Advanced Software Engineering

Programmentwurf

von

Lukas Priester

Abgabedatum:	06.06.2021
Matrikelnummer, Kurs:	6634887, TINF18B2

Inhaltsverzeichnis

Abkürzungsverzeichnis

ATRIP	Automatic, Thorough, Repeatable, Independent, Professional
DRY	Don't Repeat Yourself
GRASP	General Responsibility Assignment Software Patterns
OOP	Objekt-Orientierte Programmierung
SOLID	Single Responsibility-, Open Closed-, Liskov Substitution-, Interface Segregation-, Dependency Inversion Principle
UML	Unified Modelling Language

Abbildungsverzeichnis

Listenverzeichnis

1. Einführung

Der Programmentwurf implementiert eine Software zur Verwaltung von Finanztransaktionen. Die Applikation wird intern als *FinanceManager* betitelt.
Das Programm bietet grundlegende Möglichkeiten zum Pflegen von Transaktionen (Transactions) in verschiedenen Konten (Accounts). Der *FinanceManager* besitzt einen Übersichtstab (Abbildung A.1), in welchem die Kontostände der einzelnen Accounts summiert dargestellt wird.
Der Tab, welcher ein einzelnes Konto darstellt (Abbildung A.2), zeigt eine Übersicht über die Transaktionen in tabellarischer Form (Abbildung A.3), sowie den Kontostand. Hier findet sich auch die Möglichkeit neue Transaktionen über ein Dialogfenster dem Account hinzuzufügen (Abbildung A.4).
Die Möglichkeit, einen neuen Account dem *FinanceManager* hinzuzufügen, wird auch über ein Dialogfenster (Abbildung A.5) bereitgestellt.
Der *FinanceManager* ist in Java implementiert und nutzt Maven als Build-Framework. Es wird Java 16 genutzt. Die Bibliothekten `gson`, `LGoodDatePicker` und `radiance-substance` erweitern die Funktionalität der Anwendung. Als Test-Framework wird `JUnit 4` genutzt.

2. Programmierprinzipien

Im Folgenden werden die Programmierprinzipien Single Responsibility-, Open Closed-, Liskov Substitution-, Interface Segregation-, Dependency Inversion Principle (SOLID), General Responsibility Assignment Software Patterns (GRASP) und Don't Repeat Yourself (DRY) erläutert und ihre Umsetzung im Programmentwurf dargelegt.

2.1. SOLID

Das SOLID Programmierprinzip fasst verschiedene Prinzipien zusammen. SOLID wurde von Robert C. Martin eingeführt, mit dem Ziel, die enge Kopplung in der Objekt-Orientierte Programmierung (OOP) zu reduzieren.

2.1.1. Beschreibung der Prinzipien

Im Folgenden werden die einzelnen Prinzipien aus denen sich SOLID zusammensetzt, näher beschrieben.

Single Responsibility Beschreibt das eine Klasse nur eine Ursache oder einen Grund haben sollte sich zu ändern. Des Weiteren hat jede Klasse nur eine Zuständigkeit.

Open Closed Beschreibt das eine Klasse oder Funktion offen für Erweiterungen aber geschlossen für Änderungen sein soll. Praktisch soll die Erweiterung durch Vererbung beziehungsweise durch die Implementierung eines Interfaces erreicht werden.

Liskov Substitution Beschreibt das Ableitungsverhalten von Objekten. Abgeleitete Klassen müssen hierbei schwächere Vorbedingungen und stärkere Nachbedingungen haben.

Interface Segregation Beschreibt die Aufteilung von Interfaces, sodass schwere Interfaces und Klassen, welche viel Funktionalität bündeln, verhindert werden.

Dependency Inversion Beschreibt die Abhängigkeit von Modulen. Hierbei soll vermieden werden, dass High-Level Module von Low-Level Modulen abhängig sind.

2.1.2. Analyse des Programmentwurfs

Der vorliegende Programmentwurf implementiert einige der Prinzipien von SOLID. Das Single Responsibility-Prinzip tritt hierbei am häufigsten auf. Beispiel hierfür sind sowohl die `JsonReader`, als auch die `JsonWriter`-Klasse.

Die Modellierungsklassen `Account` und `Transaction` können zur Einhaltung des Open Closed-Prinzips vorgezeigt werden. Die beiden Klassen sind zwar nicht durch Interfaces implementiert, haben jedoch einen leicht erweiterbaren Aufbau.

Liskov Substitution-Prinzip wird nicht genutzt, da innerhalb der Anwendung Objekte nicht voneinander Ableiten.

Interface Segregation wird bei den Klassen `FinanceManagerController` und `FinanceManagerWindow` betrieben. Diese beruhen zwar nicht auf Interfaces, teilen sich aber die Funktionalität untereinander auf. Sie sind beiden für den Ablauf der Anwendung wichtig, wobei der Controller während des Starts der Anwendung und dem initialen Laden von Daten und die Window-Klasse während der Nutzung der eigentlichen Funktionalität der Anwendung genutzt wird.

