DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Software-Engineering II

Programmentwurf
TINF20B1
5.+6. Semester (2022/2023)

Thema:
Kostenrechner für Fahrgemeinschaften

Dozent:
Daniel Lindner

Bearbeitender:
Yannik Schiebelhut

Inhaltsverzeichnis

Abkürzungsverzeichnis

Abbildungsverzeichnis

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.

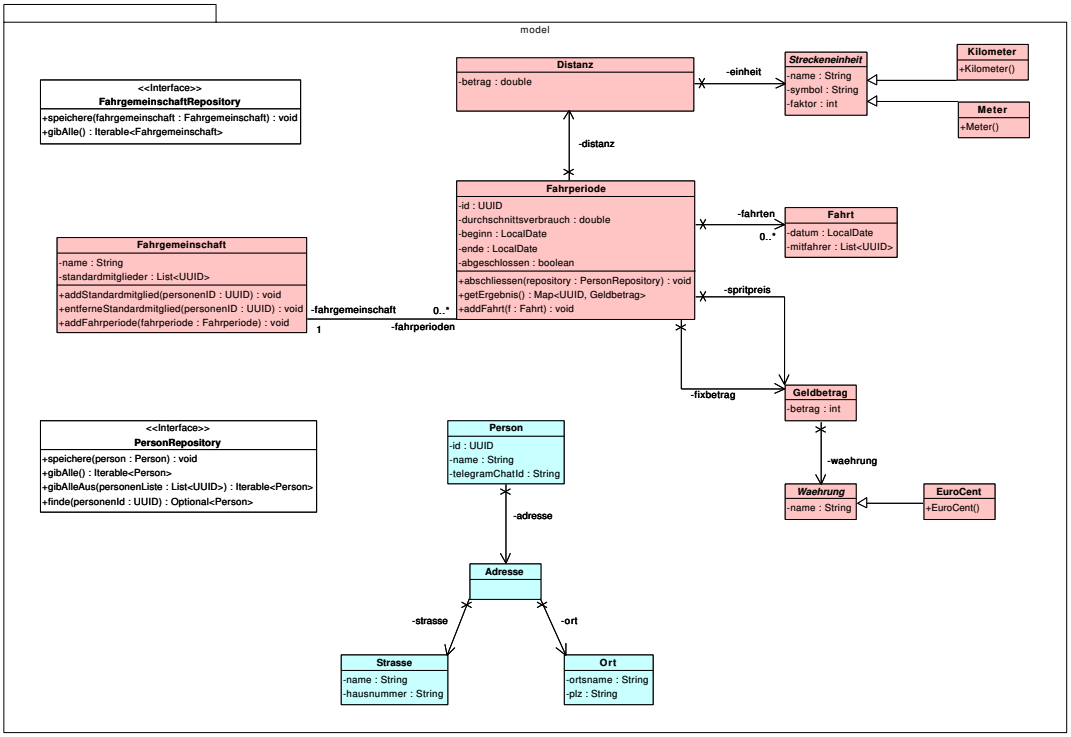

Abbildung 2.1: Domänenmodell

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 realistischerweise 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 Waehrung
- 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 Reposiotries

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 Domaine 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 Koordinierung 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 Entwurfmusters 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`,

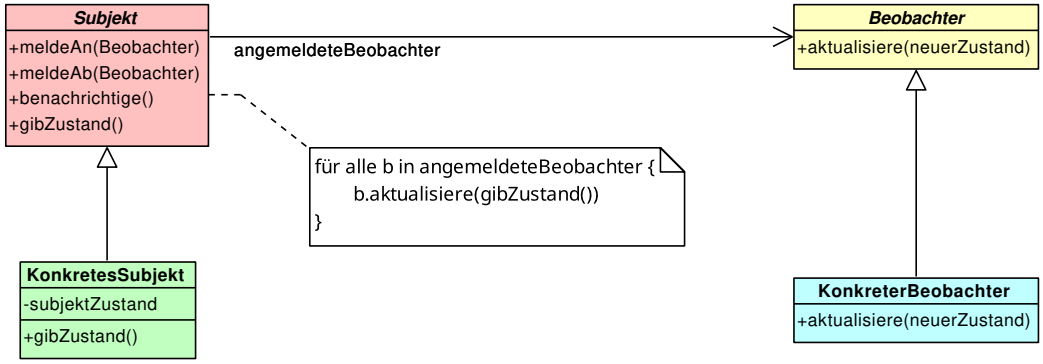

Abbildung 4.1: Abstraktes UML-Diagramm des Beobachter-Entwurfmusters

`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 ActionListeners) zu finden. Die Implementierung der Subjekte ist Teil des Java-Standards.

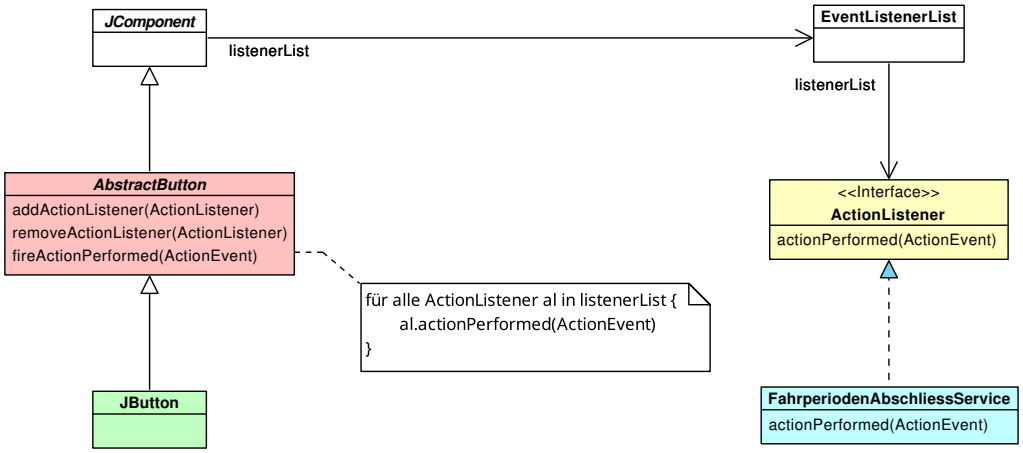

Abbildung 4.2: Konkrete Anwendung des Beobachter-Entwurfmusters

5 Unit Tests

5.1 ATRIP-Regeln

5.1.1 Automatic

5.1.2 Thorough

5.1.3 Repeatable

5.1.4 Independent

5.1.5 Professional

5.2 Code Coverage

5.3 Code Coverage