Hochschule für angewandte Wissenschaften
Fachhochschule Würzburg-Schweinfurt
Fakultät Informatik und Wirtschaftsinformatik

Schwerpunktseminar

Design Patterns

MVC, MVP und MVVM

Simon Niklaus

Eingereicht am: 02.12.2012

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 MVC, MVP und MVVM

1.1 MVC

Model-View-Controller - im Folgenden als MVC bezeichnet - ist sicherlich einer der am meist verwendeten Begriffe der objektorientierten Programmierung (Abb. 1.1). Ob und inwieweit dieses Paradigma eine solche Aufmerksamkeit verdient, soll in diesem Kapitel geklärt werden.

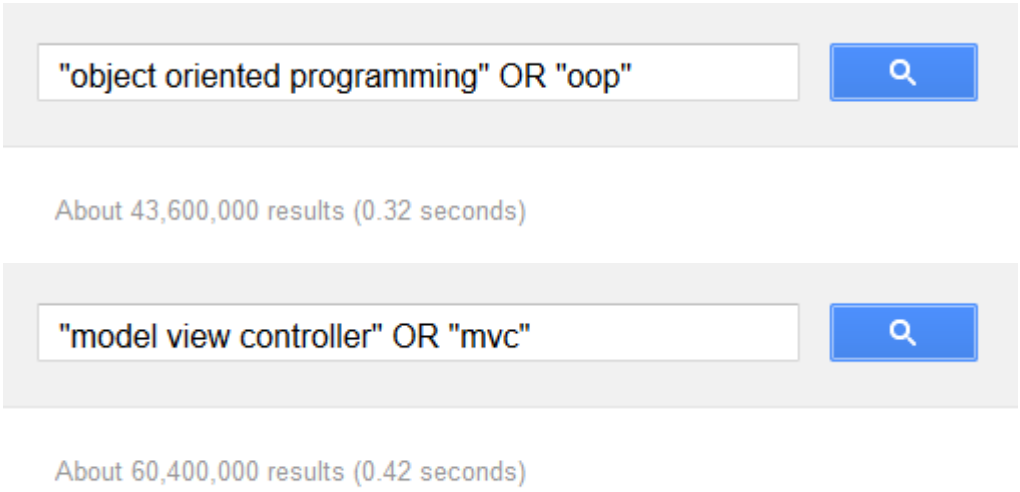

Abb. 1.1: Vergleich der Suchergebnisse bei Google

Eine Klassifizierung von MVC ist leider nicht wirklich möglich. In der Fachliteratur sind hierzu einfach zu viele verschiedene Aussagen vertreten.

> As an example of an architectural pattern in the next chapter the Model-View-Controller (MVC) pattern is introduced. [Küb00, Seite 89]

> Mit Model-View-Controller (MVC) wird ein Interaktionsmuster in der Präsentationsschicht von Software beschrieben. [Lah09, Seite 512]

> Strukturmuster: Model View Controller (MVC) [Eil10, Seite 77]

Die Gang of Four widmet der Thematik unglücklicherweise nur zwei Seiten und nimmt dabei keine konkrete Kategorisierung vor. Es wird lediglich indirekt erwähnt, dass MVC aus Patterns besteht.

> Looking at the design patterns inside MVC [...] [Gam94, Seite 4]

Bereits im Jahre 1978 wurde MVC durch Trygve Reenskaug definiert und kurz danach in die objektorientierten Programmiersprache Smalltalk aufgenommen. Grafische Benutzeroberflächen steckten zu der damaligen Zeit noch in den monochromen Kinderschuhen (Abb. 1.2), was bedeutet, dass die derzeit vorherrschenden und selbstverständlichen Konzepte zur Visualisierung noch keinerlei Bedeutung hatten.

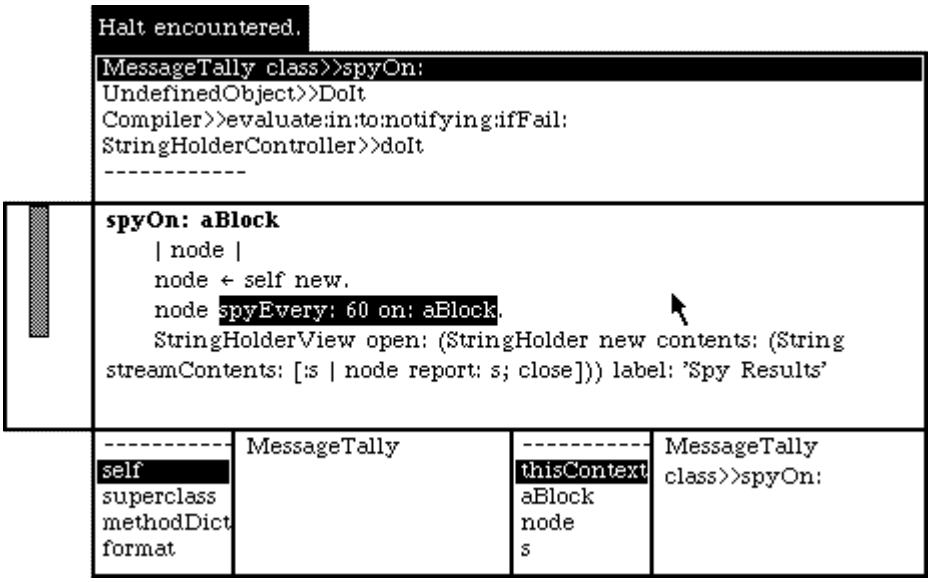

Abb. 1.2: Eine in Smalltalk implementierte grafische Benutzeroberfläche [Kri09]

Zieht man die Konsequenz daraus, kann MVC in seiner eigentlichen Form in heutigen Oberflächen nicht mehr sinnvoll eingesetzt werden. Wer also behauptet MVC zu nutzen, denunziert sich damit meist selbst.

> Probably the widest quoted pattern in UI development is Model View Controller (MVC) - it's also the most misquoted. I've lost count of the times I've seen something described as MVC which turned out to be nothing like it. Frankly a lot of the reason for this is that parts of classic MVC don't really make sense for rich clients these days. [Fow06a]

Letztendlich gibt es aber sogar bei dem Versuch MVC in seiner ursprünglichen Version zu beschreiben ein Problem. Denn selbst unter den Erfindern herrschte kein einheitlicher Konsens über die Thematik.

> Jim Althoff and others implemented a version of MVC for the Smalltalk-80

> class library after I had left Xerox PARC; I was in not involved in this work. Jim Althoff uses the term Controller somewhat differently from me. [Ree04]

Weiterhin existierte ursprünglich noch eine vierte Komponente namens Editor. Diese wird von den Erfindern in den frühen Publikationen zwar erwähnt, aber nie wirklich definiert. Schlussendlich tauchte der Editor bei den späteren Veröffentlichungen überhaupt nicht mehr auf.

