

SWE-Dokumentation

Allgemein & Übersicht

Link zum öffentlichen GitHub-Repository:

Das Ziel der Projektes war es eine Software zur Berechnung der Transaktionsgebühren bei Paypal und ausgewählten Visa Händlern zu implementieren. Dem Nutzer stehen hierbei drei verschiedene Funktionalitäten zur Verfügung. Diese können über eine Webseite erreicht und ausgeführt werden.

- **Transaktionsgebühr berechnen:** Mit Hilfe dieser Funktionalität kann der Nutzer die Höhe der Transaktionsgebühren bei dem Kauf von Produkt/Dienstleistungen bestimmen. Er muss hierzu in das Formular sowohl die Höhe des Kaufpreises als auch die verwendete Transaktionsart auswählen. Es werden hierbei zehn verschiedene Transaktionsarten als Auswahl vorgegeben. Des Weiteren werden die Konditionen der jeweiligen Transaktionsart kurz erläutert.
Für die Berechnung der Transaktionsgebühr wird die folgende Formel verwendet:
$$\text{Transaktionsgebühren} = \text{Kaufpreis} * \text{Variable Gebühr} + \text{Gebühr pro Transaktion}$$

The screenshot shows a web browser window with the URL `localhost:8080/FeeCalculator/fee.html`. The page has a navigation bar with links: 'Transaktionsgebührenrechner', 'Übersicht', 'Transaktionsgebühr berechnen', 'Restbetrag berechnen', and 'Günstigste Transaktionsart bestimmen'. The main content area is titled 'Transaktionsgebühren berechnen'. It contains a form with two main sections: 'Kaufpreis' and 'Transaktionsart'. The 'Kaufpreis' section has a text input field with the value '25' and a currency selector set to '€'. Below it is a small text label: 'Der Geldbetrag, auf welchen die Transaktionsgebühren berechnet werden sollen.' The 'Transaktionsart' section has a dropdown menu with the selected option 'Paypal: Mikrozahlung (10% und 10 Cent)'. Below the dropdown is a small text label: 'Die Transaktionsart bestimmt über die Höhe der Transaktionsgebühr.' At the bottom of the form is a blue button labeled 'Berechne Transaktionsgebühr'.

- **Restbetrag berechnen:** Mit Hilfe dieser Funktionalität kann der Nutzer den notwendigen Kaufpreis berechnen, wenn er einen bestimmten Geldbetrag (Restbetrag) erhalten möchte. Hierzu gibt er über ein Formular die Höhe des Restbetrages ein. Zusätzlich wird, wie bei der vorherigen Funktionalität, eine von zehn verschiedenen Transaktionsarten ausgewählt.
Die Berechnung der Höhe des Kaufpreises erfolgt über eine Interpolation, da keine eindeutig Gleichung des Problems vorhanden ist. Die Interpolation genügt auch den Ansprüchen, da die Berechnung lediglich auf zwei Nachkommastellen genau sein muss.

The screenshot shows a web browser window with the URL `localhost:8080/FeeCalculator/reversefee.html`. The page has the same navigation bar as the previous screenshot. The main content area is titled 'Restbetrag berechnen'. It contains a form with two main sections: 'Restbetrag' and 'Transaktionsart'. The 'Restbetrag' section has a text input field with the value '25' and a currency selector set to '€'. Below it is a small text label: 'Der Geldbetrag, der nach Abzug der Transaktionsgebühren übrig bleiben soll.' The 'Transaktionsart' section has a dropdown menu with the selected option 'Paypal: Zahlung mit Händlerkonditionen zwischen 5000,01 und 25000,00€ (1,99 % und 35 Cent)'. Below the dropdown is a small text label: 'Die Transaktionsart bestimmt über die Höhe der Transaktionsgebühr.' At the bottom of the form is a blue button labeled 'Berechne Restbetrag'.

-

Die Werte der Spalte „Currency“ müssen den Abkürzungen laut der ISO 4217 entsprechen. Des Weiteren ist zu beachten, dass zum aktuellen Zeitpunkt lediglich, die Transaktionen mit Euros für die Ermittlung der günstigsten Transaktionsart berücksichtigt werden.

Als letzter Punkt bei dieser Funktionalität ist anzumerken, dass die CSV-Datei nur die Transaktionen des letzten Monats beinhalten sollte, da anhand der Transaktionshöhe bestimmte Transaktionsarten in der Berechnung nicht berücksichtigt werden. Ebenso werden die Transaktionsarten Paypal Freunde und Familie als auch Paypal Spenden nicht bei der Auswahl der günstigsten Transaktionsart berücksichtigt.

Lokales Setup:

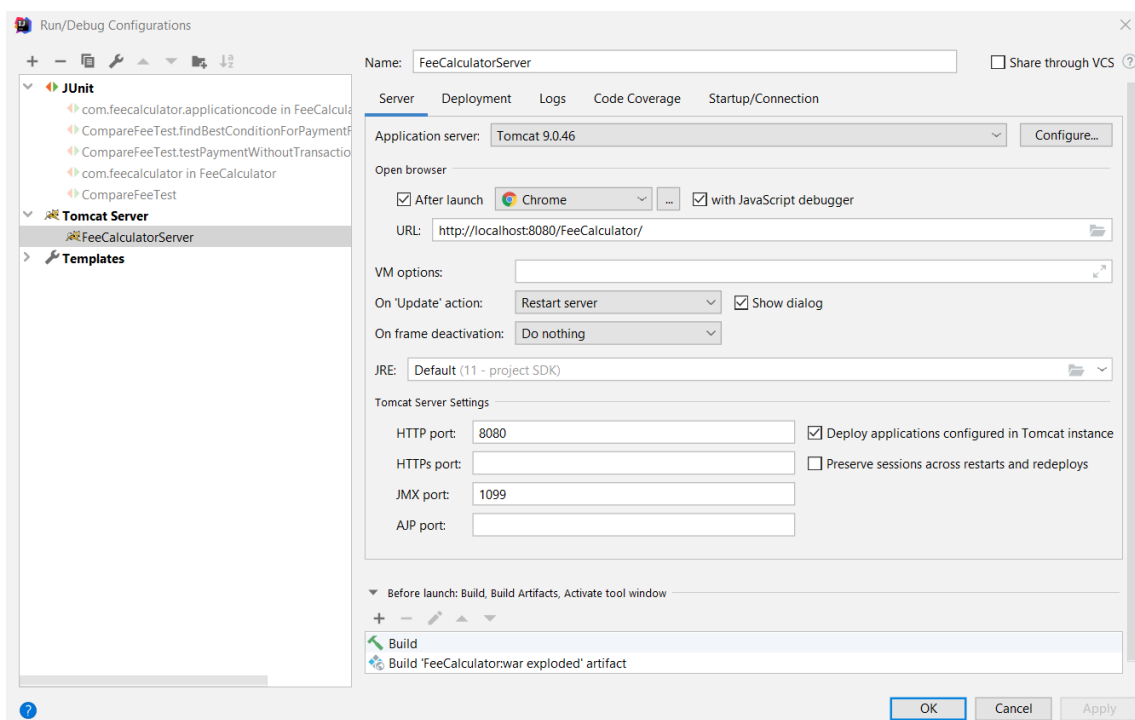
Die Implementierung des Backends der Software erfolgte in Java. Das Frontend wurde mittels HTML erstellt. Für die Unterstützung und eine schnellere Entwicklung wurde hierbei die Hilfe des Bootstrap-Frameworks in Anspruch genommen. Die Kommunikation zwischen Frontend und Backend erfolgt durch Java Servlets. Es wurde sich für diesen Technologiestack entschieden, da bereits Erfahrungen durch vorherige Projekte gesammelt werden konnten und dies den Einstieg in die Implementierung vereinfacht.