2.2. GRASP

GRASP beschreibt Prinzipien, auf welchen verschiedene Entwurfsmuster aufbauen. GRASP zielt darauf ab die Lücke zwischen Domänenmodell und Implementierung möglichst klein zu halten.

2.2.1. Low Coupling

Die Kopplung beschreibt die Abhängigkeit zweier Objekte. Vorteile von geringer beziehungsweise loser Kopplung ist, dass Objekte durch Änderungen in anderen Objekten nicht so stark beeinflusst werden. Zudem sind die Objekte einfacher testbar und verständlicher, da sie weniger Kontext benötigen. Durch die geringe Abhängigkeit können solche Objekte einfach wiederverwendet werden.

Beispiele für lose Kopplung im Programmentwurf sind die Klassen `InputValidator`, `JsonReader` und `JsonWriter`. Diese Klassen sind nicht von anderen Objekte des Programmentwurfs abhängig und somit in ihrem jeweiligen Kontext isoliert. Dies macht sie zu einfach testbaren Klassen. Zudem haben interne Änderungen der Klassen keine direkten Auswirkungen auf andere Objekte oder Klassen des Programmentwurfs.

2.2.2. High Cohesion

Kohäsion beschreibt den Zusammenhalt einer Klasse. Es geht also um die semantische Nähe der Attribute einer Klasse. Zusammen mit der losen Kopplung bilden diese beiden Prinzipien ein gutes Fundament für Code. Durch eine hohe Kohäsion wird zudem ein verständlicheres und wiederverwendbares Design erreicht.

Im Programmentwurf wird eine hohe Kohäsion unter anderem im `AccountPanel` umgesetzt. Hierbei findet die Datenhaltung eines Panels in einem `AccountPanelData`-Objekt statt. Hierdurch wird zum einen die Komplexität des `AccountPanel` reduziert und zum anderen die Funktionalität beziehungsweise die Verarbeitung der Daten in die `AccountPanelData`-Klasse ausgelagert.

2.3. DRY

DRY bedeutet übersetzt Wiederhole dich nicht. Genau das soll durch das Programmierprinzip umgesetzt werden. Duplikationen von Code aber auch von Dokumentation sollen vermieden werden. Es soll nur eine Quelle der Wahrheit im System geben. Der Aufbau

ist hierbei mit einem relationalen Datenbankmanagementsystem vergleichbar. Durch die Nutzung von DRY können bei einer Änderung einer Funktion auch alle Objekte, welche diese nutzen, geändert werden.

Als Beispiel für DRY kann die `InputValidator`-Klasse genommen werden. Die Funktion dieser Klasse wird bei allen Eingabemöglichkeiten der Anwendung eingesetzt, um zu überprüfen, ob die Eingabe dem vom Programm erwarteten Typ entspricht. Findet eine Änderung der Validierungslogik innerhalb dieser Klasse statt, so ändert sich auch das Verhalten bei allen Eingabemöglichkeiten. Durch die Nutzung einer zentralen Klasse wird zudem gewährleistet, dass die Eingabevalidierung in der gesamten Anwendung konsistent und nach den gleichen Regeln abläuft.

3. Entwurfsmuster

Observer pattern erklären und wie es im Code umgesetzt wurde, uml-diagramm vorher nachher (git history)
Der *FinanceManager* setzt das *Observable* Entwurfsmuster um. Obwohl das *Observable* seit Java 9 deprecated ist, wurde sich hierfür entschieden, da der *Observer* nur darüber informiert werden muss, dass sich etwas im *Observable* verändert hat, jedoch nicht was. Der *FinanceManager* bietet dem Nutzer mehrere Tabs an. Einer dieser Tabs ist der Overview-Tab. Die restlichen Tabs repräsentieren einzelne Accounts, welche der Nutzer pflegen kann. Der Overview-Tab ist der *Observer*. Die Daten, welche die einzelnen Account-Tabs halten, die `AccountPanelData`, sind die *Observable*. Sollte sich der Kontostand (`balance`) eines `AccountPanelData` ändern, wird der *Observer* darüber informiert, dass sich etwas verändert hat. Da dieser den über alle Accounts summierte Kontostand darstellt, berechnet er, sobald er über eine Änderung informiert wurde, diesen Kontostand neu.

Das Unified Modelling Language (UML)-Diagramm in Abbildung 3.1 zeigt den Zusammenhang der Klassen bevor das *Observer* Entwurfsmuster eingeführt wurde. Die Klassenstuktur nach der Einführung des *Observer* Entwurfmusters zeigt die Abbildung 3.2. Die `FinanceManagerPane` orchestriert hierbei die Weitergabe des *Observable* vom `AccountPanel` zum `OverviewPanel` (Funktion `addObservable(Observable)`). Diese Weitergabe geschieht jedes Mal, wenn ein neues `AccountPanel` erstellt wird.