> The note defines four terms; Model, View, Controller and Editor. [Ree04]

Abschließend lässt sich also sagen, dass über die Begrifflichkeit durchaus Diskrepanzen bestehen. Nicht zuletzt, weil es einfach kein Standardwerk über die Thematik gibt und selbst ein großer Teil der vorherrschenden Fachliteratur gravierende Mängel aufweist.

> If you put ten software architects into a room and have them discuss what the Model-View-Controller pattern is, you will end up with twelve different opinions. [Smi08]

> Different people reading about MVC in different places take different ideas from it and describe these as 'MVC'. [Fow06a]

Aus diesen Gründen wird im Folgenden versucht, MVC in der ursprünglichen Form wiederzugeben.

1.1.1 Ziel

Das oberste Ziel besteht in der Trennung von den Daten und der Visualisierung. Diese beiden Bereiche sind einfach so unterschiedlich, dass man sie separat behandeln sollte.

> [...] separation of presentation from model is one of the most fundamental heuristics of good software design. [Fow02, Seite 331]

Ist ein Programmierer letztendlich für das Datenmodell zuständig, wird er sich dabei primär mit den Datenbanken und der Geschäftslogik beschäftigen. Bei der Arbeit an einer grafischen Oberfläche werden dagegen die Benutzbarkeit und die Visualisierung im Fokus stehen. Diese Trennung führt so weit, dass diese sich sogar in der Berufswelt widerspiegelt.

1.1.2 Beschreibung

Wie der Name es bereits vermuten lässt, besteht das Pattern aus drei Elementen, deren Beziehung im nachfolgenden Diagramm dargestellt wird (Abb. 1.3).

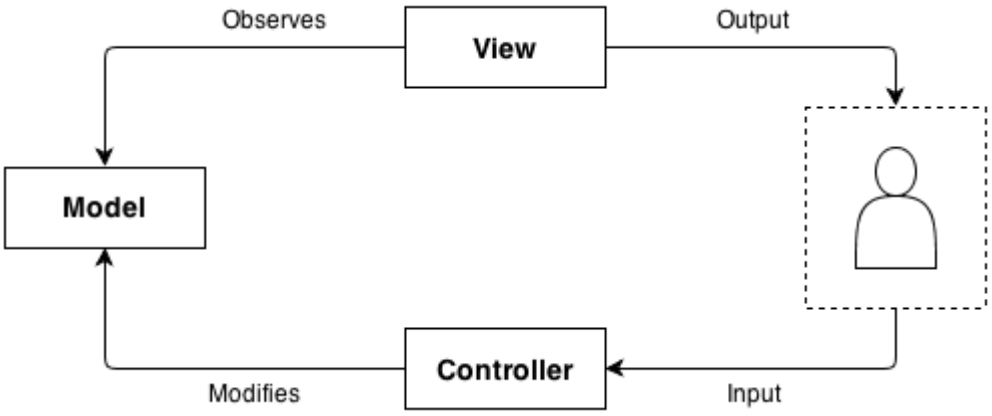

Abb. 1.3: Darstellung der Beziehungen zwischen Model, View und Controller

Bevor jedoch weiter auf diese Abbildung eingegangen wird, werden zunächst die einzelnen Elemente an und für sich betrachtet.

Model Ist für die Datenhaltung zuständig, wobei es irrelevant ist, ob die Informationen in einfachen Arrays oder einer ausgewachsenen Datenbank gespeichert werden. Nach außen müssen Funktionalitäten bereitstehen, welche einen abstrahierten Zugriff auf die hinterlegten Daten bieten. In diesem Zusammenhang ist auch vorstellbar, dass die Geschäftslogik mit in das Model integriert ist.

> Models represent knowledge. [Ree97]

View Holt sich die Informationen vom Model und präsentiert diese dem Benutzer. Dabei ist weder eine genaue Form (grafisch oder über eine Sprachausgabe) noch eine Darstellungstreue (detailgetreue oder konsolidierte Wiedergabe des Models) vorgeschrieben.

> A view is a (visual) representation of its model. It would ordinarily highlight certain attributes of the model and suppress others. It is thus acting as a presentation filter. [Ree97]

Controller Nimmt Eingaben vom Benutzer entgegen und modifiziert anschließend das Model. Wie beim View ist auch hier keine genaue Form vorgeschrieben, weshalb auch Eingabemöglichkeiten fernab von Maus und Tastatur in Betracht kommen könnten.

> A controller is the link between a user and the system. [Ree97]

Betrachtet man als nächstes die Verbindungen zwischen den einzelnen Elementen, wird zunächst ein Blick auf die Multiplizitäten geworfen (Abb. 1.4). Sowohl View, als auch Controller stellen in Bezug auf das Model mehrwertige Elemente dar. Konkret können also mehrere Views und Controller vorkommen, die alle auf ein Model zugreifen.

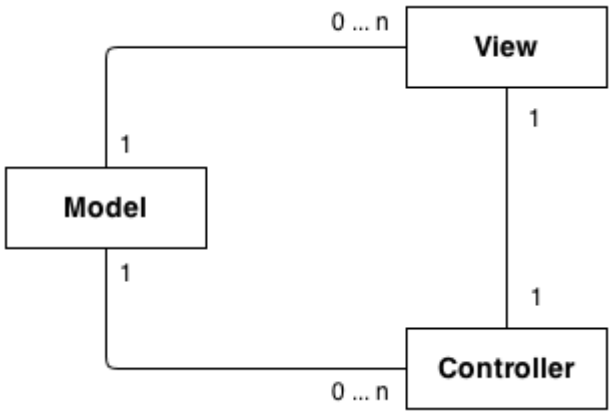

Abb. 1.4: Darstellung der Multiplizitäten von Model, View und Controller

Nachdem ein Controller aber immer nur für einen View zuständig ist, sollte es gleich viele Controller wie Views geben. Um dies zu verdeutlichen, sei auf die nachfolgende Darstellung verwiesen (Abb. 1.5). Diese zeigt exemplarisch ein Model, welches durch drei Views auf unterschiedliche Art und Weise visualisiert wird.

