

Software-Engineering II

Programmmentwurf

TINF20B1

5.+6. Semester (2022/2023)

Thema:

Kostenrechner für Fahrgemeinschaften

Dozent:

Daniel Lindner

Bearbeitender:

Yannik Schiebelhut

Inhaltsverzeichnis

Abkürzungsverzeichnis	II
Abbildungsverzeichnis	III
1 Beschreibung des Programms	1
1.1 Funktionalität	1
1.2 Technologien	2
2 Domain Driven Design	3
2.1 Analyse der Ubiquitous Language	3
2.2 Entities	5
2.3 Value Objects	5
2.4 Aggregates	6
2.5 Repositories	6
3 Clean Architecture	7
3.1 Unterteilung in Schichten	7
3.2 Dependency Inversion	10
4 Entwurfsmuster	11
4.1 Beobachter	11
5 Unit Tests	14
5.1 ATRIP-Regeln	15
5.2 Code Coverage	17
5.3 Einsatz von Mocks	18
6 Refactoring	19
6.1 Code Smells identifizieren	19
6.2 Refactorings anwenden	20

Abkürzungsverzeichnis

Abbildungsverzeichnis

2.1	Domänenmodell	4
4.1	Abstraktes UML-Diagramm des Beobachter-Entwurfsmusters	12
4.2	Konkrete Anwendung des Beobachter-Entwurfsmusters	13
5.1	Übersicht über die erstellen Unit Tests	14
5.2	Coverage-Analyse des Projekts	18

1 Beschreibung des Programms

Im Rahmen eines dualen Studiums an der DHBW bietet es sich an, für die Praxisphasen mit anderen Studenten Fahrgemeinschaften zum Betrieb zu bilden. Diese Fahrgemeinschaften haben zwar einen festen Rahmen an Mitgliedern, jedoch haben diese alle verschiedene und wechselnde Termine, wodurch sich der Anwendungsfall für ein Programm ergeben hat, welches den Fahrer einer Fahrgemeinschaft dabei unterstützt, mit wenig Aufwand eine für alle Beteiligten faire Abrechnung zu erstellen.

1.1 Funktionalität

Das Programm richtet sich an den Besitzer eines Autos, weiterhin als Fahrer bezeichnet. Der Fahrer kann mehrere „Fahrgemeinschaften“, also Gruppen von Personen, definieren. Dabei kann eine Person in mehreren Fahrgemeinschaften sein und eine Fahrgemeinschaft hat in der Regel mehrere Mitfahrer.

Innerhalb einer Fahrgemeinschaft werden Fahrperioden angelegt. Eine Fahrperiode bezeichnet dabei die Menge aller Fahrten zwischen zwei Tankstopps. Innerhalb einer Fahrperiode haben alle Fahrten dieselbe Strecke, denselben Spritverbrauch und denselben Spritpreis. Des Weiteren kann ein Fixbetrag definiert werden, um etwa Verschleißkosten des Fahrzeugs auf die Mitfahrer umzulegen. In einer Fahrperiode wiederum kann eine beliebige Menge an Fahrten angelegt werden. Für jede dieser Fahrten wird ausgewählt, welche Mitglieder der Fahrgemeinschaft im Fahrzeug saßen. Wenn wieder getankt wird, wird die Fahrperiode im System abgeschlossen. Dabei wird der Anteil an den entstandenen Fahrkosten für jeden Mitfahrer innerhalb der Fahrperiode ermittelt. Für diesen Anteil wird ein PayPal-Link generiert und der jeweiligen Person über Telegram zugesandt, um auch den Bezahlvorgang angenehm zu gestalten.

Eine Person wiederum hat einen Namen, eine Adresse und eine Telegram-Chat-ID, über die diese Person zu kontaktieren ist.

1.2 Technologien

Umgesetzt ist das Programm in Java, wobei Maven als Build-System verwendet wird. Die Datenhaltung erfolgt lokal im JSON-Format mittels der externen GSON-Bibliothek. Das Programm verfügt über eine grafische Nutzeroberfläche, welche mit Java Swing erstellt ist. Weiterhin wird auf die Telegram API zugegriffen. Hierfür ist allerdings im Rahmen einer einfachen Proof of Concept Implementierung keine externe Bibliothek vonnöten.

Der Quellcode wird in einem Git-Repository verwaltet, welches auf GitHub unter <https://github.com/yschiebelhut/carpool-java> zu finden ist.

2 Domain Driven Design

2.1 Analyse der Ubiquitous Language

Das Programm wird im Rahmen einer Fahrgemeinschaft einer deutschsprachigen Gruppe von Studenten erstellt, weshalb Deutsch als Sprache für die Ubiquitous Language gewählt wird.

Die Beschreibung in Abschnitt 1.1 entspricht dem Sprachgebrauch eines Domänenexperten, weshalb sich diese Wortwahl auch im Quellcode widerspiegeln sollte. Die wichtigsten Begriffe sind hierbei:

- Fahrgemeinschaft
- Fahrperiode
- Distanz (bestehend aus Betrag und Streckeneinheit)
- Fahrt
- Person (je nach Kontext auch als Mitfahrer bezeichnet, diese Begriffe sind austauschbar)
- Adresse (bestehend aus Straße und Ort)
- Telegram-Chat-ID
- PayPal-Link

Weiterhin wird festgelegt, dass für Informatikstudenten einzelne englische Begriffe keine Sprachbarriere darstellen. Deshalb werden Getter- und Setter-Methoden weiterhin mit englischen Vorsilben konstruiert, um den Quellcode nah an Programmiersprachweiten Standards zu halten.

Aus der Funktionsbeschreibung des Programms und der Analyse der Ubiquitous Language ergibt sich das in Abbildung 2.1 dargestellte Domänenmodell.