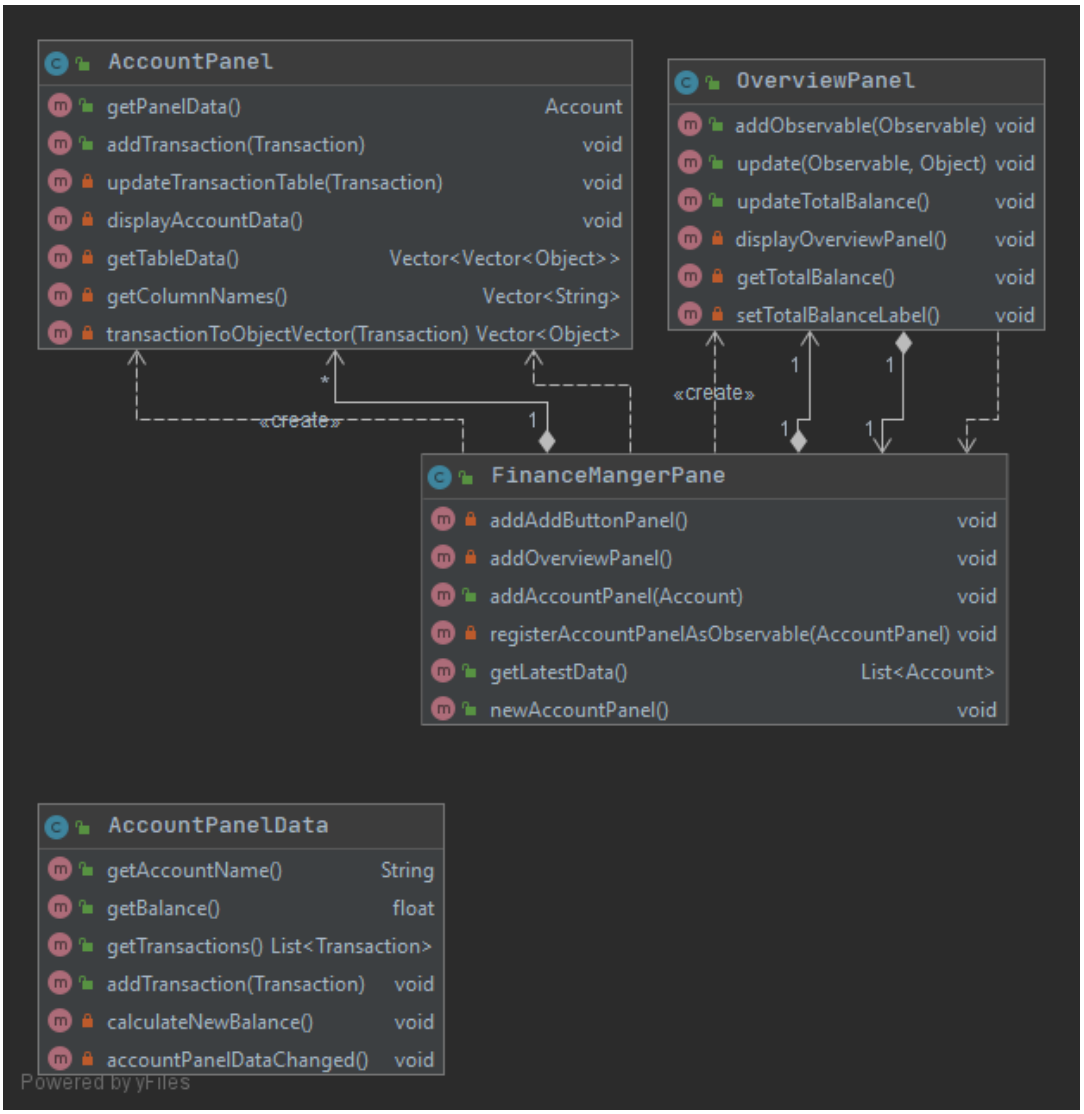

Abbildung 3.1.: Die genutzten Klassen bevor das Observer Entwurfsmuster eingeführt wurde