Die Anwendung wurde als Web Application Archive (WAR-File) entwickelt. Dies ist ein Dateiformat, in welches die Servlets als vollständige Webanwendungen zusammen mit dem HTML-Code verpackt werden. Hierzu ist eine bestimmte Verzeichnisstruktur vorgeschrieben. Dieses wird automatisch von der IDE beim Erstellen des Projektes erzeugt.

Für das lokale Setup muss ein Webserver auf dem jeweiligen Rechner installiert sein. Ich empfehle Apache Tomcat auf dem Computer zu installieren und einen Webserver über die IDE zu starten. Eine Anleitung für die Installation von Apache Tomcat ist hier (<https://tomcat.apache.org/download-10.cgi>) zu finden. Alternativ kann ich auch folgendes YouTube Video empfehlen: <https://www.youtube.com/watch?v=pKMgr8uNvGM>

Um das Projekt lokal auf dem Webserver zu deployen, empfiehlt es sich die IDE als Hilfe zu nutzen. Es muss die Konfiguration für einen Webserver erstellt werden. Hierbei muss als Artefakt, das deployed wird, „FeeCalculator“ ausgewählt werden.

Die Konfiguration für die IntelliJ ist im untenstehenden Screenshot dargestellt.



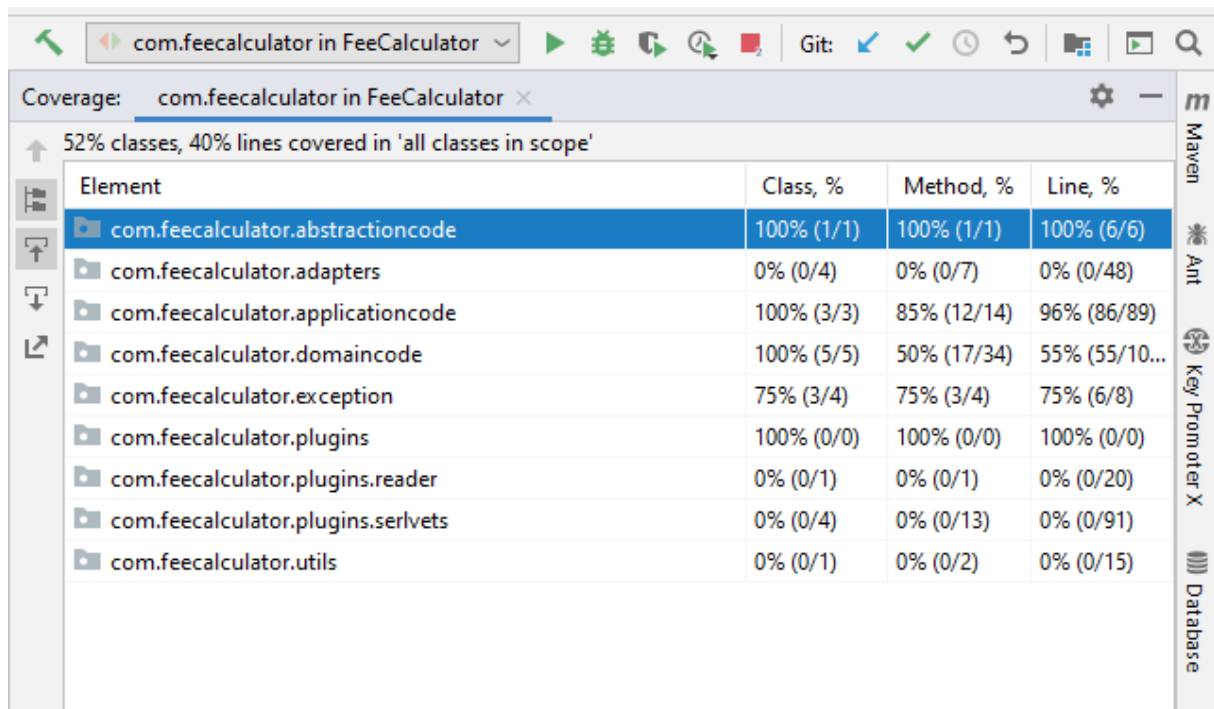
Falls Eclipse als IDE verwendet wird, muss zunächst ein Tomcat Server erstellt werden. Anschließend muss auf diesen ebenso das Artefakt „FeeCalculator“ deployed werden.

Bei Fragen zum lokalen Setup stehe ich gerne als Hilfe bereit. Das Setup wurde für sowohl IntelliJ als auch Eclipse getestet.

Unit Tests:

Es wurden Tests für das Software-Projekt für den Application und Abstraction Code geschrieben. Es wären natürlich auch Tests für die weiteren Klassen möglich gewesen, hierauf wurde allerdings aus Zeitgründen verzichtet. Die Anzahl der geschriebenen Tests beläuft sich auf 34.