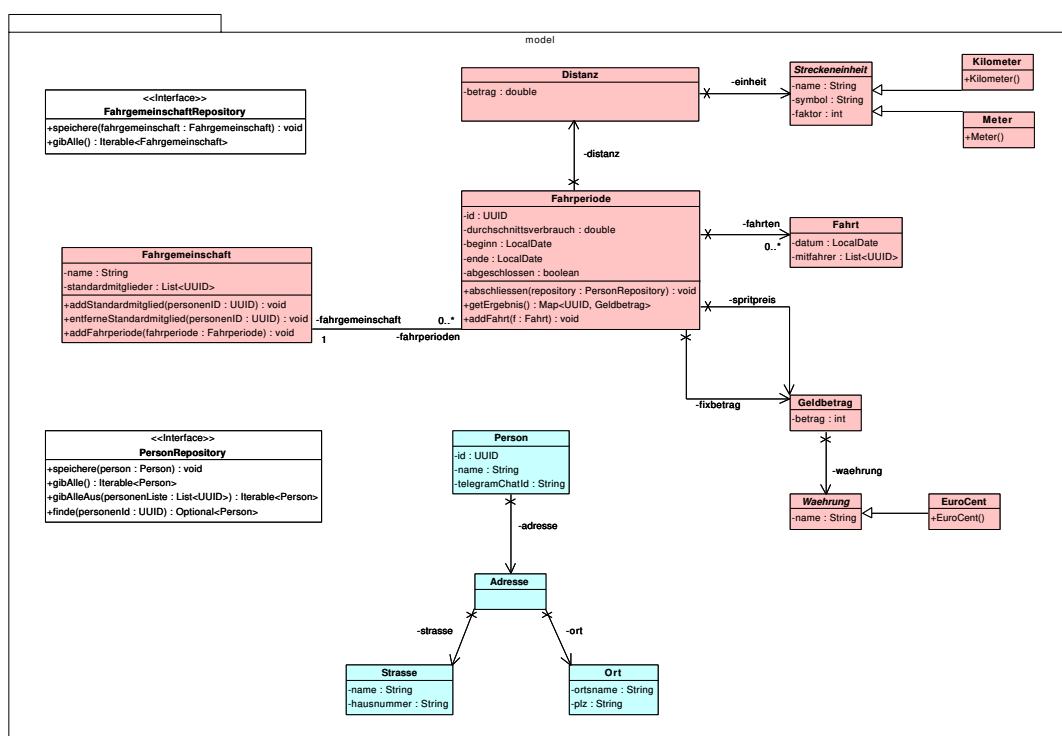


Abbildung 2.1: Domänenmodell

2.2 Entities

Im dargestellten Domänenmodell (Abbildung 2.1) finden sich folgende Entitäten:

- Person
- Fahrgemeinschaft
- Fahrt

Eine Person besitzt die Eigenschaften Name, Adresse und Telegram-Chat-ID, welche sich alle ändern können, ohne dass sich die Identität der Person ändert. Einer Fahrgemeinschaft werden im Laufe der Zeit Fahrperioden und möglicherweise Mitglieder hinzugefügt, es ist jedoch weiterhin dieselbe Fahrgemeinschaft. Ähnlich sieht es bei einer Fahrperiode aus, der nach und nach Fahrten hinzugefügt werden. Hieran lässt sich ablesen, dass alle diese Klassen einen Lebenszyklus besitzen und somit als Entitäten einzuordnen sind.

Jede Entität verfügt über eine Identität. Im Falle der Fahrgemeinschaft wird hierfür der Name der Fahrgemeinschaft als natürlicher Identifier gewählt, da ein Fahrer realistischweise kaum mehr als eine Handvoll verschiedene Gruppen mitnimmt. Für Person und Fahrperiode reichen die Attribute nicht aus, um eine Entität eindeutig zu identifizieren. Deshalb wird hier eine UUID als künstlicher Schlüssel vergeben.

2.3 Value Objects

Die restlichen, in Abbildung 2.1 dargestellten, Klassen werden als Value Objects realisiert. Dies sind:

- Distanz
- Unterklassen von Streckeneinheit
- Fahrt
- Geldbetrag
- Unterklassen von Währung
- Adresse

- Strasse
- Ort

Alle diese Klassen besitzen keinen Lebenszyklus und definieren sich rein über ihre Werte, weshalb sich auch keine Identität besitzen und sich somit nicht als Entität einordnen lassen.

2.4 Aggregates

Eine Adresse stellt eine Eigenschaft einer Person dar. Weiterhin setzt sich eine Adresse aus einer Strasse und einem Ort zusammen. Adresse, Strasse und Ort existieren nur im Kontext einer Person. Wird die Person gelöscht, so existiert auch die zugehörige Adresse nicht mehr. Deshalb werden Person, Adresse, Strasse und Ort zu einem Aggregat zusammengefasst. Person ist in diesem Aggregat die einzige Entität und ist deshalb automatisch die Root-Entität. Im Domänenmodell wird dieses Aggregat türkis hervorgehoben.

Im Falle von Fahrperioden stellen Distanz, Streckeneinheit, Fahrt, Geldbetrag und Waehrung Value Objects dar, die Eigenschaften von Fahrperiode zu klassifizieren sind. Deshalb werden diese Klassen zu einem Aggregat zusammengefasst. Weiterhin existieren Fahrperioden jedoch nur im Kontext ihrer Fahrgemeinschaft. Wird die Fahrgemeinschaft gelöscht, so entfernt dies auch die Fahrperioden. Deshalb wird diesem Aggregat auch noch die Fahrgemeinschaft hinzugefügt und diese außerdem als Root-Entität gewählt. Im Domänenmodell wird dieses Aggregat rot hervorgehoben.

