FHWS

Hochschule für angewandte Wissenschaften
Fachhochschule Würzburg-Schweinfurt

FWPM - Designpatterns

Das MVC und Derivate, Oberserverpattern als Ergänzung

vorgelegt an der
Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt
in der Fakultät Informatik und Wirtschaftsinformatik

bei Professor:	Prof. Eberhard Grötsch
Abgabetermin:	23. Mai 2013
Eingereicht von:	Florian Beckh 5110006
	Christian Petry 5110069
	Johannes Krenig 5109038

Inhaltsverzeichnis

1 Observer

Das Observer Pattern gehört zu der Kategorie Verhaltensmuster. Mit ihm ist es möglich Änderungen von einem Objekt an ein anderes Objekt, welches von ersterem strukturell abhängig ist, weiterzugeben. Es ist auch unter dem Namen **publish-subscribe** bekannt.

1.1 Problem

Als Beispiel einer Anwendung ohne Observer Pattern betrachten wir folgende Problemstellung:

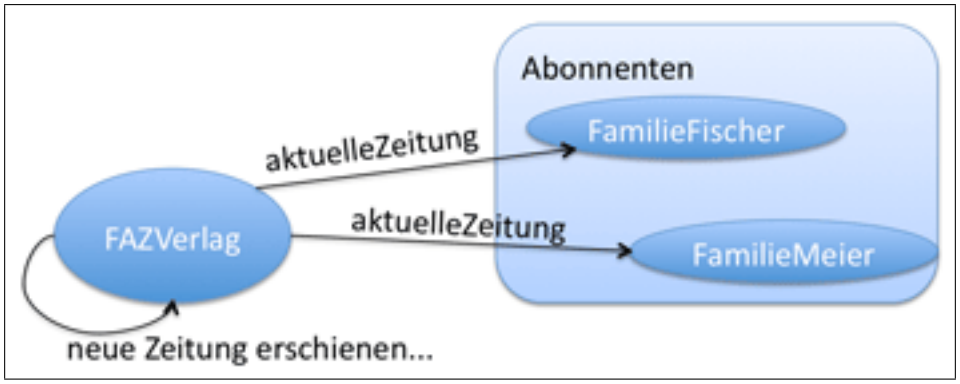

Abbildung 1.1: FAZVerlag Problem

Der FAZ-Verlag versendet die jeweils aktuelle Zeitung an seine Abonnenten, Familie Fischer und Meier.
Bei jeder neuen Zeitung soll eine Auslieferung erfolgen.

Nachfolgend ist der Sachverhalt als Klassendiagramm einer simplen Lösung (ohne Observer Pattern) dargestellt:

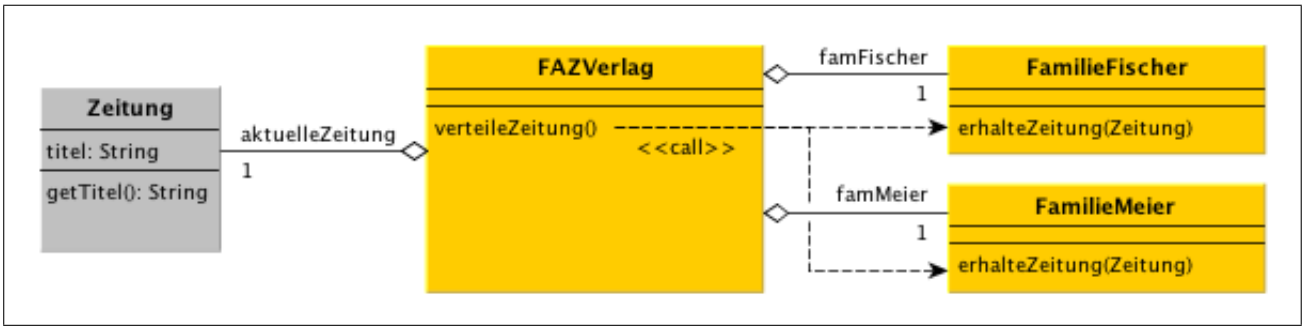

Abbildung 1.2: Klassendiagram FAZVerlag Problem

Die Familien Fischer und Meier, sowie die Zeitung sind Elemente der Klasse FAZ-Verlag.
Der FAZVerlag ruft einzeln die Funktion erhalteZeitung(Zeitung) auf.

1.1.1 Eine simple Java Implementierung

Eine schnelle, sehr unsaubere Lösung ist hier nachfolgend in Java implementiert:

```
1  public class FAZVerlag {
2
3    private Zeitung aktuelleZeitung;
4    private FamilieFischer famFischer;
5    private FamilieMeier famMeier;
6
7    //Sobald eine neue Ausgabe existiert
8    public void verteileZeitung() {
9            famFischer.erhalteZeitung(aktuelleZeitung);
10           famMeier.erhalteZeitung(aktuelleZeitung);
11   }
12 }
```

Listing 1.1: Beispiel Java

Sobald eine neue Ausgabe der Zeitung erscheint, wird für jede Familie einzeln deren implementierte Funktion “erhalteZeitung” aufgerufen. Die Familien sind fest implementiert und können nicht zur Laufzeit verändert werden.

1.1.2 Nachteile

- Enge Verbindung zwischen FAZVerlag und Abonnenten
- Die Erweiterbarkeit ist stark eingeschränkt
- Abonnement bestellen oder abbestellen während der Laufzeit nicht möglich.

1.2 Lösung: Das Observer Pattern

Für eine gute, erweiterbare Lösung: Das **Observer Pattern**.

"Definiere eine 1-zu-n-Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, das alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden."([GoF], Seite 287)

Der grobe Sachverhalt des Observer Patterns sieht wie folgt aus:

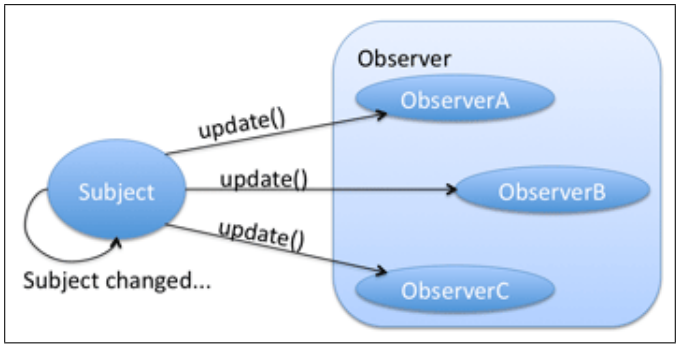

Abbildung 1.3: Observer Pattern Schema

Das Observer Pattern besteht aus Objekten (Observer), die sich bei einem anderen Objekt (Subjekt) registrieren.
Sobald sich etwas am Subjekt ändert, werden die Observer informiert (update).

Hier wird der Aufbau in Form eines Klassendiagramms noch einmal genauer betrachtet:

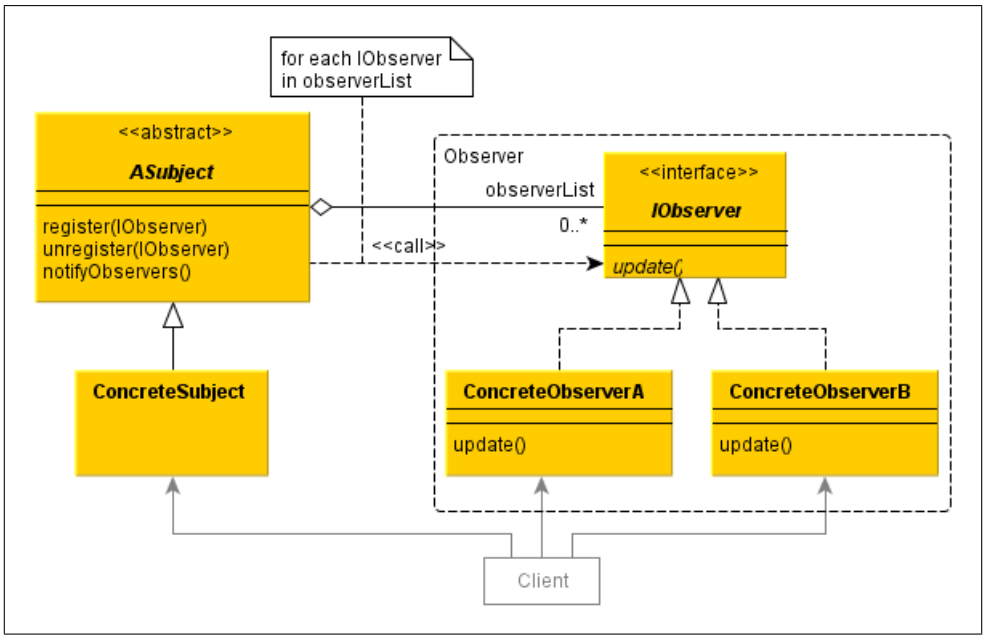

Abbildung 1.4: Observer Pattern Klassendiagramm

Um die Observer an und abzumelden braucht das Subjekt Administrationsmethoden (registrieren, abmelden) und eine Liste der Observer.
Es wird eine Schnittstelle mit mindestens einer Aktualisierungsmethode benötigt, um die Observer einheitlich zu informieren.
Von dieser Schnittstelle können sich dann beliebig viele Objekte ableiten.Jedes dieser Objekte wird durch die selbe Funktion über Änderungen informiert.

