanden.

Das Model erhält vom View-Interface schließlich eine Methode zum Anzeigen einer Ansicht, ohne dass vorher eine Aktualisierung der Daten stattgefunden haben muss.

Dependency Inversion wird ebenfalls in allen drei Komponenten berücksichtigt. Es werden Abstraktionen konstruiert, wodurch Abhängigkeiten von Implementierungen vermieden werden. Durch Programmmierung gegen Interfaces kann dieses Prinzipien leicht eingehalten werden.

9. REFERENCES

[1] S. Burbeck. Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc), 1987.

[2] 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.

[3] 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.

[4] B. Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, March 1987.

[5] T. M. H. Reenskaug. Models-views-controllers, December 1979.

[6] T. M. H. Reenskaug. The model-view-controller (mvc) – its past and present, August 2003. Präsentation über Vergangenheit und Zukunft von MVC.

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

[8] R. C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series. Prentice Hall, Boston, MA, 2017.

9.1 Glossary

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

MVC Das Model-View-Controller-Architekturmuster

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

Strategy Pattern Das Strategy-Entwurfsmuster ermöglicht Austauschbarkeit von Klassen

Observer Pattern Das Observer-Entwurfsmuster ermöglicht die lose Kopplung mehrerer Klassen oder Komponenten zur Nachrichtenübermittlung

Composite Pattern Das Composite-Entwurfsmuster ermöglicht eine baumartige Struktur von Objekten wie beispielsweise eine Ansicht bestehend aus verschachtelten Fenstern oder Steuerelementen.