2.5 Repositories

Gemäß dem Grundsatz „Ein Repository pro Aggregat“ werden zwei Repositories für den Zugriff auf den persistenten Speicher definiert. Dies sind PersonRepository und FahrgemeinschaftRepository. Diese erlauben jeweils den Zugriff auf die jeweilige Root-Entität des Repositories - also Person respektive Fahrgemeinschaft.

3 Clean Architecture

In diesem Kapitel wird die Software so in verschiedene Teilmodule restrukturiert, dass sie den Grundsätzen der in der Vorlesung vermittelten Clean Architecture genügt.

3.1 Unterteilung in Schichten

3.1.1 Schicht 4: Abstraction Code

Abstraction Code stellt domänenübergreifendes Wissen dar (wie etwa mathematische Konzepte, grundlegende Algorithmen oder Datenstrukturen). Im Kontext der vorliegenden Software wird diese Schicht nicht benötigt. Mathematische Konzepte, die zur Berechnung der Fahrtkosten notwendig sind, sowie alle in der Software verwendeten Datenstrukturen, finden sich bereits in den Java-Bibliotheken.

3.1.2 Schicht 3: Domain

In dieser Schicht wird der zentrale Domain-Code untergebracht. In diesem Falle sind dies die in Abbildung 2.1 gezeigten und im vorherigen Kapitel beschriebenen Entitäten und Value Objects (implementiert im Modul `3-carpool-java-domain`). Sie gehören zur elementaren Business Logik der Domäne und sollten sich möglichst selten, in der Regel eigentlich nie ändern. Lediglich das in der Abbildung dargestellte *Integration Package* wird in die Schicht 0 verschoben. Zwar gehört gemäß Beschreibung des Domainen-Experten das Senden von PayPal-Links via Telegram als elementarer Bestandteil zum Programm, jedoch sollte die konkrete Implementierung der Interaktion mit diesen Diensten nicht als Teil des Domainen-Codes erfolgen, da sich diese logisch abgrenzt und Änderungen im Laufe der Zeit sehr wahrscheinlich sind.

3.1.3 Schicht 2: Application

Die Application-Schicht ist gemäß der Vorlesung für Aggregat-übergreifendes Verhalten verantwortlich. In Falle der vorliegenden Software liegt dieser Anwendungsfall beim Versenden der PayPal-Links über Telegram vor. Hierbei werden die Rechnungsdaten einer Fahrperiode ausgelesen. Diese enthalten jedoch nur ein Mapping einer UUID zu einem Geldbetrag. Zum Versenden der Nachrichten müssen diese UUIDs noch zu Personen aufgelöst werden, aus welchen anschließend die Telegram-Chat-ID extrahiert werden kann. Umgesetzt ist diese Schicht im Modul `2-carpool-java-application`.

3.1.4 Schicht 1: Adapters

Die Adapters-Schicht soll in der Clean Architecture dazu dienen, Aufrufe und Daten der Plugin-Schicht an die innere Schicht zu vermitteln. Hier finden zum Beispiel Aufgaben wie Formatkonvertierungen und das Erstellen von Render-Modellen statt. Da jedoch in der Plugin-Schicht dieser Anwendung keine komplexe Umsetzungslogik erfolgt und ein Erstellen dieser Schicht keinen unmittelbaren Wert besitzt (wie auf Folie 49 der Clean Architecture beschrieben), wird auf das Erstellen dieser Schicht verzichtet. Es ist allerdings darauf hinzuweisen, dass ein Erstellen dieser Schicht theoretisch empfehlenswert wäre, vor allem in der Hinsicht, dass die aktuell mit Java-Swing erstellte Benutzeroberfläche nicht gerade schön ist und gegebenenfalls zukünftig durch eine neuere und bessere Oberfläche ersetzt werden könnte. Eine solche Oberfläche kann ohne die Adapters-Schicht nicht auf visuelle Verarbeitungsroutinen der aktuellen GUI zugreifen und müssten neu angefertigt werden, was wiederum mit einem Mehraufwand verknüpft wäre.

3.1.5 Schicht 0: Plugins

Die Plugin-Schicht stellt die äußerste Schicht der Anwendung dar. Code in dieser Schicht ist am kurzlebigsten. Hier werden für die aktuelle Anwendung 4 Module erstellt: `0-carpool-java-plugins-main`, `0-carpool-java-plugins-ui`, `0-carpool-java-plugins-json` und `0-carpool-java-plugins-integration`.

Main-Plugin

Dieses Plugin enthält die Main-Methode und somit den Einstiegspunkt in die Applikation. Hier ist keinerlei Anwendungslogik enthalten, sondern es erfolgt lediglich eine Koordination der anderen Plugins. Die Main-Methode delegiert die Datenverwaltung an das JSON-Plugin und initialisiert die vom GUI-Plugin implementierte Benutzeroberfläche. Weiterhin wird ein Runtime-Hook definiert, der dafür sorgt, dass beim Beenden der Anwendung die aktuellen Daten gespeichert werden. Das Speichern wird somit vom Lebenszyklus der GUI entkoppelt.

GUI-Plugin

Hier findet die Implementierung der GUI statt. Da die Benutzeroberfläche Pure Fabrication darstellt und häufige Änderungen und Wechsel zu erwarten sind, wird diese in der Plugin-Schicht implementiert.

JSON-Plugin

Die Software setzt aktuell auf eine JSON-basierte Datenhaltung. Zu diesem Zwecke wird die externe GSON-Library verwendet. Weil eine externe Library eingebunden wird und die Form der Datenhaltung sehr einfach veränderlich sein sollte, wird die Datenhaltung als Plugin implementiert. Dabei werden zwei Repository-Klassen erstellt, die die in der Domain-Schicht definierten Interfaces implementieren. Für das Speichern gewisser Datentypen wie beispielsweise `LocalDate` aus der Java-Standardlibrary werden von GSON Typadapter benötigt. Theoretisch wären diese der Adapters-Schicht zuzuordnen. Allerdings sind die enthaltenen Mappings speziell auf die Verwendung mit GSON zuzuschneiden. Weiterhin müssen die Adapter Interfaces implementieren, die von der Library definiert werden. Aus diesen Gründen ist es nicht sinnvoll, die Typadapter in die Adapter-Schicht auszulagern.