1.2.1 Variationen

Auf Grund des hohen Beliebtheitsgrades des Observer Patterns, haben sich mehrere Variationen gebildet. Hier soll nur auf die Varianten der Aktualisierungsmethoden eingegangen werden.

Es gibt zwei Varianten von Aktualisierungsmethoden. Das **Push** und das **Pull**-Modell.

Push-Modell

Das Subjekt übergibt die Informationen zum Update per Parameter.

Listing 1.2: Beispiel Push-Modell

```
1  public interface IObserver1{
2
3      public void update(int pLength, int pWidth, boolean pVisible,
           String pName);
4
5  }
```

✓ Observer benötigt keine Informationen über Subjekt -> Starke Entkopplung!

✗ Nicht jeder Observer benötigt alle/die selben Parameter!

✗ Bei Erweiterung müssen alle Observer angepasst werden
-> Event-Objekt als Parameter anstatt einzelner Parameter
(GUI-Objekte wie Swing nutzen diese Methode)

Pull-Modell

Das Subjekt verschickt nur eine kleine Benachrichtigung an die Observer. Die Observer müssen daraufhin sich selbst die benötigten Informationen vom Subjekt holen.

Listing 1.3: Beispiel Pull-Modell

```
1  public interface IObserver2 {
2
3      public void update(ConcreteSubject pConcreteSubject);
4
5  }
```

✓ Jeder Observer holt sich per Getter nur die benötigten Informationen.

✓ Bei mehreren Subjekten: Eindeutig von welchem Subjekt!

✗ Kann ineffizient werden, da Observer herausfinden muss was sich konkret verändert hat.

Merke:

- Weiß das Subjekt von den Anforderungen der Observer -> Push-Modell
- Weiß das Subjekt nichts über die Observer -> Pull-Modell

1.3 Beispiel

Zurück zu unserem FAZ-Beispiel.
Durch Anwenden des Observer Patterns entsteht folgendes Klassendiagramm:

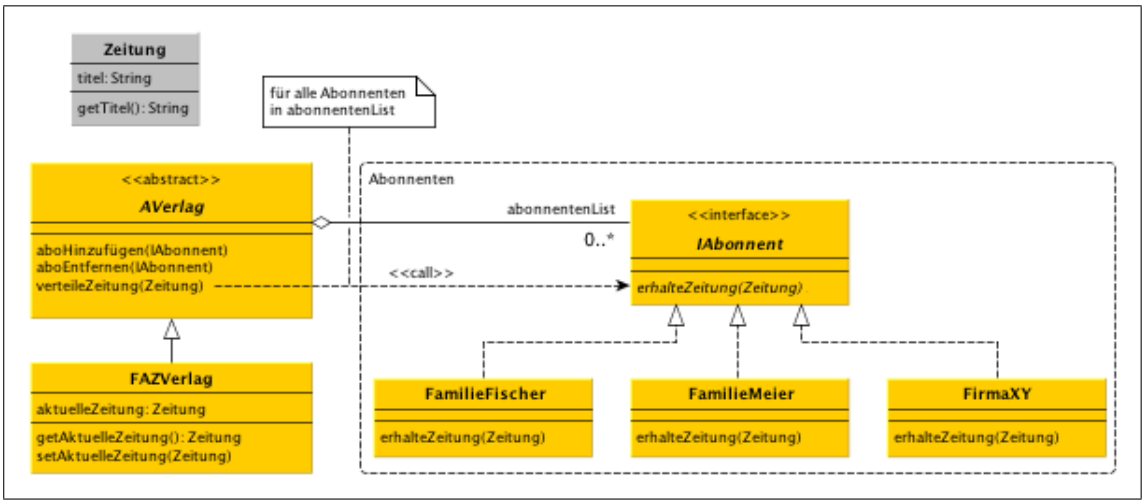

Abbildung 1.5: Observer Pattern FAZVerlag

Der konkrete FAZ-Verlag ist von einer Abstrakten Klassen AVerlag abgeleitet. Jeder Verlag muss einen Abonnenten seiner Liste hinzufügen/entfernen und eine Zeitung verteilen können. Alle Abonnenten der Zeitung implementieren das Interface IAbonnent um eine einheitliche Schnittstelle für Aktualisierungen zu bieten.

1.4 Vor und Nachteile

Vorteile

- **Zustandskonsistenz**: Änderungen werden automatisch an alle Observer weitergegeben.
- **Modularität**: Es ist möglich dass mehrere Observer ein Subjekt beobachten, aber auch ein Observer mehere Subjekte.
  Die Anzahl der Observer muss zuvor nicht bekannt sein.
- **Wiederverwendbarkeit**: Subjekt und Observer sind unabhängig variierbar. Beide sind also einzeln wiederverwendbar.

Nachteile

- **Aktualisierungszyklen**: Bei großen Systemen kann es zu Änderungsketten kommen, die im schlimmsten Fall rekursive Aufrufe nach sich ziehen (Zyklen). Ebenso können bei komplexen Systemen einige unnötige Aktualisierungen auftreten.
- **Abmeldung vom Observer**: Es kann schnell passieren, dass man vergisst Observer abzumelden. Speicher wird eventuell nicht freigegeben.

2 MVC

2.1 Problem

Während der Entwicklung einer Software kann man immer wieder auf verschiedene Probleme treffen. Manche dieser Probleme betreffen die Anzeige von Daten. Hierbei sind zwei Probleme hier einmal herausgestellt:

- Mehrere verschiedene Ansichten bei gleichen Daten.
- Änderung der Ansicht (z.B. von 2D auf 3D, Punktdiagramm, Liniendiagramm, Kreisdiagramm) bei gleichbleibenden Daten.

Bei solchen oder ähnlichen Problemstellungen kann das Design Pattern MVC (Model View Controller) helfen.

2.2 Definition

MVC behandelt drei Rollen:

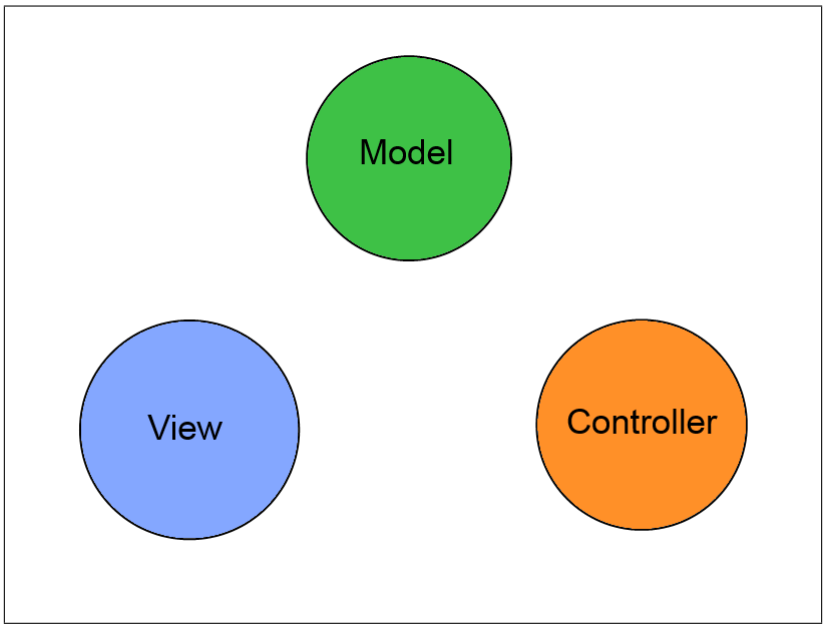

Abbildung 2.1: Model View Controller

2.2.1 Model

Implementiert die zentrale Struktur der Anwendung. (Datenhaltung)

2.2.2 View

Repräsentiert die Anzeige des Models in dem User Interface

2.2.3 Controller

Verwaltet Benutzereingaben, manipuliert das Model und aktualisiert die View Es gibt keine Standarddefinition für MVC. Viele Frameworks benutzen unterschiedliche Versionen.

Connelly Barnes: "An easy way to understand MVC: the model is the data, the view is the window on the screen, and the controller is the glue between the two."