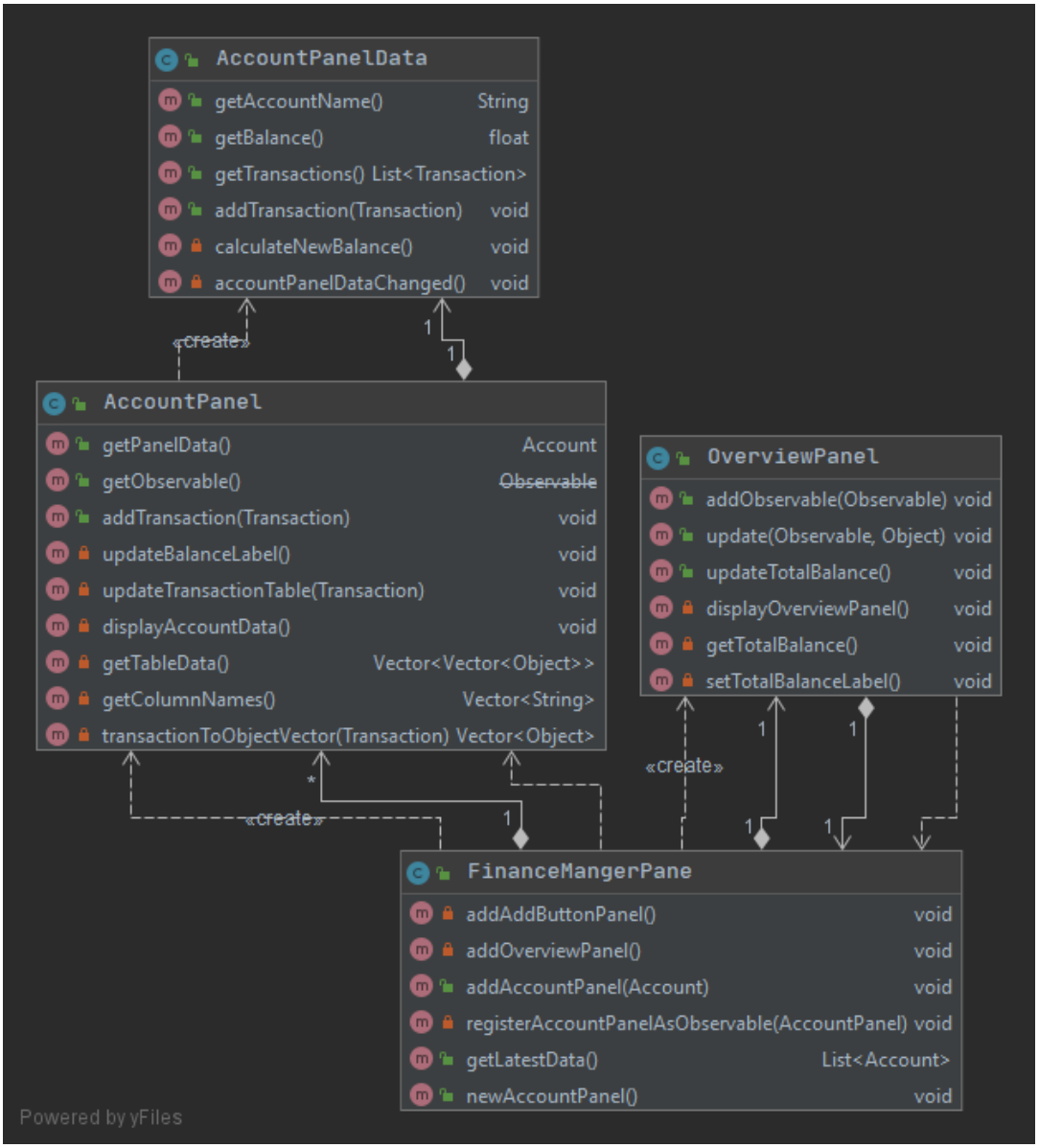

Abbildung 3.2.: Die Klassen, welche das Observer Entwurfsmuster nutzen

4. Domain Driven Design

Domain Driven Design beschreibt eine Modellierungsmethode für komplexe Software. Hierbei soll die Lücke zwischen der fachlichen Expertise und der Umsetzung möglichst klein gehalten werden. Eine Möglichkeit dies zu erreichen ist die Nutzung von Fachbegriffen der Domäne als Klassen- und Funktionsnamen. Im Programmentwurf wurde dies so umgesetzt. Da es sich bei der Anwendung um eine Software zur Verwaltung von Finanztransaktionen mit mehreren Accounts handelt, wurden eben diese Begriffe genutzt. Als prominente Beispiel können die `AccountPanel`- und die `FinanceManagerContext`-Klassen genannt werden. Diese beiden Klassen sind gute Beispiele dafür, wie die Modellierung abgelaufen ist und das die Klassen danach benannt wurden, welche Aufgabe beziehungsweise Funktionalität sie innerhalb der Anwendung abdecken. Als Architektur wurde die in Kapitel 5 näher erklärte Schichtenarchitektur genutzt.

5. Architektur

Der *FinanceManager* implementiert eine zwei Schichten-Architektur. Abbildung 5.1 zeigt die beiden Schichten *View* und *Data*. Diese Architektur wurde gewählt, da es in der Anwendung hauptsächlich um das Halten (*Data*) und das Anzeigen (*View*) von Daten, beziehungsweise Konten, geht. Die *View*-Schicht spricht als obere der beiden Schichten die *Data*-Schicht an.