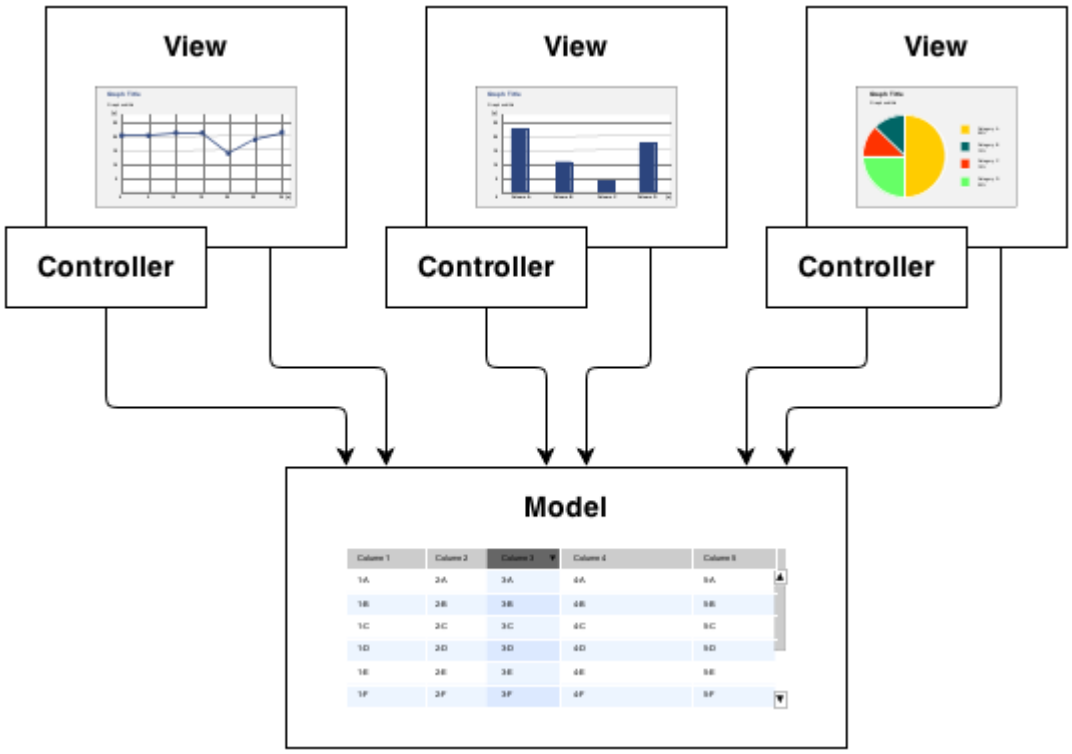

Abb. 1.5: Beispiel der Visualisierung eines Models durch verschiedene Views

Bei weiterer Betrachtung der Verbindungen fällt vor allem auf, dass der View das Model überwacht. Hier greift also das bereits behandelte Observer-Pattern (Abb. 1.6), was bei dem stattlichen Alter von MVC erstaunlich ist.

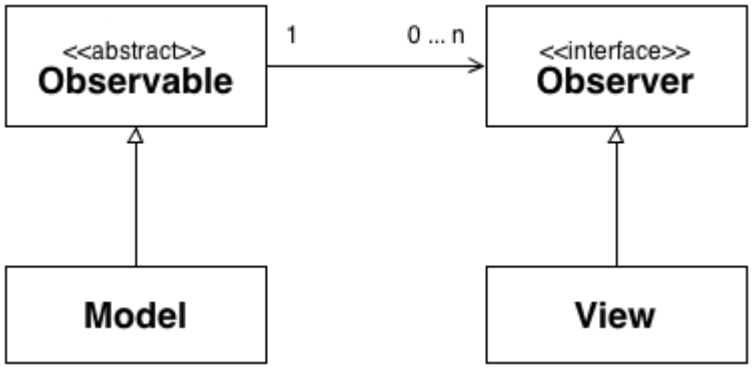

Abb. 1.6: Diagramm des Observer-Patterns im Kontext von MVC

Nimmt beispielsweise ein Nutzer eine Änderung an den Daten vor, so modifiziert im Hintergrund der Controller das Model. Im Anschluss daran greift das eben erwähnte Observer-Pattern, wodurch die zugehörigen Views benachrichtigt werden und die Änderung dementsprechend übernehmen.

In der Fachliteratur wird teilweise erwähnt, dass der Controller dem View auch Befehle er-

teilen kann. In der ursprünglichen Version wurde dies jedoch nicht beschrieben, weshalb nicht näher darauf eingegangen wird. Letztendlich ist diese Möglichkeit jedoch sowieso fraglich, da der View ohnehin über das Observer-Pattern aktualisiert wird.

> When the controller receives an action from the view, it may need to tell the view to change as a result. [Fre04, Seite 531]

Als weiteres Design-Pattern in Verbindung mit MVC ist Composite erwähnenswert. Nachdem dieses Pattern nicht in der Ausarbeitung vorkommt, soll es an dieser Stelle kurz umrissen werden. Zur besseren Erklärung wird zunächst ein Screenshot gezeigt (Abb. 1.7), welcher die hierarchische Struktur des Document Object Model einer Website darstellt.

Abb. 1.7: Grafische Darstellung des Document Object Model einer Website

Diese Abbildung soll verdeutlichen, dass sich einzelne Views immer wieder aus anderen Views zusammensetzen können. Die im Screenshot dargestellte Nachrichtenseite besteht beispielsweise unter anderem aus Beiträgen, wobei jeder einzelne Beitrag wieder aus Textfeldern und Bildern besteht.

Die Analogie zum Composite-Pattern soll im Folgenden konkret am Beispiel von AWT veranschaulicht werden (Abb. 1.8). Alle Elemente erben direkt oder indirekt von der Klasse Component und über die Klasse Composite kann eine Component wieder andere Components beherbergen. So kann beispielsweise ein Dialog aus Panels bestehen und jedes Panel wiederum aus primitiven Components wie Label oder TextField.