2.3 Schlüsselaspekte von MVC

- Die View/Ansicht von dem Modell trennen.
- Ermöglicht es mehrere verschiedene User Interfaces zu implementieren und die Module besser zu testen.
- Den Controller von der Präsentation trennen.
- Besonders nützlich mit Web Interfaces. (bei GUI frameworkds eher weniger)

Martin Fowler: "The irony ist hat almost every version of smalltalk didn't actually make a view/controller separation"

2.4 Unterschiedlche MVC Definitionen

MVC wurde von **Trygve Reenskaug** im Jahre 1979 das erste Mal beschrieben.

Eine der ersten Diskussion "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80" im JournalOfObjectOrientedProgramming (JOOP), von **Glenn Krasner** und **Stephen Pope**, erschien im August/September 1988.

Danach wurden durch ständige Weiterentwicklung und Anpassung viele verschiedene Definitionen von MVC entwickelt.

Die bekanntesten und wichtigsten werden hier aufgeführt:

2.4.1 MVC in Smalltalk '80

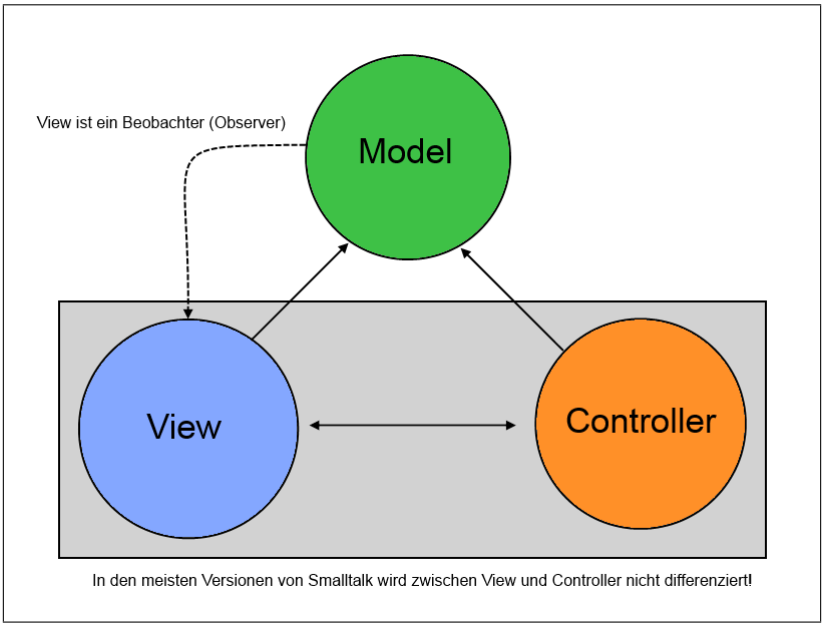

Abbildung 2.2: Smalltalk Model View Controller

Schlüsselaspekte

- Das Model ist weder von der View noch dem Controller abhängig.
- Bei Änderungen am Model werden durch das **Observer Pattern** die Views aktualisiert.
- Smalltalk '80 hat View und Controller nicht getrennt.

Nachfolgend ein kleines Beispiel mit zwei Klassen in Smalltalk:

Listing 2.1: Beispiel Smalltalk

```
class ModelCounter
    constructor: ->
        @observers = []
        @value = 1

    increaseValue: (delta) =>
        @value += delta
        @notifyObservers()

    notifyObservers: =>
        obj.notify(this) for obj in @observers

    registerObserver: (observer) =>
```

```
        @observers.push(observer)

class ViewCounterButton
    constructor: (opts) ->
        @model_counter = opts.model_counter
        @button_class = opts.button_class or 'button_counter'
        @model_counter.registerObserver(this)

    render: =>
        elm = $("<button class=\"#{@button_class}\">
                #{@model_counter.value}</button>")
        elm.click =>
            @model_counter.increaseValue(1)
        return elm

    notify: =>
        $("button.#{@button_class}").replaceWith(=> @render())
```

3 MVP

3.1 MVP: Model-View-Presenter

3.1.1 Erklärung

MVP stellt eine Weiterentwicklung von MVC dar. Da die Komponenten in MVC nicht in jedem aktuellen Anwendungsgebiet eindeutig definiert werden können und der Übergang der Komponenten teilweise fließend erfolgt, ist eine Erneuerung/Verbesserung der Grundidee dieses Patterns notwendig gewesen.

Model-View-Presenter wurde in den 90er-Jahren von Taligant und IBM entwickelt. Martin Fowler formulierte jedoch im Jahre 2004 Model-View-Presenter nach seinem Verständnis. Seine Definition ist heute ausschlaggebend.

Das Ziel war die Generalisierung des in Smalltalk implementierten MVC-Patterns. Es sollte die Anforderungen für aktuellere Programmiersprachen, wie beispielsweise Java oder C++ erfüllen.

Allgemein gehalten gibt es für MVP leider keine exakte Definition. Die konkrete Implementierung von MVP hängt vom Anwendungsfall und der verwendeten Programmiersprache ab.

Martin Fowler beschreibt auf seiner Internet-Präsenz (www.martinfowler.com) folgende zwei Möglichkeiten der Implementierung:

Supervising Controller

> "Factor the UI into a view and controller where the view handles simple mapping to the underlying model and the the controller handles input response and complex view logic."

Hierbei übernimmt die View die Datensynchronisation zu großen Teilen selbst. Somit wird der aktive Synchronisationsaufwand im Presenter möglichst gering gehalten.

Passive View

> "Passive View is a very similar pattern to Supervising Controller, but with the difference that Passive View puts all the view update behavior in the controller, including simple cases."

Die passive View wird so einfach wie möglich gehalten. Dies hat unter anderem den Vorteil, dass diese von Entwicklern/Designern erstellt werden können, die wenig bis keine Informationen über die Programmlogik haben. Hieraus folgt, dass der Presenter die Datensynchronisation weitestgehend vollständig übernimmt.

Um den Umfang der Ausarbeitung überschaubar zu halten, wird im folgenden von einer passiven View ausgegangen.

Die allgemeine Rollenverteilung in MVP ist wie folgt:

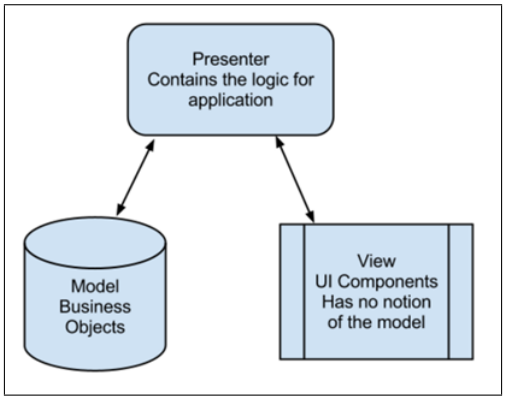

Abbildung 3.1: grundsätzliche MVP-Logik

Im Grundsatz ändert sich nur ein zentraler Aspekt im Vergleich zum klassischen MVC:

Der Presenter ist das einzige Bindeglied zwischen Model und View. Im Vergleich wird bei diesem Pattern in der Theorie strikt darauf geachtet, dass die View keine Logik bezüglich der Daten mehr enthält und hat lediglich die Funktion, die Daten darzustellen. Die zentrale Steuerung der Ansicht erfolgt durch den Presenter, woraus resultiert, dass die View keinen Zugriff auf die Funktionen des Models hat.

Im folgenden werden die drei Kernkomponenten für MVP definiert:

Das Model

Die zentrale Aufgabe des Models ist die Bereitstellung von Daten. Daraus folgen weitere Aufgaben wie Datenhaltung, Datenstrukturen, persistente Speicherung der Daten sowie die Bereitstellung einer eindeutig definierten Schnittstelle. Die Daten bzw. interne Geschäftslogik wird vollständig durch das Model gekapselt. Hierdurch können unter anderem Datenstrukturen innerhalb des Models ohne Veränderung der restlichen Programmlogik ausgetauscht werden. Zusätzlich ist es nicht notwendig, dass das Model der Datenspeicher an sich ist. Es ist auch möglich, dass es

lediglich die Funktion eines Proxy hat und die Daten von z. B. einer Datenbank "weiterreicht".