Integration-Plugin

In diesem Plugin werden die PayPal-Link-Generierung sowie die Interaktion mit Telegram implementiert. Beides bezieht sich auf Schnittstellen von externen Systemen, von denen

perse schon ein häufiger Wandel zu erwarten ist, weshalb diese Klassen auf jeden Fall in der Pluin-Schicht zu implementieren sind.

3.2 Dependency Inversion

Als zentrale Regel der Clean Architecture (Dependency Rule) sollten Abhängigkeiten nur von außen nach innen zeigen. Bis auf eine Ausnahme ist diese Regel nach der Umstrukturierung bereits erfüllt.

Die Klasse `FahrperiodenAbschliessService` aus `2-carpool-java-application` hat jedoch eine Abhängigkeit auf die Klassen `Telegram` und `PayPalLinkBuilder` aus der Schicht `0-carpool-java-plugins-integration`. Um diese Abhängigkeit zu beseitigen wird eine Dependency Inversion durchgeführt.

Hierfür werden zunächst in `2-carpool-java-application` zwei Interfaces (`TelegramClient` und `IPayPalLinkBuilder`) definiert. Diese Interfaces deklarieren die Methoden der beiden Klassen aus Schicht 0. Anschließend wird eine Abhängigkeit definiert, die vom Plugin zur Applikationsschicht zeigt. Anschließend wird die Implementierung der beiden betroffenen Klassen in Schicht 0 so verändert, dass das jeweils korrespondierte Interface implementiert wird. Weiterhin muss der `FahrperiodenAbschliessService` so modifiziert werden, dass die Abhängigkeiten per Dependency Injection übergeben werden. Der Konstruktoraufruf ist entsprechend anzupassen.

(Es ist darauf hinzuweisen, dass dadurch im vorliegenden Fall eine neue Abhängigkeit des GUI-Plugins auf das Integrations-Plugin entsteht. Theoretisch wäre es optimal, auch diese zu vermeiden. Da diese neue Abhängigkeit jedoch die Dependency Rule nicht verletzt und der Aufwand zur Vermeidung unverhältnismäßig hoch wäre, wird auf eine Umsetzung verzichtet.)

4 Entwurfsmuster

Entwurfsmuster dienen in der Softwareentwicklung als Lösungsansätze für wiederkehrende Probleme. Es handelt sich dabei nicht um fertigen Code, sondern vielmehr um konzeptionelle Bausteine, die zum Beispiel die Kommunikation unter Entwicklern unterstützen können. Auch in diesem Programmentwurf finden sich Entwurfsmuster wieder. Im Folgenden wird eins dieser Entwurfsmuster genauer erläutert.

4.1 Beobachter

Der Beobachter gehört zur Klasse der Verhaltensmuster. Kern des Konzeptes ist es, Änderungen an einem Ausgangsobjekt einer Anzahl anderer Objekte mitzuteilen. Im Gegensatz zum Polling (ein Objekt fragt periodisch ab, ob ein anderes sich geändert hat) findet hier Kommunikation nur im Änderungsfall statt.

Kernelement des Entwurfsmusters sind Subjekte, die von den Beobachtern observiert werden. Ein Subjekt hat dabei die Möglichkeit, Beobachter an- und abzumelden, sowie diese zu benachrichtigen, wenn eine Veränderung vorliegt. Bei dieser Benachrichtigung wird über alle registrierten Beobachter iteriert und deren Methode zur Aktualisierung aufgerufen. Je nach Implementierung des Entwurfsmusters kann hier auch eine Payload mitgegeben werden. Eine Darstellung des allgemeinen Schemas ist in Abbildung 4.1 zu sehen.

4.1.1 Umsetzung im Programmentwurf

In der grafischen Benutzeroberfläche des Fahrtkostenrechners spielen Knöpfe (JButtons) eine wichtige Rolle. Knöpfe werden dabei ohne eigene Funktionalität instanziiert. Stattdessen können für einen Knopf sogenannte *ActionListener* registriert werden. Dieser Vorgang (in Abbildung 4.2 dargestellt) entspricht dem Beobachter-Entwurfsmuster. Es wurde farblich hervorgehoben, welche Klassen der Umsetzung welcher Rolle des allgemeinen Schemas entsprechen. `AbstractButton` entspricht dem abstrakten Subjekt,

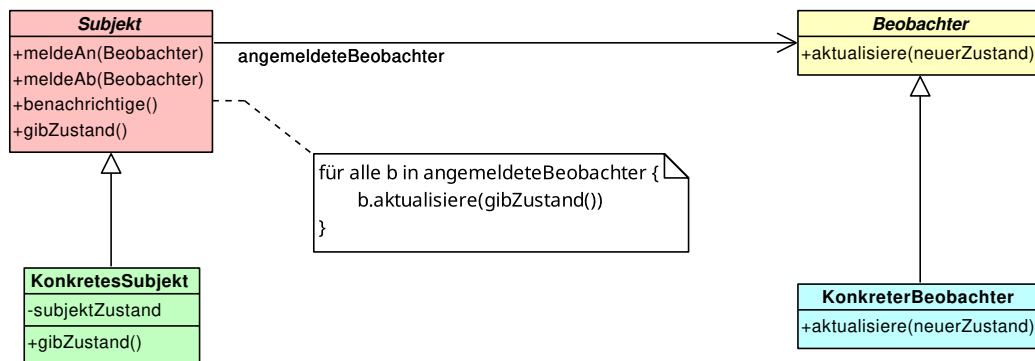


Abbildung 4.1: Abstraktes UML-Diagramm des Beobachter-Entwurfsmusters

`ActionListener` dem abstrakten `Beobachter`, wobei hier für den abstrakten Beobachter ein Interface herangezogen wird anstelle einer Klasse. Dieses Interface wird dann von der Klasse `FahrperiodenAbschliessService` implementiert. `JButton` stellt die Implementierung des konkreten Subjekts (`KonkretesSubjekt`) dar.