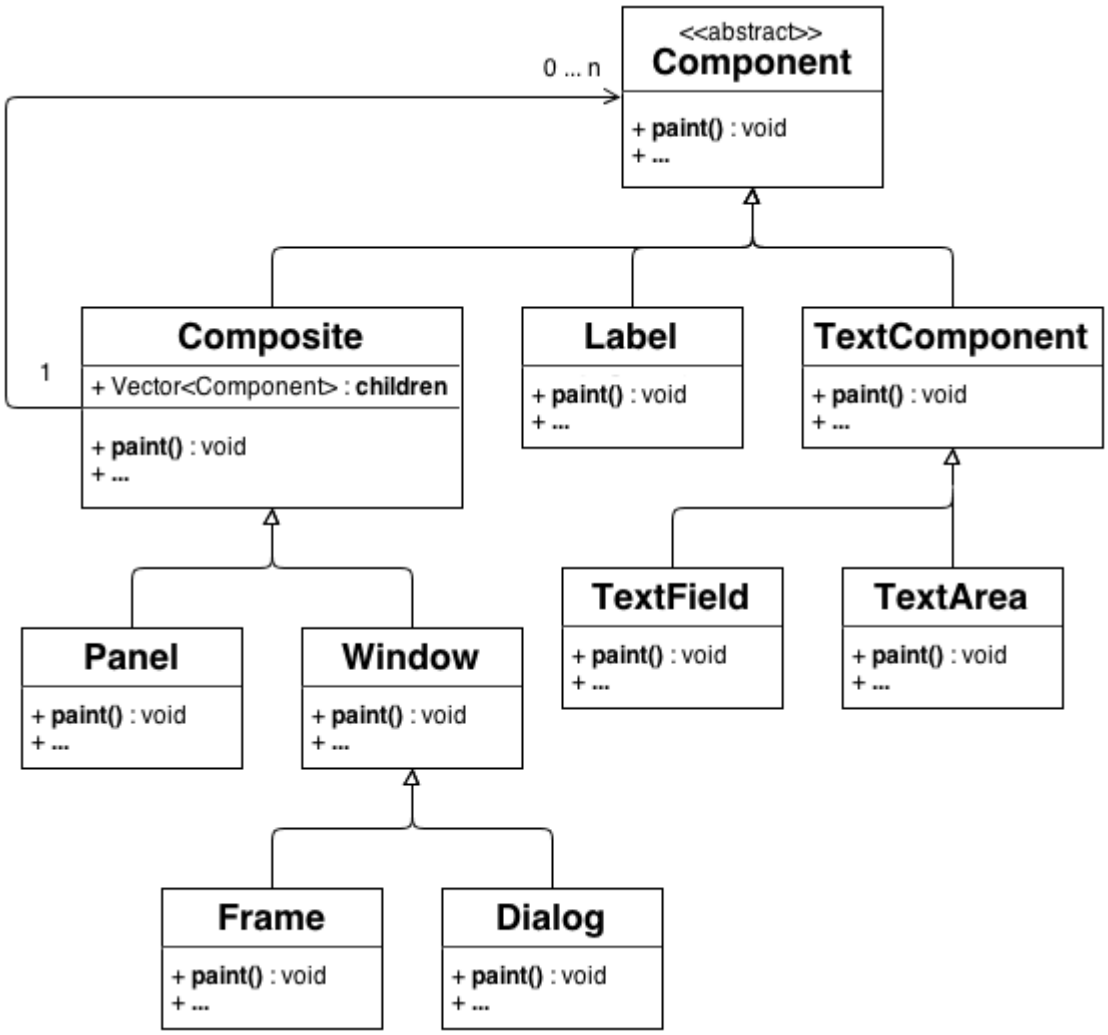

Abb. 1.8: Diagramm des Composite-Patterns im Kontext von AWT

Strategy ist das letzte Design-Pattern, welches vermehrt im Zusammenhang mit MVC genannt wird. Dieses greift beim Zusammenspiel zwischen View und Controller, allerdings wird das Pattern erst noch in einem der nachfolgenden Kapitel der Ausarbeitung erläutert. Um dem nichts vorwegzunehmen, wird es an dieser Stelle deshalb bei dem Vermerk belassen.

1.1.3 Implementierung

Der einstige Aufgabenbereich eines Controllers ist heute nicht mehr gefragt. Aus einem Handbuch von Smalltalk geht hervor, dass dieser früher primär Dinge erledigen musste, die mittlerweile vom Window-Manager übernommen werden.

> Controllers must cooperate to ensure that the proper controller is interpreting keyboard and mouse input (usually according to which view contains the cur-

sor). [Bur92]

Weiterhin sei erwähnt, dass selbst in Smalltalk die Trennung nicht immer konsequent durchgeführt worden ist.

> [...] almost every version of Smalltalk didn't actually make a view/controller separation. [Fow02, Seite 332]

Letztendlich werden Controller und View meist als eine Einheit zusammengelegt. Allerdings wirkt dies nicht dem allgemeinen Ziel, der Trennung von Daten und Visualisierung, entgegen.

> The separation of view and controller is less important, so I'd only recommend doing it when it is really helpful. For rich-client systems, that ends up being hardly ever [...] [Fow02, Seite 333]

> Sowohl bei den Microsoft Foundation Classes (MFC) als auch bei der Java-Oberflächenbibliothek Swing ist denn auch der Controller keine eigenständige Entität mehr. [Lah09, Seite 516]

Um dies praxisnah zu veranschaulichen, sei auf die Listener bei Java verwiesen (Listing 1.1), welche prinzipiell das Observer-Pattern darstellen. Durch Listener werden schließlich Aktionen, die bei der Interaktion mit der grafischen Benutzeroberfläche auftreten, direkt an den View gebunden.

```
JButton jbuttonHandle = new JButton("Der Button");

jbuttonHandle.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent arg0) {
    // Der Button wurde betätigt.
  }
});
```

Listing 1.1: Beispiel eines Listeners bei Java

Nicht zuletzt aus dem eben genannten Grund wird auf eine Beispielimplementierung an dieser Stelle verzichtet. Alternativ wird auf die Implementierung des Observers im entsprechenden Kapitel verwiesen. Letztendlich stellt das Observer-Pattern ohnehin den Kernaspekt von MVC dar.

1.1.4 Bewertung

Vorteile

- Anpassbare Visualisierung

Unterschiedliche Nutzergruppen fordern angepasste Visualisierungen, welche aber alle auf dieselben Daten zugreifen. Durch die Trennung kann für verschiedene Einsatzzwecke ein eigener View konzipiert werden - das Model dahinter bleibt aber für alle gleich.

- Verbesserte Wartbarkeit

Muss beispielsweise ein Teil an der Geschäftslogik oder an der Datenhaltung geändert werden, erfordert dies keinen tief greifenden Eingriff in die Visualisierung.

- Vereinfachte Testbarkeit

Eine grafische Oberfläche als Ganzes lässt sich nur schwer testen, da die programmatische Kommunikation mit der Visualisierung sehr mühsam ist. Ist durch Trennung von den Daten und der Visualisierung das Model separat zugänglich, so kann man dieses ohne Umwege auf Fehler testen.

Nachteile

- Höherer Implementierungsaufwand

Teilweise wird in der Fachliteratur erwähnt, dass die Nutzung von MVC einen erhöhten Implementierungsaufwand mit sich bringt.

> Der Implementierungsaufwand erhöht sich. [Eil10, Seite 79]

Es ist aber wohl legitim, dies kritisch zu sehen. Schließlich wird man sich ohne die grundsätzliche Trennung von den Daten und der Visualisierung schwer tun, weshalb also davon auszugehen ist, dass sich die Implementierung ohne diese klare Struktur ebenso erschweren kann.

1.2 MVP

Model-View-Presenter - im Folgenden als MVP bezeichnet - ist ein noch relativ selten verwendeter Begriff (Abb. 1.9). Nachdem es bei der Thematik aber prinzipiell um den Missstand des mittlerweile überflüssigen Controllers geht, verwundert dies doch etwas.

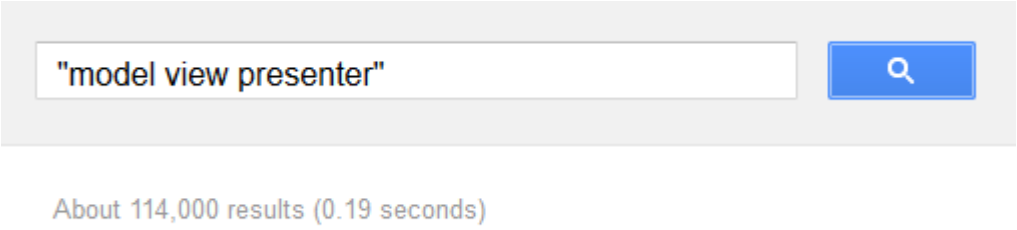

Abb. 1.9: Suchergebnisse bei Google

Wie auch bei MVC ist eine Klassifizierung von MVP nicht wirklich möglich und die Gang of Four hat diese Thematik gänzlich nicht behandelt. Allerdings wurde das Thema auch erst ein paar Jahre nach diesem Standardwerk in Publikationen aufgegriffen.