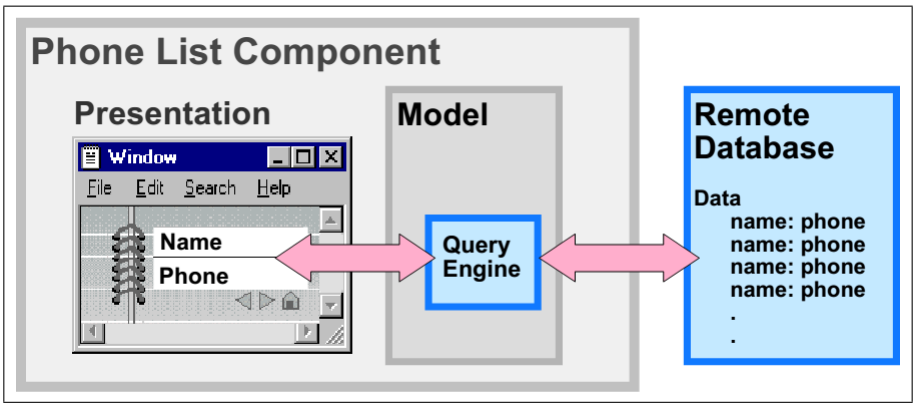

Abbildung 3.2: Beispiel für ein Model als Proxy

Für die Erstellung eins Models ergeben sich drei zentrale Fragen:

- Was sind meine Daten?
- Wie spezifiziere ich meine Daten?
- Wie ändere ich meine Daten?

Um diese Fragen klären zu können ist eine eindeutige Geschäftslogik notwendig. Zusätzlich müssen diese Fragen vor Implementierung bei der Festlegung des Software-Designs geklärt werden.

Der Presenter

Der Presenter stellt wohl die größte Änderung zum klassischen MVC dar.

Seine Aufgaben kann man wie folgt beschreiben:

- Interpretation der Ereignisse und Eingaben die durch den Benutzer ausgelöst wurden
- Manipulation des Models durch Benutzerinteraktion
- Aktualisierung der View-Komponenten durch Änderungen am Model (Hierbei wird meist das Observer-Pattern verwendet)

Man kann sich View und Model auch als Schichten vorstellen, wobei der Presenter das Bindeglied dieser Schichten ist. Dieser arbeitet konkret mit den Schnittstellen von Model und View zusammen.

Die View

Die View hat die Aufgabe die Eingabe- und Ausgabeelemente darzustellen.

Die Aufgaben der View kann man im wesentlichen wie folgt beschreiben:

- Darstellung der Model-Daten
- Entgegennahme der Benutzereingaben

Aus Sicht des Programmierers ist für die View wichtig, wie interagiert der Benutzer mit den Daten. Zum Beispiel welche Eingaben sind an der Benutzerschnittstelle erlaubt?

3.1.2 Mögliche Strukturierung einer MVP-Anwendung

Im folgenden soll eine mögliche Implementierung kurz beschrieben werden. Im darauf folgenden Abschnitt wird es für ein besseres Verständnis noch ein Beispiel geben.

Grundsätzlich müssen zu Beginn eindeutige Schnittstellen für die Views und Models unserer Anwendung definiert werden. Beim Model ist hierbei besonders darauf zu achten, welche Operationen innerhalb der Datenhaltung (von außerhalb) zulässig sind. Es ist auch möglich mehrere Interfaces für ein Model zu spezifizieren um z. B. mehrere Benutzergruppen wie Administratoren oder normale Benutze zu Verwalten. Daraus erfolgen im Normalfall verschiedene Presenter.

Bei der View müssen für das Event-Handling der einzelnen Benutzerinteraktionen jeweils Methoden bereitgestellt werden, die dem Presenter erlauben diese zu steuern. Zusätzlich muss der Presenter Zugriff auf die Ausgabe-Elemente der GUI haben, um mögliche Änderungen durch das Model oder Benutzer-Eingabe setzen zu können (Hierbei wird von einer rein passiven View ausgegangen die selbst keinen Zugriff auf die eigentliche Steuer- bzw. Programmlogik enthält).

Der Presenter verwendet die Schnittstellen, die durch View und Model bereitgestellt werden. Hierdurch entsteht ein zentrales Bindeglied.

Alle Models und Views gleicher Art können nun die definierten Schnittstellen implementieren und die entsprechende benötigte Programmlogik bereitstellen.

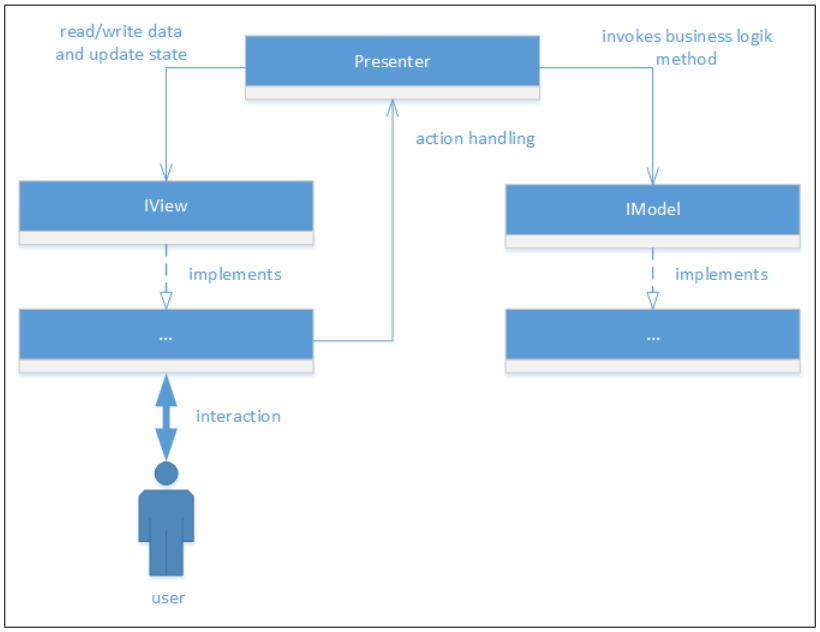

Abbildung 3.3: Beispiel für MVP-Architektur

Abbildung 3.3 zeigt, wie eine MVP-Anwendung strukturiert sein kann, wobei jede View und Model von einem einheitlichen Interface abgeleitet werden und der Presenter sowohl View- als auch Model-Interface kennt.

Hinweis: Das gezeigte Klassendiagramm ist stark vereinfacht. Ein Presenter kann zum Beispiel aus mehreren hundert Klassen bestehen. Zusätzlich wurde eine Sever/Client-Architektur in Verbindung mit MVP nicht behandelt, da dies den Umfang der Ausarbeitung übertreffen würde.

3.1.3 Konkrete Beispielanwendung in Java

Im folgenden soll ein kleines MVP-Beispiel für mehr Verständnis sorgen. Die Implementierung wird in Java 1.7 erfolgen.

Folgendenes einfache Beispiel:

Es soll ein virtueller Geldbeutel implementiert werden. In unserem Geldbeutel befinden sich zwischen 0 und 100 Euro. Der Geldbestand soll über verschiedene Benutzeroberflächen verändert werden können, die alle auf den selben Geldbeutel zugreifen. Unsere Anwendung soll zusätzlich um weitere Geldbeutel bzw. Benutzeroberflächen erweiterbar sein.

Mögliche Lösung:

Im nachfolgendem wird eine mögliche Implementierung mit zwei Schnittstellen mit Hilfe des Model-View-Presenter Patterns zum Verändern des Geldbeutels beschrieben.

Zuerst wird die Struktur der Anwendung anhand eines Klassendiagramms kurz erklärt. Danach wird auf die einzelnen Interfaces eingegangen. Abschließend werden noch Beispieloberflächen, sowie Programmaufruf präsentiert.

Hier ein mögliches Klassendiagramm:

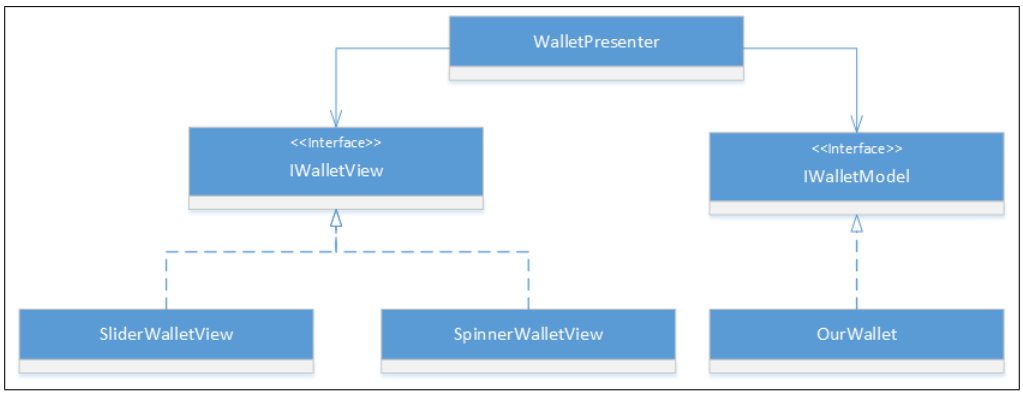

Abbildung 3.4: Klassendiagramm: Brieftasche

Unser Brieftaschen-Presenter hängt jeweils von den Brieftaschen-Interfaces der View und des Presenter ab. Zusätzlich benutzt er das Observer-Pattern um über Änderungen im Model informiert zu werden.