Anmerkung Abbildung 4.2 ist eine vereinfachte Darstellung der Implementierung des Java-Standards. Insbesondere bei der Darstellung der `listenerList` wurde die Komplexität auf das Wesentliche heruntergebrochen, da die eigentliche Detailtiefe der Implementierung für die reine Demonstration des Entwurfsmusters hinderlich ist.

Die Verwendung des Entwurfsmusters ist [hier \(Instanziierung des JButtons\)](#) und [hier \(Implementierung des ActionListener\)](#) zu finden. Die Implementierung der Subjekte ist Teil des Java-Standards.

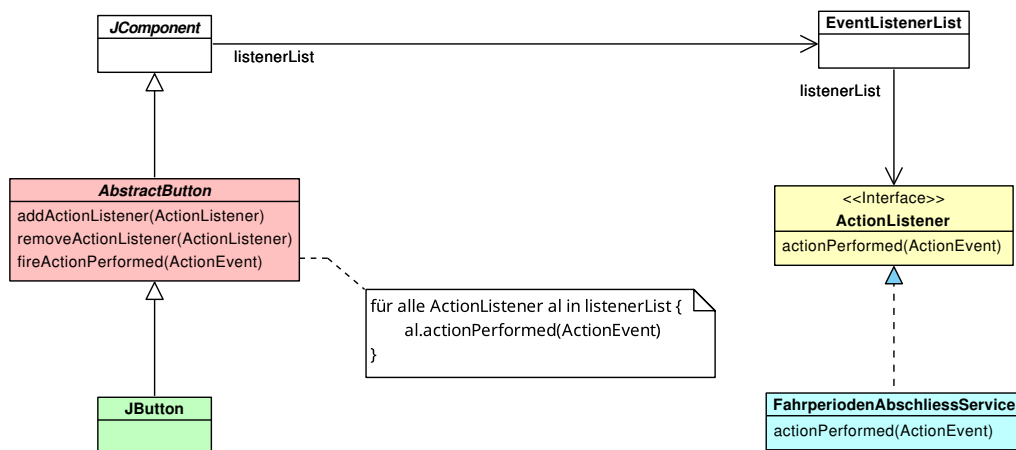
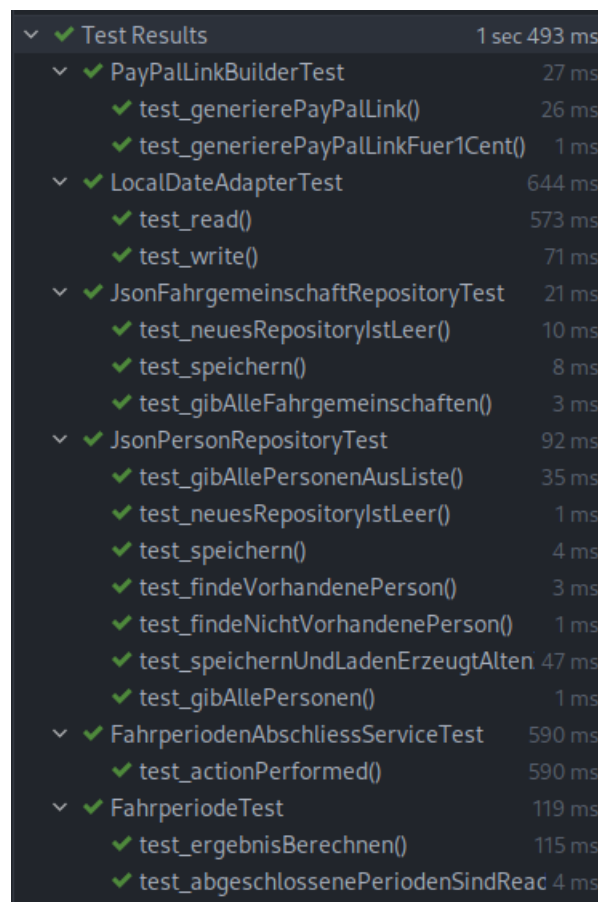


Abbildung 4.2: Konkrete Anwendung des Beobachter-Entwurfsmusters

5 Unit Tests

Zur zukünftigen Qualitätssicherung und frühzeitigen Erkennung von Fehlern wurden im Rahmen des Programmentwurfs an einigen sinnvollen Stellen exemplarische Unit Tests implementiert. Zum Einsatz kamen hier JUnit 5, AssertJ sowie EasyMock. Abbildung 5.1 stellt eine Übersicht über die erstellten Unit Tests dar.



✓ Test Results	1 sec 493 ms
✓ PayPalLinkBuilderTest	27 ms
✓ test_generierePayPalLink()	26 ms
✓ test_generierePayPalLinkFuer1Cent()	1 ms
✓ LocalDateAdapterTest	644 ms
✓ test_read()	573 ms
✓ test_write()	71 ms
✓ JsonFahrgemeinschaftRepositoryTest	21 ms
✓ test_neuesRepositoryIstLeer()	10 ms
✓ test_speichern()	8 ms
✓ test_gibAlleFahrgemeinschaften()	3 ms
✓ JsonPersonRepositoryTest	92 ms
✓ test_gibAllePersonenAusListe()	35 ms
✓ test_neuesRepositoryIstLeer()	1 ms
✓ test_speichern()	4 ms
✓ test_findeVorhandenePerson()	3 ms
✓ test_findeNichtVorhandenePerson()	1 ms
✓ test_speichernUndLadenErzeugtAlten	47 ms
✓ test_gibAllePersonen()	1 ms
✓ FahrperiodenAbschliessServiceTest	590 ms
✓ test_actionPerformed()	590 ms
✓ FahrperiodeTest	119 ms
✓ test_ergebnisBerechnen()	115 ms
✓ test_abgeschlossenePeriodenSindReac	4 ms

Abbildung 5.1: Übersicht über die erstellten Unit Tests

5.1 ATRIP-Regeln

Die ATRIP-Regeln bieten Richtlinien, die bei der Erstellung von guten und nachhaltigen Unit Tests unterstützen sollen.