Bei der Implementierung der Tests wurde darauf geachtet, dass möglichst alle Methoden des zu testenden Bereiches auch durchlaufen werden. Dies kann gut mit Hilfe der Code Coverage genauer analysiert und untersucht werden. Für die Tests des Abstraction Code konnte diese Erwartung ohne Probleme erfüllt werden. Ein Grund hierfür, dass dies so einfach war, ist, dass der Abstraction Code lediglich eine Klasse mit einer Methode umfasst. Aber auch bei dem Application Code konnte nahezu jede Methode getestet werden. Lediglich zwei Methoden wurden bei den Tests nicht durchlaufen. Da hier es sich hierbei um die Methoden `getCalculateTransactionFee()` und `setCalculateTransactionFee(...)` der Klasse `CompareFee` handelt und diese keine Logik beinhalten, kann das Testen dieser Methoden Problemlos vernachlässigt werden.



Coverage: com.feecalculator in FeeCalculator

52% classes, 40% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %
com.feecalculator.abstractioncode	100% (1/1)	100% (1/1)	100% (6/6)
com.feecalculator.adapters	0% (0/4)	0% (0/7)	0% (0/48)
com.feecalculator.applicationcode	100% (3/3)	85% (12/14)	96% (86/89)
com.feecalculator.domaincode	100% (5/5)	50% (17/34)	55% (55/100)
com.feecalculator.exception	75% (3/4)	75% (3/4)	75% (6/8)
com.feecalculator.plugins	100% (0/0)	100% (0/0)	100% (0/0)
com.feecalculator.plugins.reader	0% (0/1)	0% (0/1)	0% (0/20)
com.feecalculator.plugins.servlets	0% (0/4)	0% (0/13)	0% (0/91)
com.feecalculator.utils	0% (0/1)	0% (0/2)	0% (0/15)

Weitere Maße, die mit Hilfe der Code Coverage gemessen werden können, sind sowohl die Line als auch die Branch Coverage. Die Branch Coverage fungierte im weiteren Verlauf als wichtiges Maß für die Beurteilung der Testabdeckung, da mit dieser sichergestellt werden kann, dass möglichst Abzweigungen des Codes durchlaufen werden. Es wurde geachtet, dass die Branch Coverage für den Abstraction und Application Code bei 100% liegt. Dieses Ziel konnte erfüllt werden, indem auch die Randbedingungen, die Exceptions erzeugen, getestet wurden. Extrem wichtig ist hierbei anzumerken, dass trotz der hohen Branch Coverage und den erfolgreich durchlaufenden Tests weiterhin nicht garantiert werden kann, dass der Code fehlerfrei ist. Eine Auszeichnung der Tests mit verschiedenen Tags bzw. Motivation wurde ebenso nicht durchgeführt, da lediglich Requirements und keine Bug Fixes umgesetzt wurden. Aus vorherigen Projekten kann ich dies bei umfangreicheren Aufgaben nur empfehlen, da dadurch die Herkunft und die Motivation für einen Test nachvollzogen werden kann.

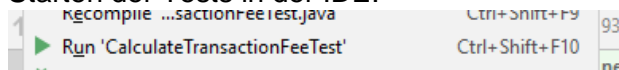
Eine weitere Anforderung war es, Mocks innerhalb der Tests einzusetzen. Während dieses Projektes konnte Mocks sinnvoll den Tests für die Klasse CompareFee eingesetzt werden. Der Link zu der Testklasse befindet sich hier (<https://github.com/schlueterkai/FeeCalculator/blob/main/src/test/java/com/feecalculator/applicationcode/CompareFeeTest.java>). Innerhalb der Klasse CompareFee sollen die Höhe der Transaktionsgebühren für verschiedene Transaktionsarten verglichen und die günstige ausgewählt werden. Da die Berechnung der Gebühren innerhalb einer anderen Klasse durchgeführt wird, wird diese Berechnung durch ein Mock zurückgegeben. Dadurch werden die Abhängigkeiten zu anderen Klassen während des Tests ersetzt und es kann die Funktionalität der Klasse isoliert getestet werden.

Für die Implementierung der Tests wurde das JUnit Framework verwendet. Die Mocks wurden mit Hilfe von Mockito erzeugt und verwaltet.

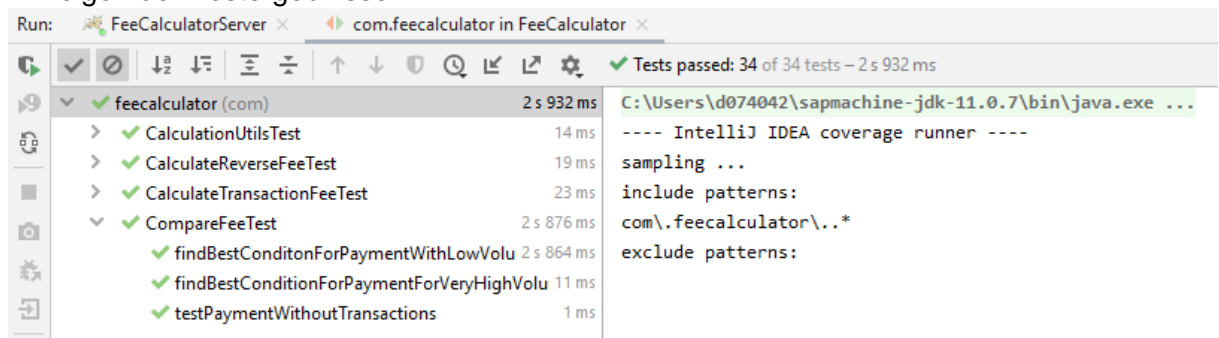
Ein weiterer wichtiger Punkt für die Implementierung von Tests ist es die ATRIP-Regeln zu beachten. Dies sind fünf verschiedene Regeln, welche in dem Buch „Pragmatic Unit Testing“ beschrieben wurden. Im Folgendem soll nun kurz auf die ATRIP-Regeln eingegangen und diese erläutert werden.