Letztendlich wurde MVP erstmalig durch Mike Potel in einer Veröffentlichung im Jahre 1996 erwähnt (Abb. 1.10). Allerdings ist die Bezeichnung dort relativ irreführend, da der Controller zwar durch einen Presenter ersetzt, aber zusätzlich noch Command, Selection und Interactor eingeführt wurden.

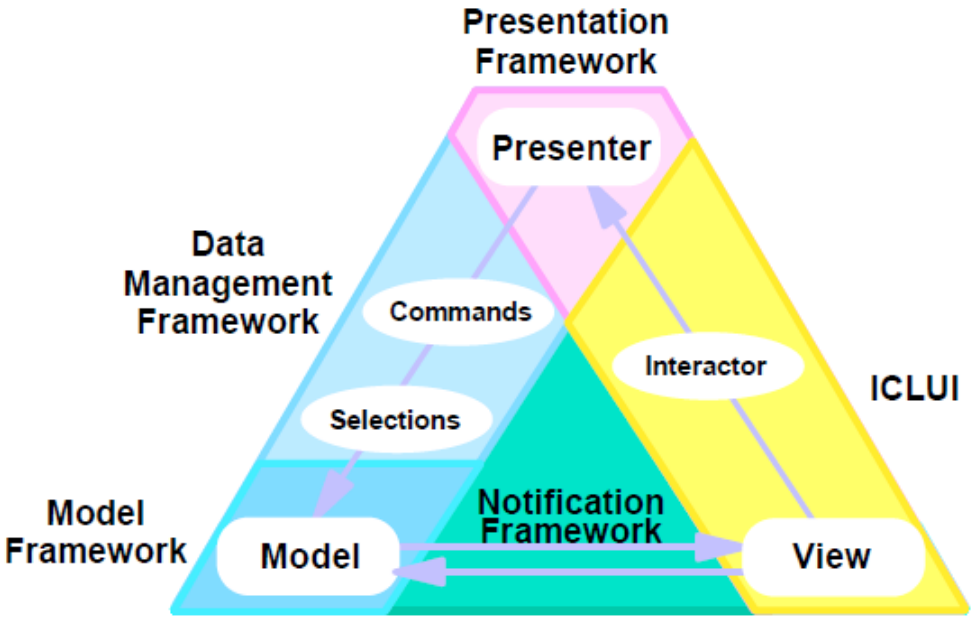

Abb. 1.10: Das von Mike Potel vorgeschlagene Programmiermodell [Pot96]

Über die Jahre wurde auch MVP auf verschiedenste Art und Weise interpretiert, was wieder zu einer Diskrepanz bezüglich der Begrifflichkeiten führte. Allerdings verfasste Martin Fowler einen Artikel, welcher mittlerweile als Standardwerk in Bezug auf MVP gilt. Er unterteilte hierbei MVP jedoch noch einmal in zwei verschiedene Patterns.

> [...] I decided that pattern that was here under the name 'Model View Presenter' needed to be split, so I have separated it into Supervising Controller and Passive View. [Fow06c]

Um den Umfang dieses Kapitel nicht unnötig auszuweiten, wird im nachfolgenden nur auf den Passive View eingegangen, welcher im Allgemeinen häufiger verwendet wird.

1.2.1 Ziel

Dem obersten Ziel, der Trennung von Daten und Visualisierung, wird bei MVP nichts hinzugefügt. Insofern kann entsprechend auf MVC verwiesen werden.

1.2.2 Beschreibung

Wie auch MVC, besteht das Pattern aus drei Elementen, deren Beziehung im nachfolgenden Diagramm dargestellt wird (Abb. 1.11).

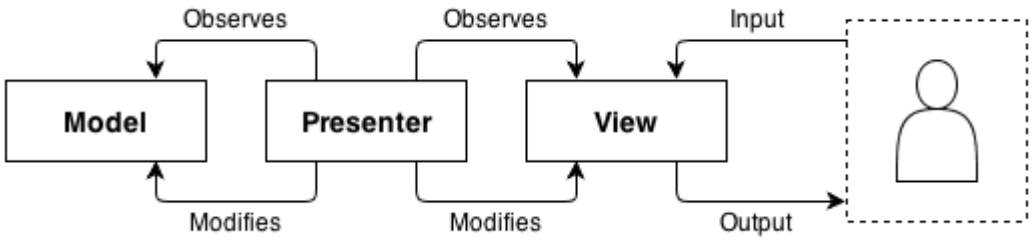

Abb. 1.11: Darstellung der Beziehungen zwischen Model, View und Presenter

Bevor jedoch weiter auf diese Abbildung eingegangen wird, werden zunächst die einzelnen Elemente an und für sich betrachtet.

Model | Ist weiterhin für die Datenhaltung zuständig und kann die Geschäftslogik beinhalten. Insofern gibt es gegenüber MVC keine Änderung.

View | Wird aus programmiertechnischer Sicht so einfach wie möglich gehalten und kümmert sich eigentlich nur noch um das letztendliche Visualisieren und das Entgegennehmen von Eingaben. Über das Model besitzt er keine direkte Kenntnis mehr.

> The significant change with Passive View is that the view is made completely passive and is no longer responsible for updating itself from the model. [Fow06b]

Presenter | Stellt die Verbindung zwischen Model und View her, wobei er eine steuernde Rolle einnimmt.

Betrachtet man als nächstes die Verbindungen zwischen den einzelnen Elementen, wird zunächst ein Blick auf die Multiplizitäten geworfen (Abb. 1.4). Konkret sollten Views und Presenter immer paarweise existieren, wobei mehrere Presenter auf ein Model zugreifen.

Abb. 1.12: Darstellung der Multiplizitäten von Model, View und Presenter

Zur Verdeutlichung sei auf die nachfolgende Darstellung verwiesen (Abb. 1.13). Diese zeigt exemplarisch ein Model, welches durch drei Views auf unterschiedliche Art und Weise visualisiert wird.