5.1.1 Automatic

Die Automatic Regel besagt, dass Tests eigenständig ausführbar sein sollten. Dies inkludiert eine selbstständige Überprüfung der Ergebnisse sowie auch eine Unabhängigkeit von Nutzerinteraktion, wie etwa Eingabedialogen. Grundsätzlich wurden alle Tests so konzipiert, dass sie gemäß den genannten Kriterien automatisch ablaufen.

Allerdings ist anzumerken, dass zwar die Durchführung aller Tests automatisiert ist, es jedoch bei den [Tests des PayPalLinkBuilders](#) zu einem Problem kommt. Für den Einsatz der Klasse `PayPalLinkBuilder` wird vorausgesetzt, dass die Umgebungsvariable `PAYPAL_USERNAME` gesetzt ist. In anderen Programmiersprachen (etwa Node.js) stellt es kein Problem dar, aus einem laufenden Prozess Umgebungsvariablen zu setzen. Somit könnte der Test diese Variable auf den erforderlichen Wert setzen. Java hingegen betrachtet die Umgebungsvariablen, mit denen die JVM aufgerufen wurde, jedoch als immutable, weshalb sie zur Laufzeit nicht verändert werden können. Deshalb muss der Test mit einer vorab erstellten Run-Configuration ausgeführt werden, die sicherstellt, dass die Umgebungsvariable auf den erforderlichen Wert gesetzt wird. Dies ist zwar unpraktisch, steht einer automatisierten Testausführung aber grundsätzlich nicht im Wege.

5.1.2 Thorough

Die Thorough Regel besagt, dass gute Tests alles Notwendige abdecken sollten. Notwendig wird hierbei durch die Rahmenbedingungen bestimmt, wie etwa Tests, die in der Vergangenheit aufgetretene Fehlerfälle abdecken. Da im Rahmen des Programmentwurfs nur eine geringe Anzahl Tests angefertigt wurde, ist natürlich klar, dass damit nicht annähernd alle Teile des Programms abgedeckt sind. Auch wurde das Programm für den Programmentwurf von Grund auf neu entwickelt, weshalb noch keine Bugs bekannt sind, die von Nutzern hätten gefunden werden können. „Glücklicherweise“ wurde jedoch beim Implementieren der Klasse `PayPalLinkBuilder` ein Bug festgestellt, der sich für die

Demonstration der Regel eignet. Bei einer ersten Implementierung gab es Probleme mit einer falschen Umwandlung von Centbeträgen, weshalb dies korrigiert und mit einem expliziten Test abgedeckt wurde ([test_generierePayPalLinkFuer1Cent](#)).

5.1.3 Repeatable

Gemäß der Repeatable Regel sollten Tests jederzeit automatisch durchführbar sein und dabei stets das gleiche Ergebnis liefern. Mit den in Unterabschnitt 5.1.1 getroffenen Maßnahmen wurde der Grundstein für Repeatable Tests bereits gelegt. Weiterhin ist Zufall eine häufige Fehlerquelle. Als Gegenstand einiger Tests werden Personen-Objekte erstellt. Diese generieren bei ihrer Erstellung eine ID, welche für den weiteren Verlauf eine wichtige Bedeutung hat. Für den (wohlgemerkt hier sehr unwahrscheinlichen) Fall, dass zufällig zweimal dieselbe ID generiert wurde, wodurch die Tests fehlschlagen würden, wurde eine [Einrichtung zur Prüfung](#) der IDs eingebaut, bevor diese im Test verwendet werden.

5.1.4 Independent

Gute Tests sollten unabhängig voneinander sein. Das heißt, dass diese jederzeit in jeder beliebigen Zusammenstellung und Reihenfolge ausgeführt werden können. Dies ist nur möglich, wenn Tests keine impliziten Abhängigkeiten untereinander aufbauen – also zum Beispiel gemeinsam genutzte Datenstrukturen. Für alle Tests wurde die Regel der Unabhängigkeit dadurch umgesetzt, dass jegliche getestete Strukturen (wie zum Beispiel das [JsonPersonRepository](#)) für jeden Test neu und in einer lokalen Variable instantiiert werden. Somit wird eine Beeinflussung durch vorher gelaufene Tests ausgeschlossen. Ein Test greift auch auf das Dateisystem zu. Um hierbei eine ungewollte Abhängigkeit zu vermeiden wird für den Zugriff ein vom [Testframework bereitgestellter Ordner](#) genutzt.

5.1.5 Professional

Unit Tests sind als produktionsrelevanter Code anzusehen. Problem ist hierbei allerdings, dass Testcode selbst keinen Tests unterliegt und fehlerhafte Tests einen Rattenschwanz an Kosten hinter sich ziehen können. In Folge leitet die Professional Regel ab, dass Unit

Tests möglichst leicht verständlich sein sollten. Im Programmentwurf finden sich sowohl Beispiele für sehr gut strukturierte Unit Tests, als auch für verworrene und schwer zu verstehende. Die Tests des `PayPalLinkBuilder` sind sehr kurz gehalten und folgen zudem der AAA-Normalform für Unit Tests (Arrange, Act, Assert). Betrachtet man hingegen [diesen Test](#) des `JsonFahrgemeinschaftRepository`, so folgt dieser auch grob der AAA-Normalform, jedoch ist er äußerst unübersichtlich geschrieben. Es wird zur Assertion Gebrauch von der Java Reflection API gemacht und das Ergebnis mittels einer inline erstellten Map verglichen. Selbst wenn man den Test selbst geschrieben hat, entsteht nach kurzer Zeit das Problem, diesen nicht mehr richtig lesen zu können.