In der Implementierung wurde durch das Aufteilen der Klassen in Packages zwischen

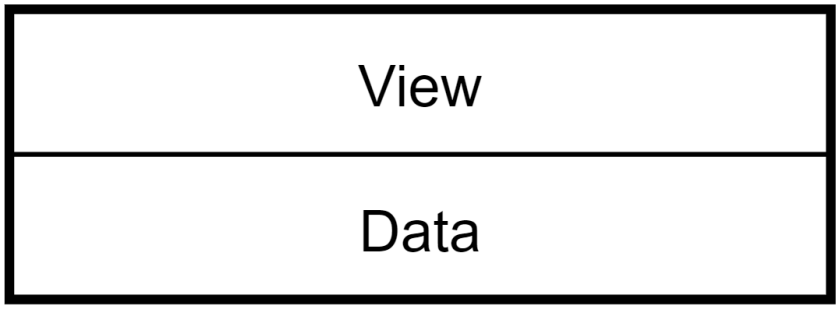

Abbildung 5.1.: Schichtenmodell des FinanceManager

den beiden Schichten unterschieden. Diese Aufteilung zeigt das UML-Diagramm in der Abbildung 5.2. Hier ist auch zu sehen, dass der `FinanceManagerController` eine Instanz der *Data*-Schicht besitzt. Diese wird gebraucht, um die Nutzereingabe beim Start der Anwendung zu verarbeiten. Die Abbildung 5.2 zeigt auch, dass nur die *View*-Schicht die *Data*-Schicht aufruft und nicht umgekehrt.

Die *View*-Schicht beinhaltet alle Klassen, welche genutzt werden, um mit dem Nutzer zu interagieren und ihm die Daten anzuzeigen. Hierzu zählt sowohl das Hauptfenster der Applikation als auch die Dialogfenster, welche genutzt werden, um Benutzereingaben abzufragen.

Auf der *Data*-Schicht ist die Repräsentation der Daten als Datenmodell implementiert. Zudem beinhaltet diese Schicht Klassen, mithilfe welcher die Benutzereingaben validiert werden. Auch Klassen, welche das Ver- und Entschlüsseln und damit verbunden auch das Lesen und Schreiben zum lokalen Dateisystem implementieren, sind in dieser Schicht enthalten.

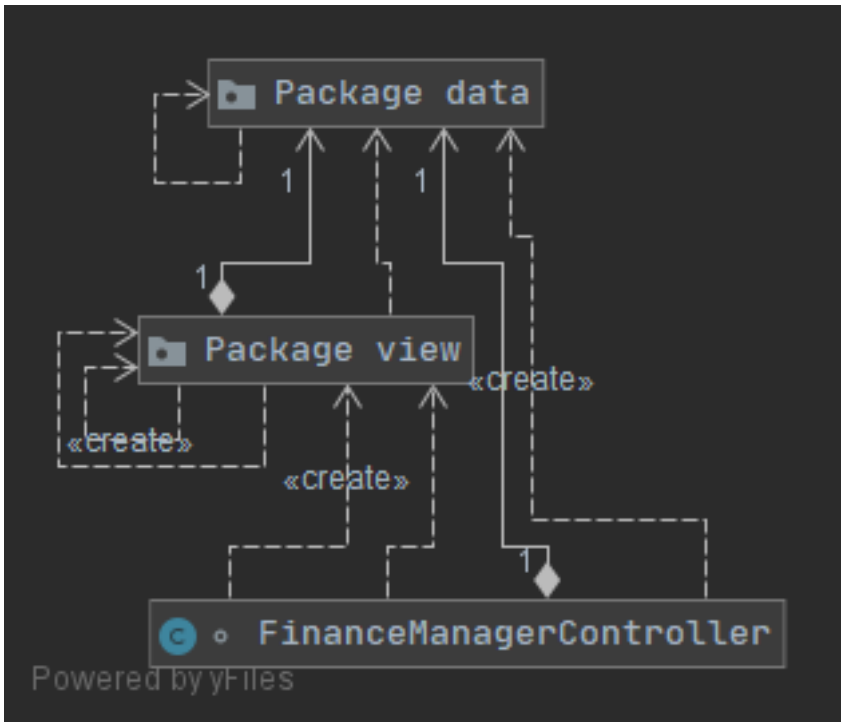

Abbildung 5.2.: Schichtenmodell als UML-Diagramm

6. Unit-Tests

Unit-Tests sind eine Art des Testens einer Software. Hierbei wird eine mögliche kleine Einheit der Software getestet. Unit-Tests sollen dir Automatic, Thorough, Repeatable, Independent, Professional (ATRIP)-Regeln erfüllen.

Automatic Tests soll mit höchstens einem Knopfdruck oder Befehl gestartet werden.