Hinweis: Da bei der passiven View die Handler für die einzelnen GUI-Elemente im Presenter liegen, wird hierbei implizit ebenfalls das Observer-Pattern eingesetzt, um Veränderungen an der GUI durch den Benutzer mitzuteilen.

Weiterer Hinweis: Auf die explizite Darstellung der Abhängigkeiten von Java-Event-Handling, sowie der Observer-Implementierung wird aus Gründen der Übersicht verzichtet.

Nun zur Spezifikation der Interfaces

Zuerst werden die Interfaces für unsere View und unser Model spezifiziert. Die entscheidenden Schnittstellen werden kurz beschrieben.

Listing 3.1: Beispiel für ein View-Interface

```
...
public interface IWalletView {

    public void setMoneyListener(ChangeListener e_ref);
    public int  getMoneyFromView();
    public void setMoneyToView(double inputMoney);
    ...
}
```

- setMoneyListener(...): Für unseren Geldbeutel-Presenter muss es später möglich sein, auf den Handler der Views zu zugreifen, da unsere View passiv ist und der Presenter Änderungen an das Model (unseren Geldbeutel) weiterleiten muss.
- getMoneyFromView(): Hierüber wird dem Model mitgeteilt, was der neue aktuelle Geldbestand ist.
- setMoneyToView(...): Bei Änderungen an unserer Brieftasche müssen die einzelnen Benutzer-Schnittstellen auf den neuen Stand synchronisiert werden.

Nachdem zentralen Schnittstellen der View festgelegt wurden, geschieht nun das Gleiche mit unserem Model:

Listing 3.2: Beispiel für ein Model-Interface

```
...
public abstract class AWallet extends Observable {

    public abstract double getWalletValue();
    public abstract void   setWalletValue(double inputMoney);
    ...
}
```

Unser Model-Interface besteht im Grunde nur aus jeweils einer Getter- und einer Setter-Methode. Zum besseren Verständnis werden diese dennoch kurz beschrieben:

- getWalletValue(): Hiermit wird der aktuelle Bestand an Geld zurückgegeben.
- setWalletValue(...): Durch diese Methode wird der aktuelle Geldbetrag in unserem Geldbeutel festgelegt.

Hinweis: Im Code-Beispiel wird eine abstrakte Klasse verwendet, obwohl im Rest der Beschreibung immer von einem Model-Interface gesprochen wird. Der Grund hierfür ist, dass jedes unserer Geldbeutel-Models von der Klasse Observable erben muss. Es wäre auch ein Interface möglich oder eine Kombination aus einem Interface und einer abstrakten Klasse in Java.

Alternativ kann hier auch auf das Java-Observer Pattern verzichtet werden und eine eigene Implementierung erfolgen.

Nun da unsere Interfaces für Model und View spezifiziert sind, kann der Presenter definiert werden.

Listing 3.3: Beispiel für einen Presenter

```
...
public class WalletPresenter {

    private IWalletView walletView_ref;
    private AWallet     walletModel_ref;

    public WalletPresenter(IWalletView inputWalletView_ref,
                           AWallet inputWalletModel_ref)
    {...}
}
```

Zuerst zum Konstruktor: Da der Presenter nur mit den Schnittstellen der einzelnen Model- und View-Objekten arbeitet, muss er diese auch nicht konkret kennen. Zusätzlich können diese im Konstruktor übergeben werden, wodurch es einfacher ist Test-Szenarien des Presenter aufzubauen, da lediglich "Mock-Objekte" (Platzhalter für Modultests) benötigt werden.

Im Konstruktor muss zusätzlich unser Presenter noch als Beobachter bei unserem Model registriert werden. Weiterhin wird festgelegt, dass bei einer Änderung am Model die View ebenfalls durch den Presenter aktualisiert wird.

Listing 3.4: Beispiel für einen Presenter

```
...
 public void update(...) {

   double value = WalletPresenter.this.walletModel_ref.getWalletValue();
   WalletPresenter.this.walletView_ref.setMoneyToView(value);

}
...
```

Anschließend muss im Konstruktor noch der Handler für unsere Benutzerschnittstellen-Events definiert werden.

Listing 3.5: Beispiel für einen Presenter

```
  ...
  this.walletView_ref.setMoneyListener(new ChangeListener() {

    public void stateChanged(ChangeEvent e) {

      int value = WalletPresenter.this.walletView_ref.getMoneyFromView();
```

```
            WalletPresenter.this.walletModel_ref.setWalletValue(value);
8
        }
10  });
    ...
```

Der Presenter ist relativ einfach gehalten und benötigt lediglich einen Konstruktor. Bei größeren Presenter werden natürlich noch weitere Hilfsmethoden benötigt. Im Idealfall jedoch hat der Presenter keine öffentliche Schnittstellen außer den Konstruktoren! Hiermit soll die Modularität der Anwendung erhalten bleiben.

Das folgende Sequenzdiagramm zeigt, was innerhalb der einzelnen Komponenten passiert, sollte der Benutzer den Inhalt des Geldbeutels durch eine der Benutzerschnittstellen ändern.

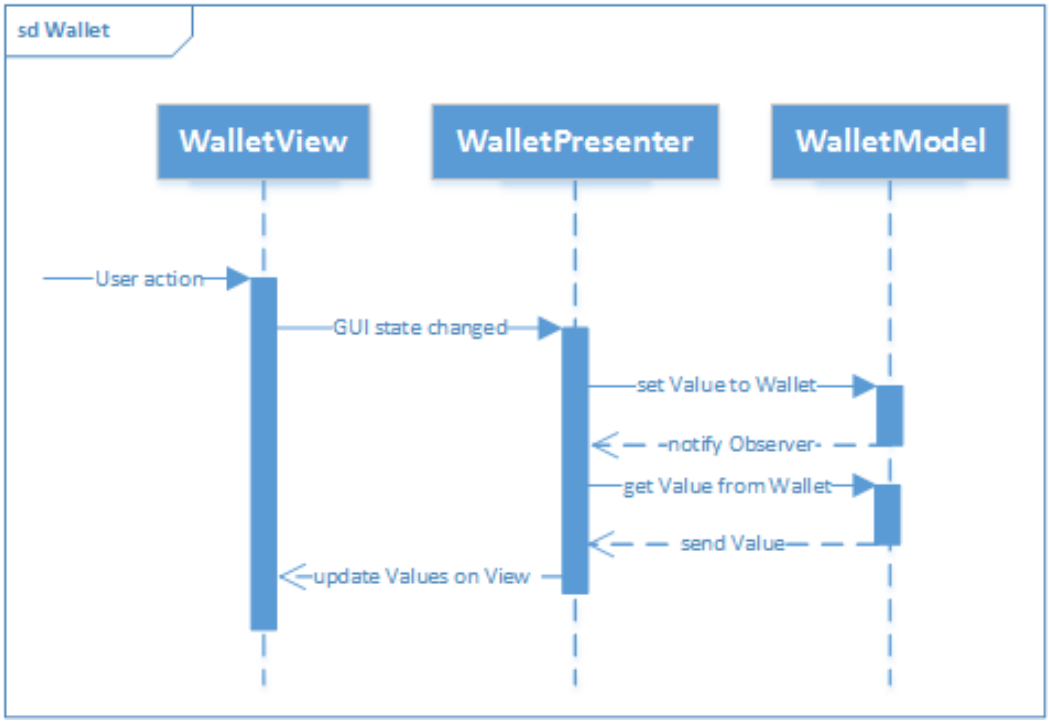

Abbildung 3.5: Ablauf bei Änderung des Geldbetrages durch den Benutzer

Durch die Veränderung des Geldbetrages durch den Benutzer wird durch das Change-Event der Benutzeroberfläche der Presenter informiert und entsprechend wird der Betrag im Model geändert. Da der Presenter ein Beobachter des Models ist, informiert dieses den Presenter über die Änderungen. Nun holt sich der Presenter den aktuellen Betrag vom Model und führt hiermit ein Update der View aus.

Hinweis: Das Konstrukt kann am Anfang etwas umständlich wirken. Der Presenter hätte auch direkt den Eingabewert benutzen können, um hiermit die View

zu aktualisieren. Die Prüfung auf Korrektheit übernimmt jedoch das Model, dies ist Teil der Datenkapselung. Wenn man das Beispiel zusätzlich noch verallgemeinert, kann das Model noch von anderen Presentern in der Zwischenzeit verändert werden, wodurch der verwendete Wert nicht mehr aktuell sein kann.