- Automatic: Diese Regel besagt, dass die Tests eigenständig durchlaufen werden müssen und kein manueller Eingriff notwendig ist. Es muss also kein Dialog ausgefüllt werden und es ist keine Werteeingabe notwendig. Des Weiteren werden die Ergebnisse selbstständig von den Tests überprüft. Dieser Punkt wurde ohne Probleme umgesetzt. Die Überprüfung der Ergebnisse wurde innerhalb der Tests mit Hilfe von Assertions durchgeführt. Beispiel in der Klasse CalculateTransactionFeeTest. Außerdem können die Tests ohne Probleme automatisch durchgeführt werden. Dies kann mit ein beliebigen IDE getestet werden. Außerdem wurde in den Tests der Klasse darauf geachtet, dass die Fließkommazahlenvergleiche die Ungenauigkeit berücksichtigen und daher eine Genauigkeit in der E-Notation mitgegeben.

Starten der Tests in der IDE:



Anzeigen der Testergebnisse:



- Thorough: Gibt an, dass die Tests alles Notwendige testen. Dies wurde sichergestellt, indem die Tests für jede Transaktionsart durchgeführt wurden und zusätzlich ebenso geworfene Exceptions und damit die Randbedingungen in den Tests behandelt wurden. Dies spiegelte sich auch in der bereits beschriebenen Code Coverage nieder. Da die Software leider noch nicht live gegangen ist und Nutzer diese nach Bugs untersuchen konnten, wurden noch keine Bug-Issues für neue Tests erstellt. Sobald diese auftreten und falls das Projekt weiter gepflegt und aktiv genutzt wird, ist geplant die Tests verschiedene Auszeichnungen zu verleihen, um eine bessere Übersichtlichkeit zu gewährleisten.

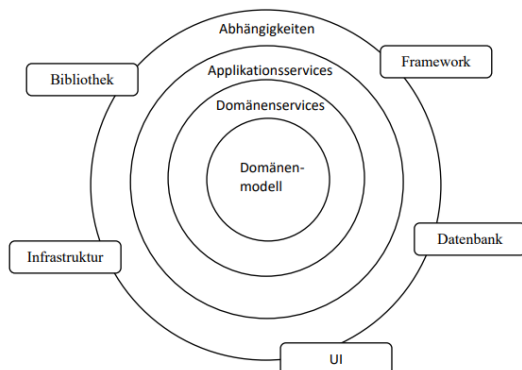
- Repeatable: Ein Test sollte jeder Zeit automatisch durchführbar sein und immer das gleiche Ergebnis zurückliefern. Dieser Punkt wurde ebenfalls ohne Probleme umgesetzt. Falls bei einigen Funktionalitäten Zufall vorhanden gewesen wäre, hätten diese Teile weggemockt werden müssen, was die Isolation und Unabhängigkeit von der Umgebung gewährleistet hätte. In dem Projekt liegen keine Tests vor, die spontan abbrechen oder unterschiedliche Ergebnisse zurückliefern.
- Idendent: diese Regel besagt, dass die Tests jederzeit in beliebiger Zusammenstellung und Reihenfolge durchführbar sein müssen. Es dürfen daher keine implizite Abhängigkeiten zwischen den Tests bestehen. Dies ist beispielsweise der Fall, wenn der erste Test eine Datei öffnet und der zweite Test auf diese geöffnete Datei zugreifen würde. Dieses Problem wurde umgegangen, indem darauf geachtet wurde, dass jeder Tests genau einen Aspekt der jeweiligen Klasse testet. Dies hat den Vorteil, dass im Fehlerfall nur ein Test und nicht gleich direkt mehrere kaputt gehen.
- Professional: der letzte Punkt der ATRIP-Regeln besagt, dass der geschriebene Testcode ebenfalls produktionsrelevant ist und deshalb möglichst leicht verständlich sei sollte. Ebenfalls sollten nur relevante Aspekte des Coding getestet werden. Besonders gut sind die lesbaren Tests in der CalculateTransactionFeeTest gelungen. Der Fokus wurde hierbei klar auf die Lesbarkeit gelegt, weshalb die Benennung der Variablen möglichst verständlich und dem Sachverhalt gerecht, durchgeführt wurde. Ebenso wurde der konkret getestete und wiederverwendbare Teil in den Methoden performTransactionFeeTestWith(...) und performPaymentFeeTestWith(...) ausgelagert.

Allgemein ist zu den ATRIP-Regeln anzumerken, dass für alle Tests umgesetzt wurden. Besonders erfolgreich ist dies meines Erachtens in der Klasse CalculateTransactionFeeTest gelungen, weshalb die Test dieser Klasse für die Bewertung der ATRIP-Regeln am besten genutzt werden kann.

Clean Architecture

Ein Problem bei der Softwareentwicklung ist, dass diese sich im beständigen Wandel befindet. Technologien und Frameworks ändern sich ständig, sodass sich der Quellcode ändern muss, damit die Software aktuell bleibt und den neusten Anforderungen entspricht. Eine große Herausforderung ist es daher eine nachhaltige und langfristige Architektur zu entwerfen, in welcher die Technologien und Plugins einfach und ohne viel Aufwand ausgetauscht werden können. Ein Ansatz hierzu ist die Verwendung der Clean Architecture.

Diese unterscheidet zwischen langlebigem (zentralen) und kurzlebigen (peripherem) Sourcecode. Ziel ist es einen technologieunabhängigen Kern zu besitzen, in welchem die eigentliche Anwendung und Logik implementiert ist. Diese Architektur wird als Onion-Architektur bezeichnet, da diese wie eine Zwiebel in verschiedene Schichten aufgebaut ist.



Wichtig hierbei zu beachten, dass sich im Kern der Architektur der langlebige Code befindet und weiter außer der Code kurzlebiger wird. Dies bedeutet, dass sich dieser häufiger ändert und daher angepasst werden muss. Es sind nur Abhängigkeiten von außen nach innen erlaubt. Innerhalb des Kerns sollte möglichst keine Frameworks oder Bibliotheken verwendet werden.

In der Clean Architecture wird zwischen fünf verschiedenen Schichten unterschieden. Im Folgendem soll kurz die Umsetzung der Clean Architecture für das Software-Projekt erläutert werden. Gefordert war eine Umsetzung von mindestens zwei Schichten. Es wurde allerdings versucht die ganze Applikation mit der Clean Architecture umzusetzen, weshalb auf alle Schichten kurz eingegangen wird.