5.2 Code Coverage

Für das gesamte Projekt wurde anhand der Unit Tests der Module die Testabdeckung analysiert. Diese ist in Abbildung 5.2 dargestellt. Für das gesamte Projekt liegt die Line-Coverage bei 21 %. Auffällig ist, dass Pakete wie `gui` und `main` keinerlei Abdeckung haben, was schlicht daran liegt, dass hierfür keine Tests geschrieben wurden. Für das `model`-Paket (welches den Domain-Code enthält) hingegen wurden auch lediglich ein Test angefertigt, jedoch liegt hier die Abdeckung bei rund 55 %. Dies ist darauf zurückzuführen, dass Klassen aus diesem Paket in den Tests anderer Module verwendet werden. Natürlich ist solch eine indirekte Testabdeckung eigentlich nicht wünschenswert. Einerseits könnte man argumentieren, dass dies der Umsetzung Independent Regel entgegenwirkt, andererseits und vorrangig ergibt sich das Problem, dass die Domain-Schicht aktuell fast ausschließlich von Code aus höheren Schichten getestet wird. Diese Schichten sind kurzlebig und könnten theoretisch jederzeit obsolet wäre, wodurch die Domain-Schicht praktisch von einem Moment auf den anderen nicht mehr von Tests abgedeckt wäre.

Element	Class, %	Method, %	Line, %
all	46% (19/41)	27% (48/176)	21% (139/660)
devutil	0% (0/2)	0% (0/6)	0% (0/43)
DataGenerator	0% (0/1)	0% (0/4)	0% (0/30)
TelegramTest	0% (0/1)	0% (0/2)	0% (0/13)
gui	0% (0/16)	0% (0/51)	0% (0/347)
Controller	0% (0/1)	0% (0/3)	0% (0/6)
ErgebnisGUI	0% (0/1)	0% (0/3)	0% (0/17)
FahrgemeinschaftGUI	0% (0/4)	0% (0/12)	0% (0/45)
FahrperiodeGUI	0% (0/3)	0% (0/10)	0% (0/72)
IPopup	100% (0/0)	100% (0/0)	100% (0/0)
MainGUI	0% (0/2)	0% (0/5)	0% (0/37)
MitgliederFahrgemeinschaftGUI	0% (0/2)	0% (0/9)	0% (0/60)
NeueFahrperiodeGUI	0% (0/1)	0% (0/3)	0% (0/33)
NeueFahrtGUI	0% (0/1)	0% (0/3)	0% (0/34)
NeuePersonGUI	0% (0/1)	0% (0/3)	0% (0/43)
main	0% (0/1)	0% (0/2)	0% (0/10)
Main	0% (0/1)	0% (0/2)	0% (0/10)
model	100% (14/14)	37% (34/91)	55% (97/175)
Adresse	100% (1/1)	40% (2/5)	70% (7/10)
Distanz	100% (1/1)	40% (2/5)	30% (4/13)
Fahrgemeinschaft	100% (1/1)	33% (3/9)	55% (10/18)
FahrgemeinschaftRepository	100% (0/0)	100% (0/0)	100% (0/0)
Fahrperiode	100% (1/1)	24% (6/25)	54% (23/42)
Fahrt	100% (1/1)	40% (2/5)	40% (4/10)
Geldbetrag	100% (1/1)	50% (3/6)	72% (8/11)
Ort	100% (1/1)	40% (2/5)	66% (8/12)
Person	100% (1/1)	36% (4/11)	60% (12/20)
PersonRepository	100% (0/0)	100% (0/0)	100% (0/0)
Strasse	100% (1/1)	40% (2/5)	66% (8/12)
Streckeneinheit	100% (3/3)	44% (4/9)	50% (8/16)
Waehrung	100% (2/2)	66% (4/6)	50% (5/10)
paypal	100% (1/1)	100% (1/1)	88% (8/9)
PayPalLinkBuilder	100% (1/1)	100% (1/1)	88% (8/9)
services	100% (1/1)	100% (3/3)	100% (10/10)
FahrperiodenAbschliessService	100% (1/1)	100% (3/3)	100% (10/10)
IPayPalLinkBuilder	100% (0/0)	100% (0/0)	100% (0/0)
TelegramClient	100% (0/0)	100% (0/0)	100% (0/0)
speicher	60% (3/5)	50% (10/20)	51% (24/47)
JsonFahrgemeinschaftRepository	100% (1/1)	25% (2/8)	17% (4/23)
JsonPersonRepository	100% (1/1)	75% (6/8)	89% (17/19)
LocalDateAdapter	100% (1/1)	100% (2/2)	100% (3/3)
StreckeneinheitInstanceCreator	0% (0/1)	0% (0/1)	0% (0/1)
WaehrungInstanceCreator	0% (0/1)	0% (0/1)	0% (0/1)
telegram	0% (0/1)	0% (0/3)	0% (0/13)
Telegram	0% (0/1)	0% (0/3)	0% (0/13)

Abbildung 5.2: Coverage-Analyse des Projekts

5.3 Einsatz von Mocks

Mocks werden bei Unit Tests eingesetzt, um Abhängigkeiten während eines Tests zu ersetzen. So hängt beispielsweise das Ergebnis eines Tests nicht von einer externen Datenbank ab. Weiterhin könnte so auch einer Kopplung, wie im letzten Absatz beschrieben, entgegengewirkt werden.