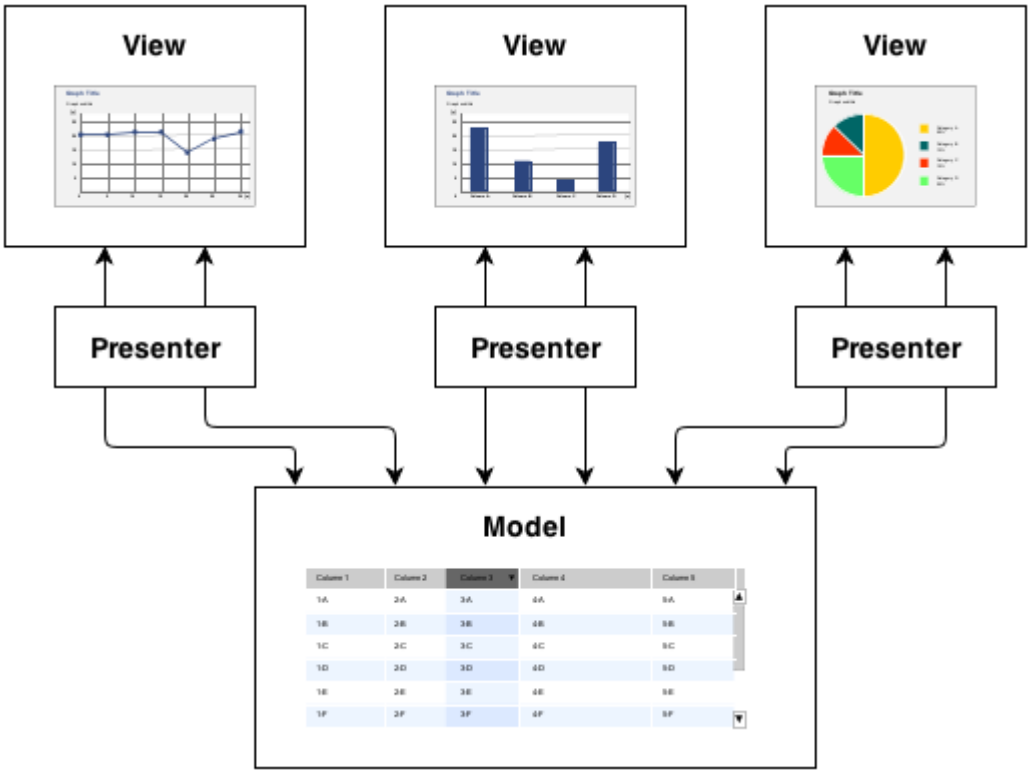

Abb. 1.13: Beispiel der Visualisierung eines Models durch verschiedene Views

Bei weiterer Betrachtung der Verbindungen fällt wie auch bei MVC vor allem das Observer-Pattern auf. Der Vollständigkeit halber und zum besseren Verständnis wird noch einmal kurz auf dieses Pattern eingegangen (Abb. 1.14).

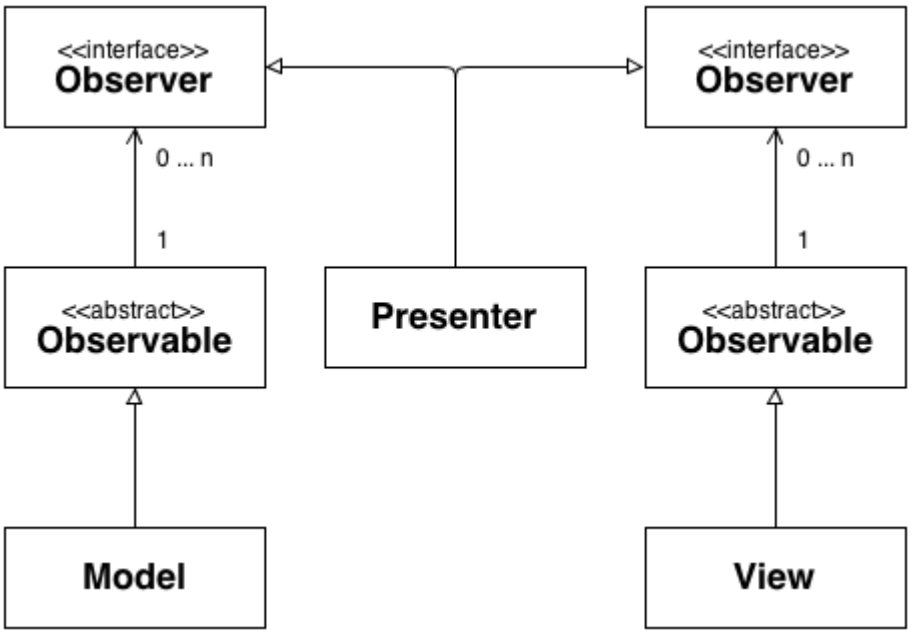

Abb. 1.14: Diagramm des Observer-Patterns im Kontext von MVP

Nimmt beispielsweise ein Nutzer eine Änderung an den Daten vor, so wird im Hintergrund der Presenter durch das Observer-Pattern über diese Änderung informiert und aktualisiert das Model dementsprechend. Im Anschluss daran wird der Presenter über diese Aktualisierung am Model ebenfalls durch das Observer-Pattern in Kenntnis gesetzt und modifiziert den View.

Bezüglich der Patterns Composite und Strategy kann an dieser Stelle wieder entsprechend auf MVC verwiesen werden, da diese analog übertragen werden können.

1.2.3 Implementierung

Anders als MVC ist der Einsatz von MVP durchaus realistisch, weshalb an dieser Stelle eine Beispielimplementierung präsentiert wird. Aus Gründen der Einfachheit wird hierbei aber nur ein elementarer Integer-Wert durch zwei verschiedene Views dargestellt (Abb. 1.15).

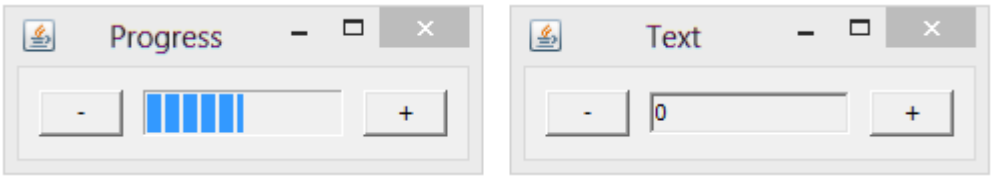

Abb. 1.15: Screenshot der Beispielimplementierung

Trotz der Einfachheit des Beispiels ist das Klassendiagramm bereits relativ umfangreich (Abb. 1.15). Allerdings wird dies nicht sonderlich komplexer, sollte das Model erweitert werden. Dementsprechend ist nur der initiale Aufwand erhöht.