Thorough Test muss alles notwendige überprüfen. Hierbei ist problematisch, dass das Notwendige subjektives Empfinden des Entwicklers ist.

Repeatable Test muss beliebig oft wiederholbar sein und immer das gleiche Ergebnis liefern. Des weiteren muss er von seiner Umgebung unabhängig sein.

Independent Test darf keine Abhängigkeit zu anderen Tests besitzen. Setup und Teardown Funktionalitäten vereinfachen beispielsweise eine Initialisierung.

Professional Tests sollen den selben Qualitätsansprüchen unterliegen wie Produktivcode. Tests sollten nur geschrieben werden wenn sie notwendig sind.

Die Unit-Tests des Programmentwurf erfüllen die ATRIP-Regeln. Sie werden sowohl vor jeder Build-Phase ausgeführt, als auch bei jedem Push in das Repository. Die Tests prüfen vollständig und sind beliebig oft wiederholbar, da keine Techniken wie Zufall genutzt werden. Die einzelnen Tests sind voneinander unabhängig. Die Codequalität der Tests entspricht dem des Produktivcodes.

Current scope: all classes

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	31,2% (10/32)	14% (20/143)	7,7% (49/639)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
dev.hottek	0% (0/2)	0% (0/8)	0% (0/30)
dev.hottek.data	71,4% (5/7)	25,7% (9/35)	17,1% (20/117)
dev.hottek.data.encryption	0% (0/2)	0% (0/6)	0% (0/24)
dev.hottek.data.exception	50% (1/2)	50% (1/2)	50% (1/2)
dev.hottek.data.model	80% (4/5)	35,7% (10/28)	46,7% (28/60)
dev.hottek.view	0% (0/2)	0% (0/10)	0% (0/67)
dev.hottek.view.detail	0% (0/5)	0% (0/36)	0% (0/183)
dev.hottek.view.dialog	0% (0/4)	0% (0/8)	0% (0/88)
dev.hottek.view.listener	0% (0/3)	0% (0/10)	0% (0/68)

Abbildung 6.1.: Code Coverage Unit-Tests

7. Refactoring

Refactoring ist eine Änderung der internen Struktur von Code, um diesen besser verständlich, leichter veränderbar und seine Qualität zu verbessern. Refactoring ist ein Schritt im alltäglichen Entwicklungsprozess. Durch das regelmäßige Anwenden von Refactorings wird langfristig die Qualität des Codes und somit auch die Entwicklungsgeschwindigkeit erhöht. In besser lesbarem Code können Fehler leichter gefunden und Änderungen mit geringerem Aufwand umgesetzt werden. Refactoring behandelt das Auftreten von Code Smells.

Im Programmentwurf sind eine Reihe von Refactorings durchgeführt worden (siehe git history). Im Folgenden werden zwei Code Smells beschrieben und die an ihnen angewandten Refactorings:

Duplicated Code Dieser Code Smell trat in der Klasse `OpeningWindowListener` auf. Hierbei liegt doppelt vorhandener Code vor. Um diesen zu beheben wird eine neue Struktur erschaffen, welche den duplizierten Code ersetzt.

```
1            initialHistory.add(instanceCreated);
2            initialHistory.add(accountCreated);
3            FMcontext.setHistoryEntries(initialHistory);
4            FMcontext.setInitialAccountList(initialAccounts);
5            FMcontext.setWait(false);
6    } else if ("open".equals(actionCommand)) {
7        JFileChooser fileChooser = setupFileChooser();
8        int returnValue = fileChooser.showOpenDialog(null);
9        if (returnValue == JFileChooser.APPROVE_OPTION) {
10           String filePath = fileChooser.getSelectedFile().getPath();
11           String filePathWithoutFileType = filePath.split("\\.")[0];
12           String ivFilePath = filePathWithoutFileType + "_iv.fm";
13           JsonReader jsonReader = new JsonReader();
14           if (checkForPasswordProtection()) {
```

```
15                SafeFormat data = getAccountsFromEncryptedFile(filePath, ↙
                      ↳ ivFilePath, jsonReader);
16                FMcontext.setInitialAccountList(data.getAccountList());
17                FMcontext.setHistoryEntries(data.getHistoryEntryList());
18                FMcontext.setWait(false);
19                return;
20            }
21            SafeFormat data = jsonReader.readJsonFromFile(filePath);
22            FMcontext.setInitialAccountList(data.getAccountList());
23            FMcontext.setHistoryEntries(data.getHistoryEntryList());
24            FMcontext.setWait(false);
25        }
26    }
```