Im Programmwurf wurden in mehreren Tests Mocks eingesetzt. Besonders gut zu sehen ist dies in den [Tests des LocalDateAdapters](#). Hier wird bei beiden Tests mittels Mocks vermieden, dass ein echter JsonReader/-Writer erstellt werden muss, der wiederum einen Ein-/Ausgabestream voraussetzt – beispielsweise aus einer Datei.

6 Refactoring

6.1 Code Smells identifizieren

Prägnante Fälle von Shotgun Surgery und Large Classes sind im Code des Fahrtkostenrechners praktisch nicht vorhanden, was auch darin begründet liegt, dass das Projekt relativ klein ist. Switch Statements wurden in der Codebasis nicht verwendet und mit Kommentaren war der bearbeitende Student auch sparsam. Nichtsdestotrotz finden sich im Code-Substrat einige Code Smells, welche größtenteils aus Duplicated Code und Long Methods bestehen.

Insbesondere im GUI Plugin finden sich sehr viele *Code-Dopplungen*. Die wohl prominenteste dieser Dopplungen entsteht durch die [Erzeugung von Popups](#). Diese wurde so umgesetzt, dass das GUI-Fenster, aus dem das Popup hervorgeht, gesperrt wird, wenn das Popup gestartet wird. Wird das Popup geschlossen, so wird auch das Ursprungsfenster wieder freigegeben.

Long Methods finden sich praktisch in allen GUI-Klassen abgesehen vom Controller. Alle GUI-Klassen folgen grob demselben schematischen Aufbau, bei dem sämtliche Elemente im Konstrukt der jeweiligen Ansicht aufgebaut werden. Das macht zum einen den Konstruktor sehr unübersichtlich, andererseits sind die Ansichten auch sehr unflexible gegenüber nachträglichen Änderungen. Das Anzeigen neuer Elemente oder das Aktualisieren der angezeigten Daten ist meist nur über einen erneuten Aufruf des Konstruktors möglich.

Auch in der Klasse `Telegram` im Integrations-Plugin findet sich ein Code-Smell, der sich trotz kompakter Erscheinung der Methode am ehesten als *Long Method* einordnen lässt. Hauptproblem ist hier, dass sehr viele verschiedene Aktionen ausgeführt werden, die thematisch nicht zusammenhängen, was die Lesbarkeit stark beeinträchtigt.

Eine weitere lange und unübersichtliche Methode findet sich in der Klasse [Fahrperiode](#). In dieser Periode werden die Kosten für jeden Mitfahrer innerhalb einer Fahrperiode berechnet. In der Folge enthält diese Methode mehrere ineinander geschachtelte Schleifen-

aufrufe, die über alle Fahrten in der Periode und über alle jeweils beteiligten Mitfahrer iterieren. Auch diese Methode ist kaum lesbar und bedarf dringend einer Verbesserung.

6.2 Refactorings anwenden

Um den erwähnten Code Smells entgegenzuwirken, werden verschiedene Refactorings angewendet. Das wichtigste hierbei ist *Extract Method*, da sich mit diesem Refactoring die meisten Problemstellen schon deutlich verbessern lassen.

Die ersten Refactorings werden an der Klasse `Telegram` vorgenommen. Bislang enthielt die Klasse lediglich die Methode „send“. Aus dieser wird nun zunächst die Methode `telegramToken()` extrahiert, welche das benötigte Telegram Token aus der Umgebungsvariable ausliest. Anschließend wird für die beiden Deklaration der beiden Variablen `String uri` und `HttpRequest request` jeweils das Refactoring *Replace Temp with Query* durchgeführt. Dabei wird für beide Aufrufe jeweils eine eigene Methode erstellt, welche anschließend anstelle der deklarierten Variable verwendet wird. Durch die durchgeführten Refactorings konnte die ursprüngliche Methode erfolgreich auf vier einzelne Methoden aufgeteilt werden, die jeweils deutlich besser verständlich sind.

Als zweites Code Smell wird die Duplizierung der Popup-Sperren im GUI-Plugin angegangen. Da die sich wiederholende Funktionalität sehr generisch ist, ist hier der einfachste Weg, die entsprechende Funktionalität in eine weitere Klasse auszulagern (Method Extraction in andere Klasse). Hierfür empfiehlt es sich, dem Controller eine statische Methode hinzuzufügen, die die entsprechende Funktionalität übernimmt. Dazu wird im Controller die *neue Methode* mit zwei `JFrames` als Parameter (Parent und Popup) definiert. Für den Methodenkörper genügt es, eine der vorhandenen Sperrroutinen leicht hinsichtlich der Methodenparameter zu generalisieren. Anschließend können sämtliche Duplikationen durch einen Aufruf der statischen Methode ersetzt werden. Hierdurch konnten Duplikationen in vier Klassen entfernt werden (`MainGUI`, `FahrgemeinschaftGUI`, `FahrperiodeGUI` und `MitgliederFahrgemeinschaftGUI`). Weiterhin wurde durch das Refactoring ein Interface obsolet, welches bislang für die Bereitstellung der Funktionalität unterstützend wirkte.

Ein letztes Refactoring wurde im Domain-Code für die Klasse `Fahrperiode` durchgeführt. Die Methode `getErgebnis()` konnte Mittels *Method Extraction* deutlich simpler und übersichtlicher gestaltet werden, indem die Berechnung der einzelnen Fahrten

ausgelagert wird. Weiterhin wurde ein weiteres *Replace Temp with Query* Refactoring durchgeführt, um die Berechnung des Fahrtbeitrages je Mitfahrer in eine weitere eigenen Methode (`betragProMitfahrer(Fahrt)`) auszulagern, welche dann anstelle der temporären Variable verwendet wird.