Abstraction Code (Schicht 4): Der Abstraction Code enthält domänenübergreifendes Wissen, wie mathematische Konzepte oder Algorithmen und Datenstrukturen. Das Coding wird in vielen Projekten nicht benötigt und ändert sich meist nie. Für das Projekt wurde eine Klasse bzw. eine Methode in der Abstraction Code Schicht implementiert. Da in der Anwendung mit Geldbeträgen gerechnet wird und diese nur auf zwei Kommastellen genau sein müssen, wurde eine Klasse implementiert, die für das kaufmännische Runden von Gleitkommazahlen genutzt werden kann. Diese Methode musste implementiert werden, da keine geeignete Programmbibliothek gefunden wurde, die ebenso einfach hätte genutzt werden können.

```
package com.feecalculator.abstractioncode;
```

```
public class CalculationUtils {
```

Da keine Imports für die notwendig sind, kann auch sichergestellt werden, dass keine Abhängigkeiten einer äußeren Schicht vorhanden sind.

Link zum Package:

<https://github.com/schlueterkai/FeeCalculator/tree/main/src/main/java/com/feecalculator/abstractioncode>

Domain Code (Schicht 3): Der Domain Code enthält vor allem Entitäten und implementiert organisationsweite gültige Geschäftslogik. Der Code sollte sich nur möglichst selten ändern und er Immun gegen Änderungen der Anzeigedetails, Datenbank oder dem Transport sein. Es ist daher wichtig, dass er keine Abhängigkeiten zu äußeren Schichten besitzt. Im vorliegenden Projekt befinden sich hauptsächlich Entitäten und Value Objects in dem Domain Code Package. Funktionalitäten, wie die Berechnung von Transaktionsgebühren, wurden in den Application Code verlegt, da diese sich häufiger als die Entitäten ändern. Trotzdem hätte diese ebenso sich im Domain Code befinden können. Der Übergang zwischen den beiden Schichten ist sowieso nicht genau definiert.

Durch die Überprüfung der Imports konnte erneut sichergestellt werden, dass keine Abhängigkeiten zu den äußeren Schichten vorliegen.

Link zum Package:

<https://github.com/schlueterkai/FeeCalculator/tree/main/src/main/java/com/feecalculator/domaincode>

Application Code (Schicht 2): Der Application Code einer Anwendung enthält die Anwendungsfälle, die sich aus den Anforderungen resultieren. Aus diesem Grund hatte ich mich auch entschieden die Berechnung der Transaktionsgebühren und des Restbetrags in diese Schicht zu ziehen, da dies meines Erachtens ein bisschen besser passt. Der Application Code enthält damit die anwendungsspezifische Geschäftslogik. Des Weiteren wird bspw. in der Klasse CompareFee der Fluss der Daten und Aktionen zwischen den Entitäten gesteuert.

Wie bei den vorherigen Schichten ist es auch hier wichtig, dass nur Abhängigkeiten zu den inneren Schichten bestehen. In diesem Fall liegen auch Abhängigkeiten zu dem Domain Code vor, da die Entitäten für die Anwendungsfälle natürlich benötigt werden. Anhand der Imports in den verschiedenen Klasse wurde allerdings auch wieder überprüft, dass keine Klassen der äußeren Schichten importiert werden.

Link zum Package:

<https://github.com/schlueterkai/FeeCalculator/tree/main/src/main/java/com/feecalculator/applicationcode>

Adapters (Schicht 1): Adapters werden genutzt, um die Applikation tauglich und die Plugins frisch zu halten. Sie enthalten bspw. Render-Modelle, die direkt in der GUI einer Applikation verwendet werden können. Da sich die Anzeige in der GUI im Gegensatz zu den Entitäten häufiger ändert, ist es sinnvoll Adapter einzusetzen, damit keine Anpassungen im Domain Code vorgenommen werden müssen. In diesem Projekt wurden ebenfalls Render-Modelle für die Verwendung der Daten im UI genutzt. In diesem wurde beispielsweise die Klasse Amount so formatiert, dass über eine Map direkt auf einen String, der die Höhe und die Währung enthält, auf die Daten zugegriffen werden kann. Ändert sich nun beispielsweise die Reihenfolge der Betragsanzeigen (€ 1,5 anstatt 1,5 €), kann diese Änderung an einer konkreten Stelle im Coding angepasst werden. Wichtig ist hierbei, dass die Render Modelle alle veränderlichen Inhalte der Seite berechnen und beinhalten. Da nicht alle Entitäten im Frontend angezeigt werden, wurden nur Render Modelle für die Klassen Transaction und Payment implementiert.

Link zum Package:

<https://github.com/schlueterkai/FeeCalculator/tree/main/src/main/java/com/feecalculator/adapters>

Plugins (Schicht 0): Die Plugins sind von schnellen Änderungen betroffen und enthalten meist Datentransportmittel, Frameworks und andere Werkzeuge. Sie greifen grundsätzlich nur über die Adapter zu. Hier ist auch erste Punkt, bei welchem meine Software von der Clean Architecture abweicht, da hier auch direkt auf den Application Code teilweise zugegriffen wird (Bsp. TransactionFeeServlet). Des Weiteren kommt hinzu, dass bei den Entitäten mittels setter zugegriffen wird. Bei einer sauberen Umsetzung wäre dies nicht möglich gewesen und hätte verhindert werden müssen. Trotzdem enthalten bei mir das Plugin Package den Code, der schnell kurzlebig ist und schnell ausgetauscht werden kann. Bspw. kann sich das Framework der verwendeten Web-Technologie ohne Probleme relativ schnell geändert werden ohne, dass Eingriffe in den Application, Domain oder Adapter Code notwendig ist. Des Weiteren enthält dieses Package auch das Einlesen der Transaktionsdaten. Da sich das Format der Daten auch schnell ändern kann und CSV-Dateien durch bspw. JSON-Dateien ersetzt werden können, wurde diese Funktionalität ebenso im Plugin Code implementiert.

Link zum Package:

<https://github.com/schlueterkai/FeeCalculator/tree/main/src/main/java/com/feecalculator/plugins>

Noch besser wäre die Umsetzung der einzelnen Schichten als Maven Projekt gewesen. Dies wurde allerdings aufgrund von Zeitmangel nicht mehr umgesetzt. Zudem konnten die Imports aufgrund der relativ kleinen Größe des Projektes problemlos überprüft werden.