Listing 7.1: Duplicated Code - vor Refactoring

Listing 7.1 zeigt den entsprechenden Codeausschnitt. Die Zeilen 3-5, 16-18 und 22-24 enthalten den gleichen Code. Auf diese Zeilen wurde das Refactoring Extract Method angewandt. Listing 7.2 zeigt die gewonnene Methode. Diese wurde anstelle den Zeilen in Listing 7.1 aufgerufen.

```
1  private void notifyFMcontext(List<Account> accountList, ↙
       ↳ List<HistoryEntry> historyEntries) {
2      FMcontext.setHistoryEntries(historyEntries);
3      FMcontext.setInitialAccountList(accountList);
4      FMcontext.setWait(false);
5  }
```

Listing 7.2: Duplicated Code - nach Refactoring

Long Method Dieser Code Smell trat auch in der Klasse `OpeningWindowListener` auf. Bei diesem Code Smell liegt eine große Methode oder Funktion vor, welche unübersichtlich ist. Oft ist hierbei der Funktionsname nicht mehr beschreiben für das Verhalten der Funktion. Um diesen Code Smell zu beheben wird die Funktion in kleinere Funktionen aufgeteilt, welche die Lesbarkeit erhöhen sollen. Im gleichen Schritt wurde in dieser Funktion die Nutzung von Switch Statements aufgelöst. Listing 7.3 zeigt den entsprechenden Codeabschnitt vor dem Refactoring.

```
1  JFileChooser fileChooser = new JFileChooser();
2  fileChooser.setDialogTitle("Select a Finance Manager file");
```

```
3  fileChooser.setAcceptAllFileFilterUsed(false);
4  FileNameExtensionFilter filter = new ↙
       ↳ FileNameExtensionFilter("Finance Manager file", "fm");
5  fileChooser.addChoosableFileFilter(filter);
6  int returnValue = fileChooser.showOpenDialog(null);
7  switch (returnValue) { //TODO: Handle other returnValues
8      case JFileChooser.APPROVE_OPTION:
9          String filePath = fileChooser.getSelectedFile().getPath();
10         String filePathWithoutFileType = filePath.split("\\.")[0];
11         String ivFilePath = filePathWithoutFileType + "_iv.fm";
12         JsonReader jsonReader = new JsonReader();
13         boolean isPasswordProtected = checkForPasswordProtection();
14         if (isPasswordProtected) {
15             String password = retrievePassword();
16             DataHandler dataHandler = new DataHandler(password, ↙
                   ↳ filePath, ivFilePath);
17             String data = dataHandler.loadData();
18             List<Account> accounts = ↙
                   ↳ jsonReader.readJsonFromString(data);
19             FMcontext.setAccountList(accounts);
20             FMcontext.setWait(false);
21             break;
22         }
23         List<Account> accounts = jsonReader.readJsonFromFile(filePath);
24         FMcontext.setAccountList(accounts);
25         FMcontext.setWait(false);
26         break;
27     default:
28         break;
29 }
```

Listing 7.3: Long Method - vor Refactoring

Listing 7.4 zeigt zum einen die durch das Refactoring gewonnenen Methoden, als auch die Auflösung des Switch Statements.

```
1  JFileChooser fileChooser = setupFileChooser();
2  int returnValue = fileChooser.showOpenDialog(null);
3  if (returnValue == JFileChooser.APPROVE_OPTION) {
4      String filePath = fileChooser.getSelectedFile().getPath();
5      String filePathWithoutFileType = filePath.split("\\.")[0];
6      String ivFilePath = filePathWithoutFileType + "_iv.fm";
```

```
7       JsonReader jsonReader = new JsonReader();
8       if (checkForPasswordProtection()) {
9           List<Account> accounts = ↙
                ↳ getAccountsFromEncryptedFile(filePath, ivFilePath, ↙
                ↳ jsonReader);
10          FMcontext.setAccountList(accounts);
11          FMcontext.setWait(false);
12          break;
13      }
14      List<Account> accounts = jsonReader.readJsonFromFile(filePath);
15      FMcontext.setAccountList(accounts);
16      FMcontext.setWait(false);
17  }
18
19  ...
20
21  private JFileChooser setupFileChooser() {
22      JFileChooser fileChooser = new JFileChooser();
23      fileChooser.setDialogTitle("Select a Finance Manager file");
24      fileChooser.setAcceptAllFileFilterUsed(false);
25      FileNameExtensionFilter filter = new ↙
            ↳ FileNameExtensionFilter("Finance Manager file", "fm");
26      fileChooser.addChoosableFileFilter(filter);
27      return fileChooser;
28  }
29
30  private List<Account> getAccountsFromEncryptedFile(String filePath, ↙
        ↳ String ivFilePath, JsonReader jsonReader) {
31      String password = retrievePassword();
32      DataHandler dataHandler = new DataHandler(password, filePath, ↙
            ↳ ivFilePath);
33      String data = dataHandler.loadData();
34      return jsonReader.readJsonFromString(data);
35  }
```