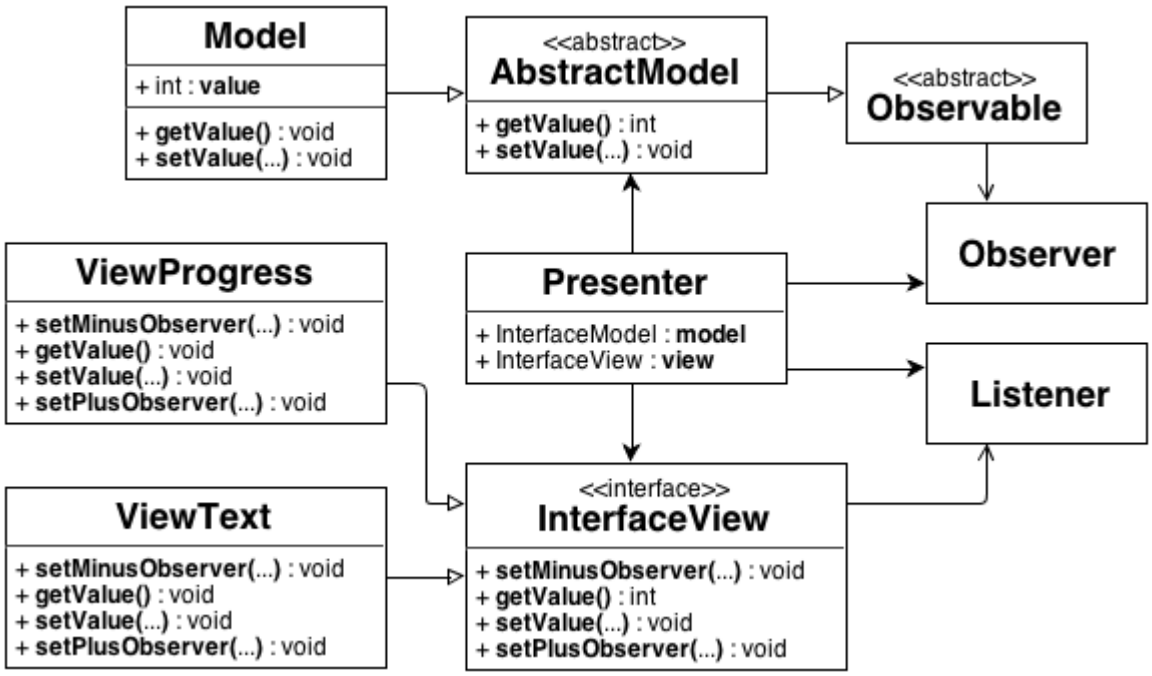

Abb. 1.16: Klassendiagramm der Beispielimplementierung

Bei näherer Betrachtung des Klassendiagramms fallen vor allem AbstracModel und InterfaceView auf, welche der Presenter referenziert. Bei MVP abstrahiert man in der Regel die Methoden vom Model und den Views durch eben solche Elternelemente, um wohl definierte Schnittstellen zu erhalten. Dies hat den Vorteil, dass man wie in der Beispielimplementierung nur eine Presenter-Klasse benötigt, da der Zugriff auf die Views einheitlich ist. Zur Laufzeit wird dann aber natürlich eine Instanz der Presenter-Klasse für jede Instanz der View-Klassen erstellt.

Nachdem AbstractModel von der Observable-Klasse erbt, kann diese Schnittstelle nur durch unschöne Tricks zu einem Interface gewandelt werden. Nachdem Interfaces logischerweise nicht von Klassen erben können, müsste das entsprechende Interface die Methoden des Observable beinhalten und zusätzlich die Model-Klasse von Observable erben.

Der Sourcecode des Beispiels wird an dieser Stelle nicht präsentiert, da er dieses Kapitel nur unnötig aufblähen würde und dieser sowieso als fertiges Eclipse-Projekt mitgeliefert wird. Zum besseren Verständnis ist es sinnvoller, den Verlauf einer Interaktion des Benutzers in einem Timingdiagramm zu verfolgen (Abb. 1.17).

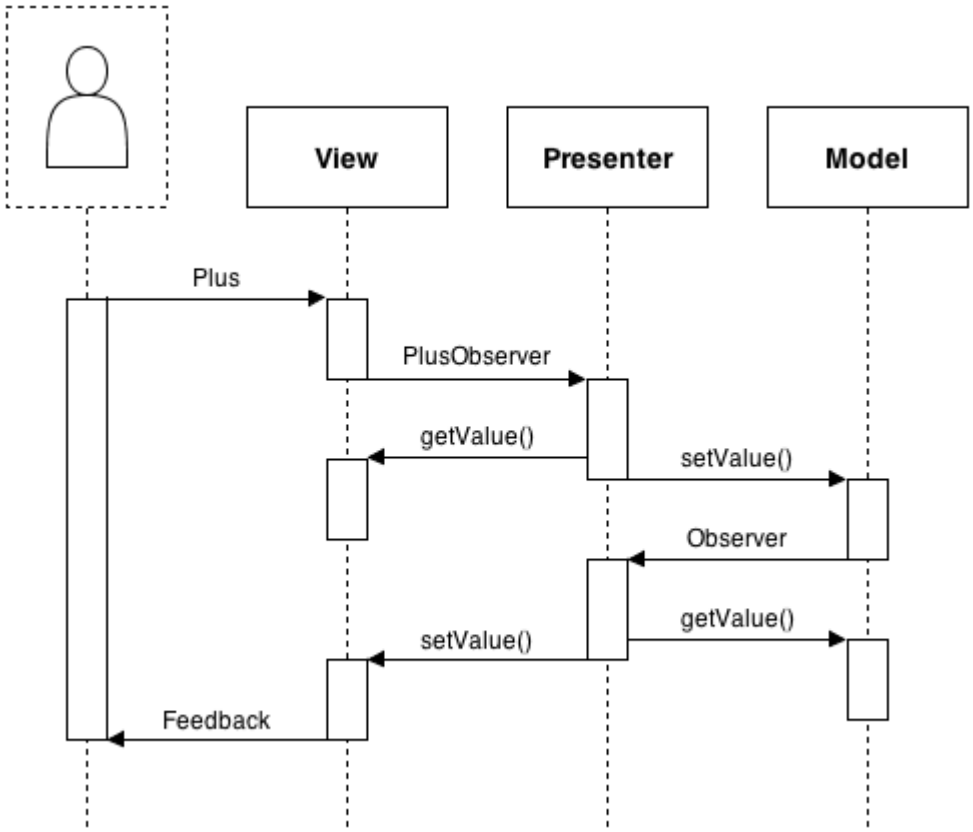

Abb. 1.17: Timingdiagramm für das Betätigen des Plus-Buttons

Durch das Betätigen des Plus-Buttons informiert der aktive View seinen zugehörigen Presenter durch ein Observer-Pattern (konkret als Listener implementiert), woraufhin der Presenter den aktuellen Wert vom View bezieht, diesen inkrementiert und das Model dementsprechend aktualisiert. Anschließend wird der Presenter vom Model wieder durch ein Observer-Pattern über eine Änderung benachrichtigt. Der Presenter holt sich sodann den Wert vom Model und aktualisiert den View, wodurch der Benutzer den aktuellen Wert aus dem Model erhält.

1.2.4 Bewertung

Generell lässt sich sagen, dass MVP eine zeitgemäße Variante des ursprünglichen MVC darstellt, was schließlich auch die Beispielimplementierung gezeigt hat. In diesem Zusammenhang treffen die Vor- und Nachteile, welche MVC mit sich bringt, im Großen und Ganzen auch auf MVP zu. Aus diesem Grund werden diese im Folgenden nur ergänzt und es kann an dieser Stelle wieder entsprechend auf MVC verwiesen werden.