Programming Principles:

Im Folgendem sollen die Umsetzung verschiedener Programmierprinzipien genauer erläutert und die Verwendung begründet werden. Es werden hierbei die Prinzipien SOLID, GRASP und DRY analysiert.

SOLID selbst besteht aus fünf weiteren Prinzipien, die alle unterschiedliche Ziele verfolgen:

- Single Responsibility Principle (SRP): Das SRP besagt, dass jede Klasse nur eine einzige Verantwortung besitzen soll. Verantwortung ist hierbei als „Grund für Veränderungen zu verstehen“. Dieses Prinzip führt dazu, dass es für eine Klasse maximal einen Grund darf diese zu verändern. Hält eine Klasse nun mehrere Verantwortungen gibt es mehr Gründe diese zu ändern, weshalb das Risiko für Fehler, die bei Änderungen auftreten können, steigt. Es wurde versucht dieses Prinzip umzusetzen, indem jedem Klasse eine klare Funktion zugeordnet wurde. Die Funktion sollte anhand des Names ableitbar sein. Ein Beispiel für die Umsetzung dieses Prinzips ist die Unterteilung der Berechnung des Restbetrags (Reverse Fee) und der Transaktionsgebühr (Transaction Fee) in zwei verschiedene Klassen (Reverse Fee nutzt die Klasse CalculateReverseFee und Transaction Fee nutzt die Klasse CalculateTransactionFee). Der Vorteil durch diese Unterteilung ist zum einem die Übersichtlichkeit, da die Klassen deutlich kleiner sind und zum anderen die einfachere Umsetzung von Änderungen. Es kann sich deutlich besser im Coding zurechtgefunden und verstanden werden. Des Weiteren führt dieses Prinzip zu einer hohen Kohäsion innerhalb der Klassen, da nur Methoden und Funktionalitäten enthalten sind, die einen starken gemeinsamen Bezug zueinander haben.
- Open-Closed Principle (OCP): Das OCP besagt, dass eine Klasse offen für Erweiterungen sein sollte ohne, dass das Verhalten sich ändern muss. Konkret bedeutet dies, dass Erweiterungen ohne Anpassung des Source Codes oder der Schnittstellen möglich sein sollten. Dieses Prinzip wurde in den Klassen für die Berechnung des Restbetrags und der Transaktionsgebühr (CalculateReverseFee und CalculateTransacitonFee) besonders gut umgesetzt. Anstatt die erlaubte Währung und den erlaubten Transaktionstypen über eine Modifikation abzufragen (If-Else), wurde stattdessen eine Liste mit den erlaubten Währungen bzw. Transaktionstypen gepflegt. Wenn nun eine neue Währung oder neuer Transaktionstyp hinzugefügt werden soll, geht dies ohne Probleme indem die Liste um das jeweilige Objekt ergänzt wird und dieses bei dem Aufruf des Konstruktors übergeben wird. Das OCP wurde für diese beiden Erweiterungen angewendet, da dies aus der Analyse des Anforderungen für zukünftige Funktionalitäten hervorging.
- Liskov Substitution Principle (LSR): Das LSR fordert, dass die Instanz einer vererbten Klasse sich so verhalten muss, dass jemand, der glaubt mit der Oberklasse bzw. Basisklasse zu arbeiten, nicht durch unerwartetes Verhalten überrascht wird, falls es sich doch um eine Instanz der Unterklasse handelt. Dieses Prinzip konnte nicht im Coding untergebracht werden, da lediglich einmal im Projekt Vererbung genutzt wurde. Da bei der vererbten Klasse um eine abstrakte Klasse handelte konnte dieses Prinzip nicht direkt verwendet werden. Um das Verhalten von Objekten genauer zu spezifizieren wurden in diesem Projekt Interfaces gegenüber Vererbungen bevorzugt.
- Interface Segregation Principle (ISR): Das ISR dient dazu zu große Interfaces in mehrere kleine Interfaces aufzuteilen. Da die Interfaces in diesem Projekt generell nicht allzu groß waren, war die Erkennung und Verwendung dieses Prinzip schwieriger als zuvor gedacht. Trotzdem wurde dieses Prinzip bei der Berechnung für die Transaktionsgebühren in der Klasse CalculateTransactionFee verwendet. Die Funktionalität dieser Klasse wurde in zwei verschiedene Interfaces IChargePayment und IChargeTransaction unterteilt, sodass die Interfaces jeweils nur eine Funktionalität spezifizieren. Dadurch konnte das Interface IChargeTransaction auch für in der Klasse CalculateReverseFee verwendet werden ohne, dass die Methode IChargePayment implementiert werden muss. Damit wurde gleichzeitig die Implementierung von Methoden verändert, die nicht benötigt werden (YAGNI-Prinzip).