Listing 7.4: Long Method - nach Refactoring

A. Anhang

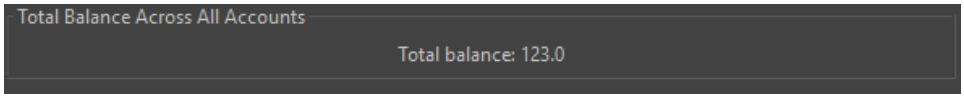

Abbildung A.1.: Der Übersichtstab (Overviewtab) zeigt die über alle Accounts aufsummierte Bilanz an

Abbildung A.2.: Der Tab für einen Account

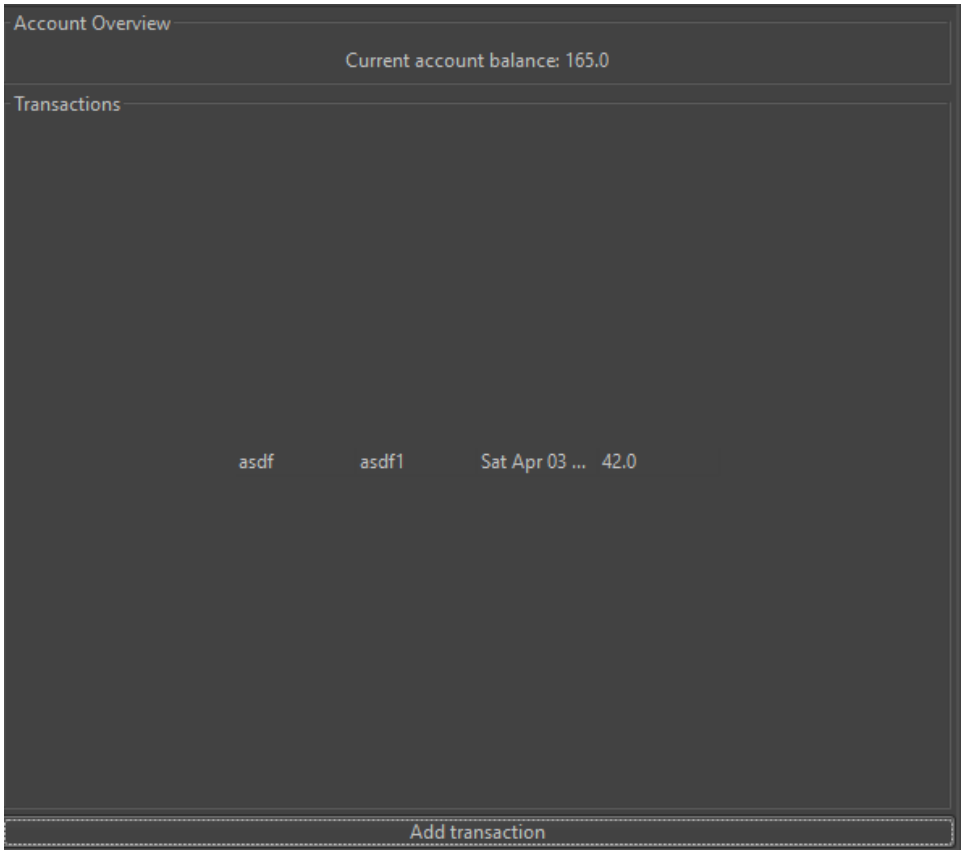

Abbildung A.3.: Der Tab für einen Account mit einer Transaktion

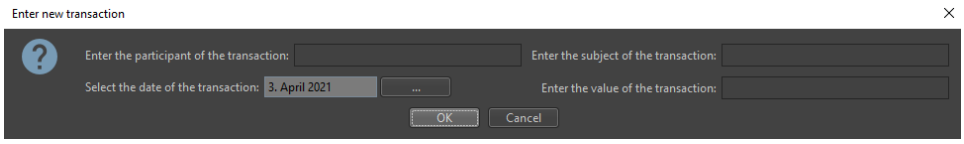

Abbildung A.4.: Dialogfenster zum Hinzufügen einer Transaktion zu einem Account

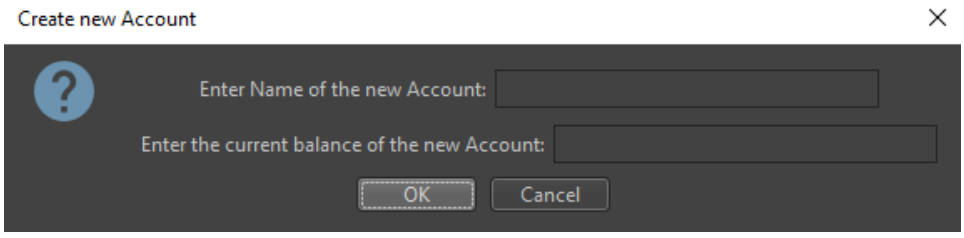

Abbildung A.5.: Dialogfenster zum Hinzufügen eines neuen Accounts