Vorteile

- Sehr gute Testbarkeit

Nachdem die Views nur sehr einfach gestrickt sind, lassen sich diese durch Mock-Objekte ersetzen, welche auf einfache Art und Weise den dazugehörigen Presenter und das Model testen. Da eine solche Klasse relativ kompakt ist, wird zur besseren Veranschaulichung der Sourcecode einer solchen im Zusammenhang mit der Beispielimplementierung präsentiert (Listing 1.2).

```
public class ViewMock implements InterfaceView {
  public ActionListener observerMinus;

  public int value;

  public ActionListener observerPlus;

  public void test() {
    System.out.println("All right, paradox time. This - sentence - is - false!");

    {
      this.value = 0;
      this.observerMinus.actionPerformed(null);
      if (this.value != -1) {
        System.out.println("Test failed!");
      }
    }
    {
      this.value = -10;
      this.observerMinus.actionPerformed(null);
      if (this.value != -10) {
        System.out.println("Test failed!");
      }
    }
    {
      this.value = 0;
      this.observerPlus.actionPerformed(null);
      if (this.value != 1) {
        System.out.println("Test failed!");
      }
    }
    {
      this.value = 10;
      this.observerPlus.actionPerformed(null);
      if (this.value != 10) {
        System.out.println("Test failed!");
      }
    }

    System.out.println("Two plus two is ... ten. In base four! I'm fine!");
  }

  public void setMinusObserver(ActionListener observer) { this.observerMinus = observer; }

  public int getValue() { return this.value; }

  public void setValue(int value) { this.value = value; }

  public void setPlusObserver(ActionListener observer) { this.observerPlus = observer; }
}
```

Listing 1.2: Beispiel eines Mock-Objekts

Die gezeigte Klasse befindet sich ebenfalls im fertigen Eclipse-Projekt und kann separat ausgeführt werden.

Nachteile

- Schwieriger zu Debuggen

Durch die vielen Observer-Patterns und den damit einhergehend verschachtelten Methodenaufrufen erschwert sich das Debuggen. Schließlich wird oft zwischen den einzelnen Elementen von MVP hin und hergesprungen, weshalb die Übersichtlichkeit einfach verloren geht.

1.3 MVVM

Um den Rahmen dieses Kapitels nicht übermäßig auszudehnen, wird im Folgenden nur auf einen interessanten Kernaspekt von Model-View-ViewModel - im Folgenden als MVVM bezeichnet - eingegangen, weshalb die bisherige Kapitelstruktur verworfen wird.

Nachdem der View bei MVP eigentlich keine Logik mehr besitzt, liegt es nahe, diesen in einer Markupsprache zu definieren. Vor allem für Gestaltungstools ist dies eine geschickte Methode, da diese so einfacher zu realisieren sind. Weiterhin hat HTML bereits gezeigt, dass sich vollwertige Anwendungen durch Markupsprache effizient darstellen lassen.

Letztendlich wird dieses Kapitel mit einem Beispiel abgeschlossen, welches exemplarisch einen View zeigt, wie er in Android definiert wird (Listing 1.3).

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="10dp"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/textviewImage"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="5dp"
        android:layout_marginRight="5dp"
        android:layout_marginTop="5dp"
        android:text="Image:"
        android:textAppearance="?android:attr/textAppearanceLarge" />

    <ImageView
        android:id="@+id/imageviewImage"
        android:layout_width="fill_parent"
        android:layout_height="200dp"
        android:layout_margin="5dp"
        android:src="@drawable/cake" />

    <TextView
        android:id="@+id/textviewRating"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="5dp"
        android:layout_marginRight="5dp"
        android:layout_marginTop="5dp"
        android:text="Rating:"
        android:textAppearance="?android:attr/textAppearanceLarge" />

    <RatingBar
        android:id="@+id/ratingbarRating"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="5dp"
        android:progress="90" />

</LinearLayout>
```

Listing 1.3: Definition eines View in einer Metasprache

Schließlich führt ein so definierter View zu folgendem Ergebnis (Abb. 1.18). Dabei sei erwähnt, dass dieses Beispiel in Gänze mit den Entwicklungstools von Android erstellt wurde und sogar der Screenshot aus diesen Tools stammt.

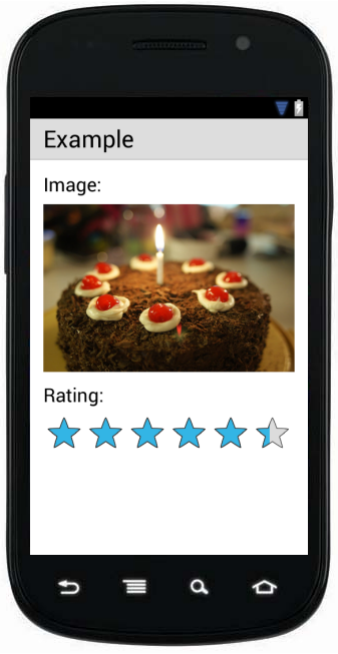

Abb. 1.18: Screenshot des generierten View

Literaturverzeichnis

[Bur92] Burbeck, S.: *Applications Programming in Smalltalk-80: How to use Model-View-Controller (MVC)*, http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html, 1992.

[Eil10] Eilebrecht, K.; Starke, G.: *Patterns kompakt: Entwurfsmuster für effektive Software-Entwicklung*, 2010.

[Fow02] Fowler, M.: *Patterns of Enterprise Application Architecture*, 2002.

[Fow06a] Fowler, M.: *GUI Architectures*, http://martinfowler.com/eaaDev/uiArchs.html, 2006.

[Fow06b] Fowler, M.: *Passive View*, http://martinfowler.com/eaaDev/PassiveScreen.html, 2006.

[Fow06c] Fowler, M.: *Retirement note for Model View Presenter Pattern*, http://martinfowler.com/eaaDev/ModelViewPresenter.html, 2006.

[Fre04] Freeman, E.; Freeman, E.; Sierra, K.; Bates, B.: *Head First Design Patterns*, 2004.

[Gam94] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, 1994.

[Küb00] Kübler; Rieck; Stanullo; Vollmer; Weiss: *Ferienkurs Design Patterns*, 2000.

[Kri09] Krivanek, P.: *Vyvojove prostredi davnoveku - Smalltalk-80*, http://abclinuxu.cz/clanky/programovani/vyvojove-prostredi-davnoveku-smalltalk-80, 2009.

[Lah09] Lahres, B.; Rayman, G.: *Objektorientierte Programmierung: Einstieg und Praxis*, 2009.

[Pot96] Potel, M.: *MVP: Model-View-Presenter*, http://wildcrest.com/Potel/Portfolio/mvp.pdf, 1996.

[Ree97] Reenskaug, T.: *MODELS - VIEWS - CONTROLLERS*, http://heim.ifi.

uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf, 1997.

[Ree04] Reenskaug, T.: *MVC: XEROX PARC 1978-79*, http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html, 2004.

[Smi08] Smith, J.: *Using MVC to Unit Test WPF Applications*, http://codeproject.com/Articles/23241/Using-MVC-to-Unit-Test-WPF-Applications, 2008.