- Dependency-Inversion-Prinzip (DIP): Das letzte Prinzip, das Bestandteil von SOLID ist, ist das DIP. Es wird auch als Prinzip der Entkoppelung bezeichnet. Die Ziel dieses Prinzip ist es, dass Klassen einer höheren Ebene nicht von Klassen einer niedrigen Ebene abhängen sollen. Stattdessen soll eine Abhängigkeit auf ein Interface genutzt werden und sich eine Referenz auf die konkrete Instanz gegeben lassen. Das Ziel und die Effekte sind hierbei eine bessere Wiederverwendbarkeit und eine Erstellung von klareren Schnittstellen. Ebenso die konkrete Implementierung entkoppelt, sodass die Kopplung sinkt. Ein Beispiel, in dem das DIP verwendet wurde, ist die Klasse CompareFee. Anstatt, dass die Klasse von der konkreten Implementierung CalculateTransactionFee abhängt, wurde eine Referenz zu dem Interface IChargePayment genutzt. Dies ermöglicht, dass bspw. bei einer Erweiterung der Funktionalität der Software zum Vergleich der Restbetrag eines Payments, dieselbe Klasse erneut verwendet werden kann.

```
public class CompareFee implements IFeeComparison{

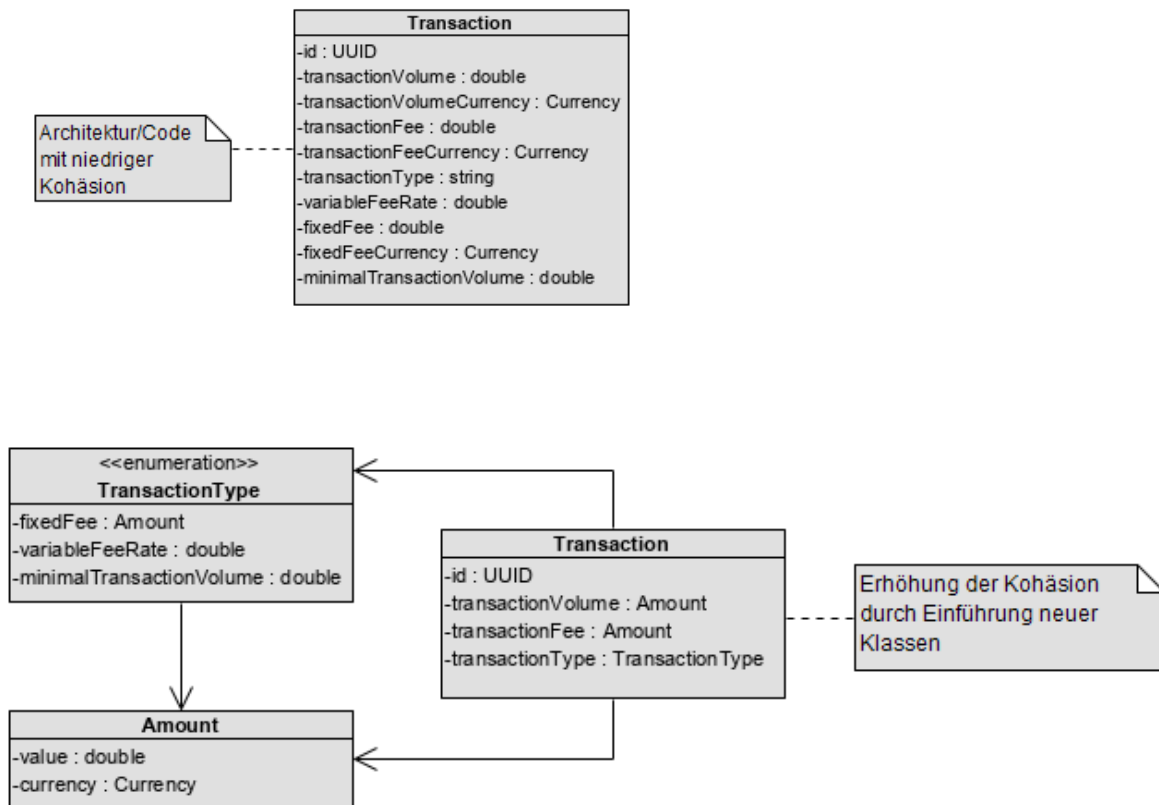
    private IChargePayment calculateTransactionFee;

    public CompareFee(IChargePayment calculateTransactionFee) {
        this.calculateTransactionFee = calculateTransactionFee;
    }
}
```

Ein weiteres wichtiges Programmierprinzip, was häufig verwendet wird, ist GRASP (General Responsibility Assignment Software Patterns). Im Folgendem soll hierzu die Kopplung und Kohäsion des Coings genauer analysiert werden:

- Kopplung: Die Kopplung ist ein Maß für die Abhängigkeit einer Klasse von ihrer Umgebung. Ziel bei der Programmierung ist es eine möglichst geringe bzw. lose Kopplung im Coding zu haben. Dies hat den Vorteil, dass die Software einfacher angepasst und getestet werden kann. Des Weiteren wird die Wiederverwendbarkeit und die Verständlichkeit des Coding durch eine niedrige Kopplung verbessert. Es wurde verschiedene Maßnahmen während des Projektes getroffen, damit die Kopplung der einzelnen Komponenten möglichst gering ist. Eine Maßnahme, die getroffen wurde, ist die Umsetzung des Single Responsibility Principles (Bsp. CalculatReverseFee), welches dafür sorgt, dass jede Klasse auch nur eine Aufgabe übernimmt. Des Weiteren konnte die Kopplung verringert werden, indem für die Referenzen Interfaces anstatt konkrete Klassen verwendet wurden. Die Abhängigkeiten wurden stattdessen mit Hilfe von Dependency Injection aufgelöst (Beispiel CompareFee). Trotzdem hätte die Kopplung auch teilweise verbessert werden können, indem anstatt einigen statischen Methodenaufrufen ein polymorpher Aufruf an ein Interface getätigt worden wäre oder ein Eventsbus-Architekturmuster für die Kommunikation zwischen einzelnen Komponenten genutzt worden wäre. Ebenso ist die Kopplung an Datenformate (CSV-Dateien) im jetzigen Zustand des Projektes ziemlich groß. Durch das Hinzufügen weiterer Formate könnte diese allerdings mit relativ wenig Aufwand gesenkt werden.
- Kohäsion: Die Kohäsion ist ein Maß für den inneren Zusammenhalt einer Klasse. Er gibt die semantische Nähe zwischen Methoden und Attribute einer Klasse an (damit ist die menschliche Einschätzung ein sehr wichtiger Faktor und entscheidend für eine hohe Kohäsion). Ziel ist es, dass der Code eine möglichst hohe Kohäsion besitzt. Die Kohäsion kann durch verschiedene Maßnahmen erhöht werden. Eine ist die Verwendung des Interface Segregation Principles. Wie bereits zuvor beschrieben, wurde dieses Prinzip auch versucht in dem Software-Projekt umzusetzen. Eine weitere

Möglichkeit ist es die Klassen und so zu strukturieren, dass die Kohäsion nach der menschlichen Einschätzung am höchsten ist. Ein Beispiel, welches zu der Erhöhung der Kohäsion in diesem Projekt geführt hat, ist in den unteren beiden UML-Diagrammen dargestellt. Durch die geschickte Einführung neuer Klasse konnte die Kohäsion der verschiedenen Klasse deutlich erhöht werden. Dadurch konnte von den zuvor bereits beschriebenen Vorteilen hoher Kohäsion profitiert werden.



GRASP umfasst neben der Kopplung und der Kohäsion noch weitere Mechanismen und Prinzipien. Diese werden allerdings nicht genauer erläutert.