Weitere Überlegung: Es wäre auch eine Multi-Threading-Umgebung hier möglich gewesen. Dies hätte eine asynchrone Abarbeitung erlaubt. Besonders bei größeren Anwendungen ist dies von Vorteil, weil hierdurch Verzögerungen innerhalb der einzelnen Komponenten vermieden werden und keine Komponente "direkt" auf die andere warten muss. Doch müssen hierbei Abhängigkeiten zwischen den Modulen und Daten berücksichtigt werden.

Zur besseren Veranschaulichung werden noch mögliche einfache Benutzeroberflächen gezeigt.

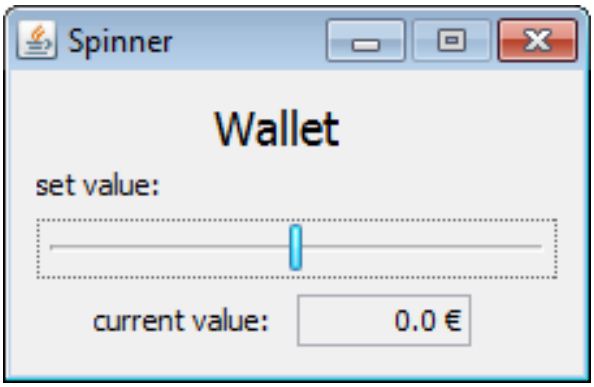

Abbildung 3.6: Beispiel: Geldbeutel-View mit JSlider und Textanzeige

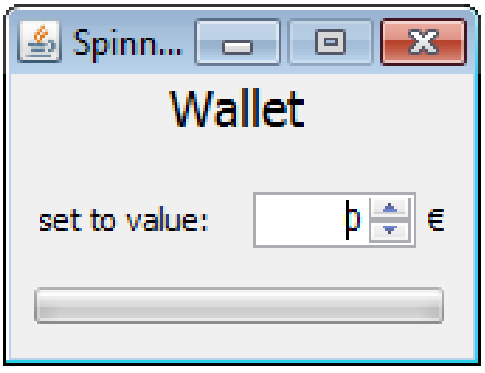

Abbildung 3.7: Beispiel: Geldbeutel-View mit JSpinner und JProgressBar

Durch den JSlider und den JSpinner der Benutzerschnittstellen kann der aktuelle Betrag verändert werden. Wobei die Anzeige des Geldbetrags jeweils bei allen Schnittstellen synchron gehalten wird.

Abschließend kann die Anwendung mit folgenden Befehlen gestartet werden.

Listing 3.6: Beispiel für Aufruf der Anwendung

```
1
        public static void main(String[] args) {
3               AWallet wallet_ref =  new OurWallet();
                new WalletPresenter(new SliderWalletView() ,wallet_ref);
5               new WalletPresenter(new SpinnerWalletView(),wallet_ref);
                ...
7           }
          ...
```

Für unserer beiden Benutzeroberflächen wird jeweils ein ein Präsenter-Objekt erstellt. Das Model bleibt in beiden Fällen das Selbe.

3.1.4 Vor- und Nachteile

Vorteile

- Konzept mit strukturierter Rollenverteilung, dadurch entsteht eine relativ klare Aufgabenteilung.
- Modultests der Anwendung sind durch Austauschbarkeit der Komponenten relativ einfach möglich.
- Die Aufteilung erlaubt es Entwicklern sich einzubringen ohne die komplette Anwendungslogik verstehen zu müssen.
- Das Pattern erlaubt es die einzelnen Elemente wie Model und View zu erweitern.

Nachteile

- Höherer Implementierungs- sowie Designaufwand.
- Die Verwendung des Patterns erfordert klare Definition der einzelnen Rollen, was nicht auf jede Problemstellung einfach übertragen werden kann.

3.1.5 Fazit

Das Model-View-Presenter Pattern bietet eine strukturierte Vorgehensweise zum erstellen größerer Benutzeroberflächen mit klar abgrenzender Geschäftslogik. Die

Rollen der einzelnen Komponenten innerhalb der Anwendung sind hierbei klar definiert und die einzelne Module können weiterhin ausgetauscht oder ergänzt werden.

Doch besonders für kleinere Anwendungen ist im Verhältnis ein relativ hoher Design-Aufwand nötig. Die Interfaces von View und Model müssen zu Beginn schon möglichst exakt spezifiziert werden können, da nachträgliche Änderungen schnell zu Chaos im Konzept der Anwendung führen. Zusätzlich gilt auch hier, wie für jedes Pattern, dass nicht jede Anwendung mit diesem Konzept realisiert werden kann.

4 MVVM

Während die Ursprünge des MVC-Entwurfsmusters in den 1980er Jahren liegen und das MVP-Entwurfsmuster erst in den 1990 Jahren zum ersten Mal beschrieben wurde, ist das MVVM noch relativ jungen Alters. Das MVVM ist zudem anders als die beiden anderen Entwurfsmuster das spezifischste Muster, denn während MVC und MVP relativ geringe Voraussetzungen für ihre Verwendung haben, ist MVVM speziell für moderne Technologien konzipiert. So setzt die Verwendung des MVVM den Einsatz von Microsofts WPF oder Silverlight zwar nicht unbedingt voraus, ist aber ratsam. Trotz dieser Einschränkung, ist es kein Fehler sich mit diesen Muster zu beschäftigen, schließlich werden mittlerweile ein Großteil aller Benutzeroberflächen, die auf .NET-Framework basieren damit erstellt.

4.1 Erklärung

So wie das MVP eine Weiterentwicklung des MVC ist, ist das MVVM eine Weiterentwicklung des MVP-Entwurfsmusters. Das Ziel dabei ist eine vollständige Entkopplung der View von der Logik.

4.1.1 Model

Das Model hat die selbe Aufgaben wie im MVC und MVP: Datenzugriffsschicht für die Inhalte, dem Benutzer angezeigt und von ihm manipuliert werden. Dazu benachrichtigt es über Datenänderungen und führt eine Validierung der vom Benutzer übergebenen Daten durch. Hierdurch wird vor allem in der View der Code-Behind minimiert.

4.1.2 ViewModel

Das ViewModel ist laut Martin Fowler praktisch dem Presenter im MVP gleichzusetzen. Es beinhaltet die UI-Logik (Model der View) und dient als Bindeglied zwischen View und obigen Model. Einerseits tauscht es Information mit dem Model aus, ruft also Methoden oder Dienste auf. Andererseits stellt es der View öffentliche Eigenschaften und Befehle zur Verfügung. Diese werden von der View an Steuerelemente gebunden, um Inhalte auszugeben bzw. UI-Ereignisse weiterzuleiten. Das ViewModel darf dabei keinerlei Kenntnis der View besitzen.

4.1.3 View

Die View ist wie im MVC und MVP für die Bereitstellung der Benutzeroberfläche zuständig. Außerdem gilt im MVVM-Entwurfsmuster: Die View bindet sich an Eigenschaften des ViewModel, um Inhalte darzustellen und zu manipulieren sowie Benutzereingaben weiterzuleiten. Dieses sogenannte Binding ermöglicht die Austauschbarkeit der View. Der Codebehind kann ebenfalls gering gehalten werden.

Die View muss einzig per Binding an bestimmte Properties des ViewModels gebunden sein. In Abbildung 4.1 sind die Schichten des MVVM-Entwurfsmusters schematisch dargestellt.

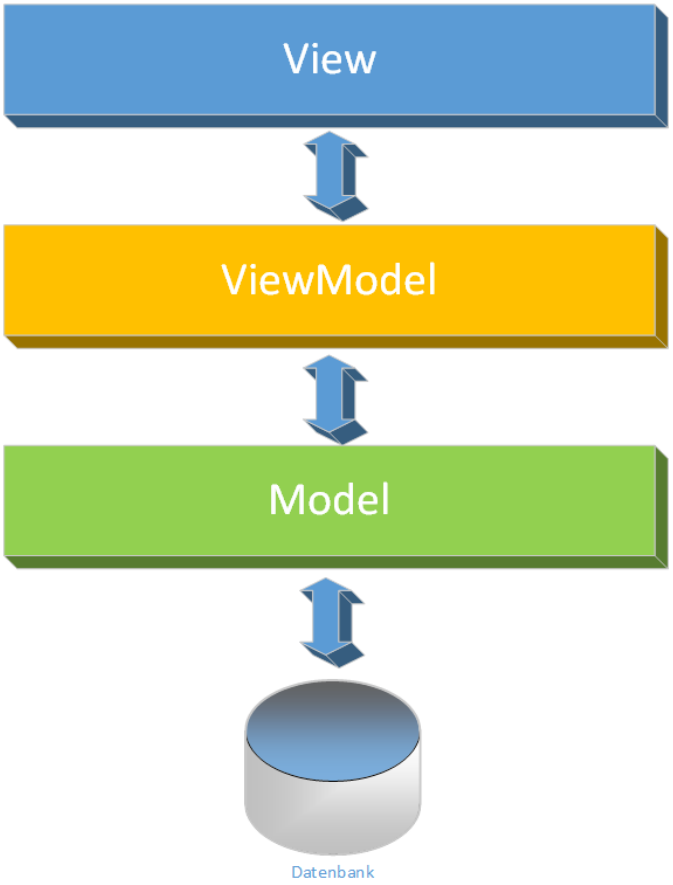

Abbildung 4.1: Die Schichten des MVVM-Entwurfsmusters

4.2 Problem

Bei der Erstellung einfacher Anwendungen reicht es oftmals eine Benutzeroberfläche zu erstellen, die für alle Auslieferungen gleich ist und für immer gleich bleiben wird. Ein Beispiel hierfür ist der bekannte Taschenrechner, der bei wahrscheinlich jeder Windowsanwendung mitgeliefert wird, siehe Abbildung 4.2.

Jedoch ist die Welt bekanntlich nicht immer so einfach wie bei obigen Beispiel. Benutzeroberflächen sind oftmals die ersten Komponenten einer Anwendung, die dem Missfallen des Endkunden unterliegen. Grund hierfür ist das allgemeine Desinteresse und meist sogar Unvermögen von Softwareentwicklern optisch reizvolle

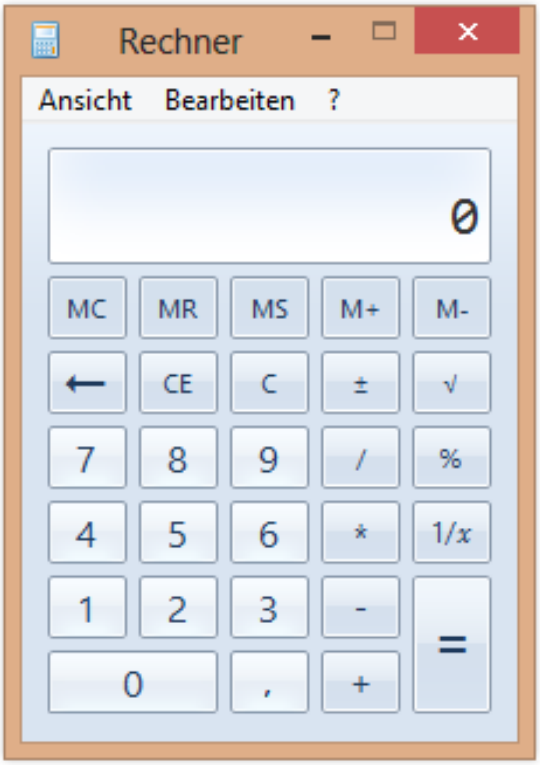

Abbildung 4.2: Windows-Taschenrechner als Beispiel einer gleichbleibenden Benutzeroberfläche

Benutzeroberflächen zu entwickeln. Dies ist sicherlich verständlich, schließlich hat ein Entwickler genug andere Baustellen um seine Anwendung zum Laufen zu bringen. Es macht daher Sinn, die Designarbeit an den für Design begabten Designer auszulagern. Jedoch versteht dieser in der Regel nicht, was dem Entwickler besonders liegt. Eine Möglichkeit dem Standarddesigner ohne Programmierkenntnisse die Entwicklung der Benutzeroberfläche zu ermöglichen, ohne den Entwickler damit zu "belästigen" ist daher wünschenswert. Hierfür ist jedoch eine Entkopplung der UI-Logik, die dem Entwickler obliegen muss von dem Design der Benutzeroberfläche nötig.

4.3 Lösung mit MVVM und WPF

Eine Lösung des Problems kann durch den Einsatz des MVVM-Entwurfsmusters in Verbindung mit WPF von Microsoft erreicht werden. Bei der WPF handelt es sich um ein auf dem .NET-Framework von Microsoft basierenden GUI-Framework, als moderne alternative zu Winforms. Mehr zu WPF am Ende des Abschnittes. In Abbildung 4.3 sind die Schichten des MVVM in Verbindung mit der WPF dargestellt.

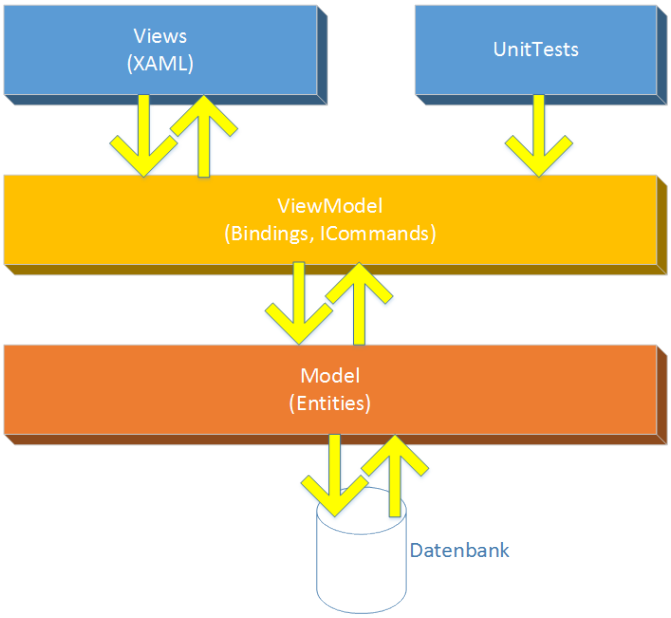

Abbildung 4.3: Die Schichten des MVVM-Entwurfsmusters in Verbindung mit der WPF

4.3.1 Databinding

Um MVVM und die WPF auszureizen wird sog. Databinding verwendet. Hierbei wird die gewünschte Eigenschaft der View an die entsprechende Eigenschaft des ViewModels "'gebunden"', wobei Änderungen auf der einen Seite auch auf der anderen Seite sichtbar werden (Abb. 4.4. Dafür wird im ViewModel das Interface INotifyPropertyChanged implementiert welches ein PropertyChangedEvent werfen kann sobald sich der Wert einer Eigenschaft ändert. Dadurch bekommt die View mit wenn sich am ViewModel etwas ändert. Das DataBinding kann dabei die Werte OneTime, OneWay, TwoWay annehmen, wobei meist letzteres zum Einsatz kommt

um Änderungen an der Oberfläche wieder an das ViewModel zurückzugeben. Im Projekt erbt jedes ViewModel von einer Basisklasse welche diese und andere für die Oberfläche nützliche Eigenschaften schon bereitstellt. In Listing 4.1 ist das Einbinden eines PropertyChangedEventHandler in C# zu sehen.

```
using System.ComponentModel;

namespace MVVMPattern.ViewModel
{
  public class ViewModelBase : INotifyPropertyChanged
  {
    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnChanged(string propertyName)
    {
      var handler = PropertyChanged;
      if (handler != null)
        handler(this, new PropertyChangedEventArgs(propertyName));
    }
  }
}
```

Listing 4.1: Binding mittels PropertyChangedEventHandler im ViewModel

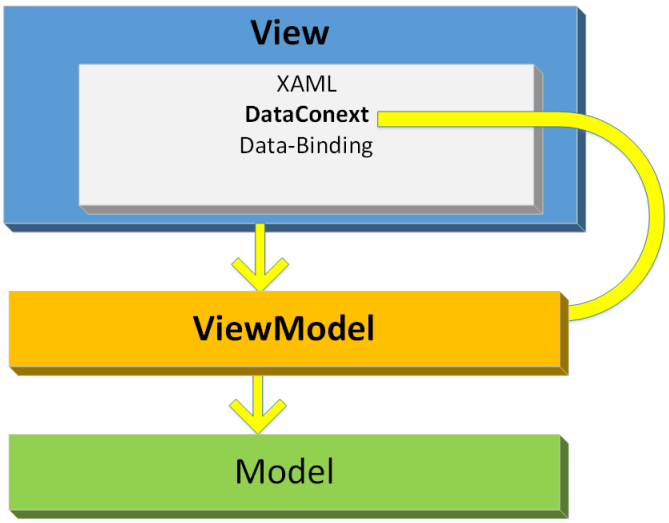

Abbildung 4.4: Das Binding eines Datacontext an Properties der Viewmodel als Schema

4.3.2 Commandkonzept

Moderne Benutzeroberflächen bieten meist verschiedene Möglichkeiten, eine Operation durchzuführen. Wenn Sie beispielsweise in Microsoft Word ein vorhandenes Dokument öffnen wollen, können Sie entweder die Menükombination Datei Öffnen oder die entsprechende Toolbar-Schaltfläche anwählen. Um eine vergleichbare Funktionalität in einer Windows Form-Anwendung umzusetzen, müssen zwei separate Event-Handler für das Click-Ereignis des Menüs und der Toolbar-Schaltfläche implementiert werden.

Beim Command-Konzept wurd nun versucht, die Abhängigkeit von grafischen Elementen und den zugehörigen Event-Handler zu entkoppeln. Zudem kann man die Steuerelemente, wie Menüs und Toolbar-Schaltflächen, deklarativ mit Commands verknüpfen. Hierbei vermeidet man Abhängigkeiten zwischen Darstellungselementen und Code-Behind-Methoden, da letztere nur an die Commands gebunden werden müssen. In Abbildung 4.5 sind beide Konzepte schematisch dargestellt. Zu beachten ist hierbei die Entkoppelung beim Command-Konzept im Vergleich zum Event-Konzept

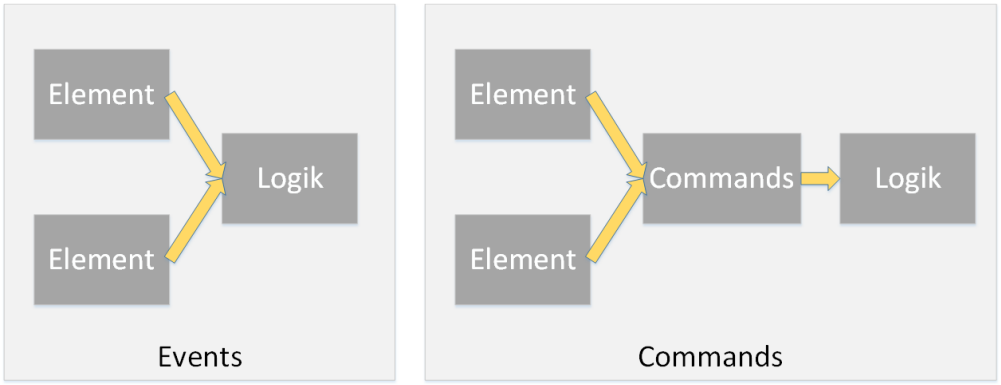

Abbildung 4.5: Vergleich Events und Commands

Listing 4.2 zeigt eine Klasse, die das Interface ICommand einbezieht und ActionCommands definiert.

```
namespace MVVMPattern.Command
{
  public class ActionCommand : ICommand
  {
```

```
    private readonly Action<object> _executeHandler;
    private readonly Func<object, bool> _canExecuteHandler;

    public ActionCommand(Action<object> execute, Func<object, bool>
        canExecute)
    {
      if (execute == null)
        throw new ArgumentNullException("Execute cannot be null");
      _executeHandler = execute;
      _canExecuteHandler = canExecute;
    }

    public event EventHandler CanExecuteChanged
    {
      add { CommandManager.RequerySuggested += value; }
      remove { CommandManager.RequerySuggested -= value; }
    }
    public void Execute(object parameter)
    {
      _executeHandler(parameter);
    }
    public bool CanExecute(object parameter)
    {
      if (_canExecuteHandler == null)
        return true;
      return _canExecuteHandler(parameter);
    }
  }
}
```

Listing 4.2: Erstellung von Actioncommand

4.3.3 XAML

Ein weiterer Vorteil moderner GUI-Konzepte wie der WPF sind die Verwendung von Deklarationssprachen zur Beschreibung der Benutzeroberflächen. In WPF wird die View standardmäßig mittels der Deklarationssprache XAML erstellt. Diese ist wie der Name bereits impliziert an XML angelehnt. Der Designer der Benutzeroberfläche kann somit alleine mit der Kenntnis von XAML eine View designen und benötigt keine Programmierkenntnisse. In Listing 4.3 ist der XAML-Code eines einfachen Formulars mit zwei Ausgabefeldern und zwei Steuerbuttons zu sehen. In Abbildung 4.6 ist die Ausgabe dazu zu sehen.

```
<Window x:Class="MVVMPattern.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:vm="clr-namespace:MVVMPattern.ViewModel"
        Title="MainWindow" SizeToContent="WidthAndHeight" >
    <Window.Resources>
```

```
        <vm:MainViewModel x:Key="mainViewModel"/>
    </Window.Resources>
    <Grid DataContext="{StaticResource mainViewModel}" Margin="10">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <TextBlock Text="Vorname:" VerticalAlignment="Bottom" Margin="2"/>
        <TextBox Text="{Binding Friends/FirstName}" Grid.Column="1" Margin=
            "2"/>
        <TextBlock Text="Nachname:" Grid.Row="1" VerticalAlignment="Bottom"
             Margin="2"/>
        <TextBox Text="{Binding Friends/LastName}" Grid.Row="1" Grid.Column
            ="1" Margin="2"/>

        <StackPanel Orientation="Horizontal" Grid.Row="3" Grid.Column="1">
            <Button Content="prev" Command="{Binding PreviousCommand}"
                Width="75" Margin="10"/>
            <Button Content="next" Command="{Binding NextCommand}" Width="
                75" Margin="10"/>
        </StackPanel>
    </Grid>
</Window>
```

Listing 4.3: Beispielcode einer einfachen auf XAML basierenden Benutzeroberfläche

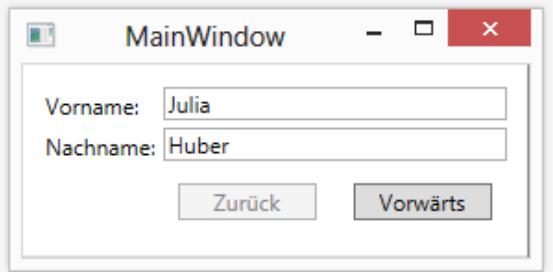

Abbildung 4.6: Beispiel einer mittels XAML erstellten Benutzeroberfläche

4.3.4 Allgemeine Eigenschaften der WPF

Um das Kapitel abzurunden sollen weitere Eigenschaften einer WPF-Anwendung im Überblick aufgelistet sein:

- Design mit der Auszeichnungssprache XAML, alternativ auch ausprogrammiert möglich.
- Unterstützung von 2D und 3D Grafiken
- Die Ausgabe ist vektorbasiert anstatt pixelbasiert. Daraus erfolgt eine bessere Skalierbarkeit der Bildschirmausgabe.
- Vielfältige Datenbindungsmöglichkeiten
- Nutzung von DirectX
- Rendern und Grafikberechnung auf der GPU anstatt CPU

Um in die Materie WPF und MVVM tiefer einzudringen sind folgende Bücher zu empfehlen: [hube12], [schw2008mic] und [geir12].
Nützliche Software zur Erstellung eines optisch hervorragenden Benutzeroberfläche ist neben dem im Visual Studio mitgelieferten Designer auch die von Microsoft angebotene Blend Express, das standardmäßig ab Visual Studio 2012 Express ebenfalls mit ausgeliefert wird.

4.4 Fazit

Das Binding bewirkt eine starke Entkopplung von View und ViewModel, die durch die Verwendung des Commandkonzepts im Vergleich zum Eventkonzept weiter verstärkt wird. Eine gewisse Modularität wird durch die strikte Trennung von UI und UI-Logik bewirkt, als Folge davon steigt die Wartbarkeit der Anwendung. Durch Einsatz von XAML ist die Trennung von Benutzeroberflächendesignern und Entwickeln sehr gut umgesetzt und hat sich bewährt. Da die UI-Logik sich im ViewModel befindet, ist sie sehr gut testbar. Die Konsequenz der Entkoppelung von View und ViewModel ist jedoch auch eine Erhöhung der Komplexität des Entwurfsmusters. Dieser Overhead macht das MVVM nicht für jede Anwendung sinnvoll. Gründe für den Einsatz von MVVM:

- Trennung von Entwickler- und Designarbeit
- viele unterschiedliche Benutzeroberflächen werden nötig
- Einsatz von WPF oder Silverlight
- gute Testbarkeit der UI-Logik nötig
- Datacontext möglich

Gründe gegen den Einsatz von MVVM:

- nur in Windows lauffähig (bei Einsatz der WPF)
- Entwicklung einer kleinen Anwendung
- kein Datacontext möglich
- kein Einsatz von WPF oder Silverlight möglich

Meine persönliche Meinung zu MVVM, besonders in Verbindung mit der WPF ist auf Grund der starken Entkopplung insgesamt positiv. Die oben genannte Problemstellung kann mit MVVM und WPF vernünftig gelöst werden.

Literaturverzeichnis

[geir12] Matthias Geirhos. *Visual C# 2012*, volume 1. Galileo Computing, 2012.

[hube12] Thomas Huber. *Windows Presentation Foundation 4.5 - Das umfassende Handbuch*, volume 5. Galileo Computing, 2010.

[schw2008mic] Holger Schwichtenberg. *Microsoft. NET 3.5 Crashkurs.*

Listings

Abbildungsverzeichnis