Ein letztes wichtiges Prinzip, auf welches im folgendem nochmal genauer betrachtet werden soll, ist das DRY (don't repeat yourself)-Prinzip. Das Ziel ist hierbei ist es Redundanz im Sourcecode möglichst zu vermeiden. Ein einfacher Fall, wie dies vermieden werden kann, ist die Code-Duplikate im Quelltext zu vermeiden. Ein Beispiel für die Beachtung des DRY-Prinzips im Projekt ist die Einführung der Klasse `AbstractChargeTransactionServlet`, welches eingeführt wurde, damit die Methode `getTransactionType(String radioButtonValue)`, die von mehreren Klassen genutzt wurde, vererbt werden kann und daher nur einmal implementiert werden muss.

Der Link zum Commit für DRY ist hier zu finden:

<https://github.com/schlueterkai/FeeCalculator/commit/bf3e3d67c17ea524a22720e93ca0ee650351fe26>

Domain Driven Design

Domain Driven Design versucht durch die Verwendung der gleichen Begriffe in der Domäne und im Sourcecode eine klarere Modellierung und eine Reduktion des Übersetzungsaufwand zu ermöglichen.

Analyse der Ubiquitous Language:

Bei der Nutzung der Ubiquitous Language sollen die gleichen Begriffe sowohl in der Domäne als auch im Sourcecode verwendet werden. Die Domäne hängt von dem jeweiligen Fachbereich und der Tätigkeit ab. Im vorliegenden Fall wird die Domäne von Angestellten verwendet, die viel mit Transaktionen und Gebühren arbeiten. Da ich leider keinen Kontakt zu einem englischsprachigen Person hatte, die in diesem Umfeld arbeitet, hatte ich versucht diese Begriffe und die Fachsprache durch Internetrecherche zu finden. Einige Übersetzung vielen dabei sehr leicht, währenddessen andere etwas komplizierter und anspruchsvoller waren. Im Folgendem soll kurz der verwendete Domänenbegriff erläutert werden:

- Amount: Übersetzung von Geldbetrag. Wird im Zusammenhang mit einer Höhe und einer Währung verwendet
 - Value: Höhe des Geldbetrags
 - Currency: verwendete Währung
- Transaction: Übersetzung von Transaction. Paypal selbst nennt einen Bezahlvorgang Transaktion, weshalb dies soweit in Ordnung geht
 - Id: Identifier wird in der IT häufig verwendet. Mittlerweile können Domänenexperten auch etwas mit diesem Begriff anfangen, allerdings gibt es je nach Kreditinstitut und genutzter Plattform unterschiedliche Bezeichnungen für diesen Begriff.
 - Transaction Volume: bezeichnet die Höhe einer Transaktion bzw. das Transaktionsvolumen
 - Transaction Fee: bezeichnet die fällige Transaktionsgebühr
 - Transaction Type: bezeichnet die Art der verwendeten Transaktion
- Payment: Im Nachhinein bin ich nicht wirklich glücklich mit der Auswahl dieses Begriffes, allerdings ist mir aufgrund meiner fehlenden fachlichen Expertise bis heute auch kein besserer Begriff eingefallen. Bezeichnet eine Liste von verschiedenen Transaktionen, welche den Zahlungseingang in einem Unternehmen darstellen
 - Id: siehe oben
 - Transactions: Liste von Transaktionen
 - Transaction Fees: Bezeichnet die Höhe und Währung der Transaktionsgebühren, die auf die Zahlungseingängen gezahlt werden müssen

Es wurde trotzdem versucht während des Projektes den Code so verständlich wie möglich zu schreiben. Ein Beispiel hierzu ist sicherlich die Umsetzung der Berechnung der Transaktionsgebühr in der Klasse CalculateTransactionFee, die der Sichtbarkeitsstufe 2 entspricht. Gleiches gilt für die Klasse CalculateReverseFee.

Analyse und Begründung zu Repositories: Werden meist für die Abstrahierung der Persistenz eingesetzt. Ebenso können sie die Suchen von Objekten erleichtern. Zudem trennen diese die technische Infrastruktur von der Geschäftslogik. Da in der Software noch keine Datenpersistenz notwendig ist bzw. das Abspeichern von Entitäten nach nicht möglich ist, wurden Repositories nicht eingesetzt. Für die Zukunft könnten Repositories zur Verwaltung und Laden der Transaktionen und Payments eingesetzt werden.

Analyse und Begründung zu Aggregates: Aggregates können als Zusammenfassung von Entities und Value Objects aufgefasst werden, die eine Einheit für CRUD-Operationen bieten. Es können so komplexe Objektbeziehungen entkoppelt werden und es bilden sich natürliche Transaktionsgrenzen. Es macht vor allem Sinn Aggregates einzusetzen, wenn eine komplexe Beziehung zwischen den Entitäten und Value Objects vorliegt und diese zum Teil auch bidirektional sind. Da die Beziehungen zwischen den Objekten in der vorliegenden Software nicht allzu komplex sind bietet es sich noch nicht an, ein Aggregat für die Verwaltung zu nutzen. Allerdings kann der Einsatz von Aggregates in Zukunft sinnvoll sein, falls die Software

um weitere Funktionalitäten, wie zum Beispiel der Zuordnung einer Transaktion zu einer Person oder einem Unternehmen ergänzt wird.

Analyse und Begründung zu Entities: Entitäten sind Objekte, die nicht durch ihre Eigenschaften, sondern stattdessen über die ihre Identität definiert werden. Sie zeichnen sie daher daran aus, dass sie über einen natürlichen oder künstlichen Schlüssel verfügen. In dem vorliegendem Software ist sowohl Transaction als auch die Payment eine Entität, da diese jeweils über eine eigene Identität verfügen.

Analyse und Begründung zu Value Objects Dies sind Objekte, die rien durch ihre Eigenschaften definiert werden. Laut der Vorlesung müssen sechs verschiedene Punkte bei der Implementierung von Value Objects beachtet werden. In dem vorliegendem Projekt handelt es sich bei der Klasse Amount um ein Value Object. Dieses erfüllt alle in der Vorlesung genannten Eigenschaften. Es handelt sich um ein Value Object, da es über keine eigene Identität verfügt.

Value Objects implementieren

- 1. Alle Felder sind „blank final“ deklariert
- 2. Nur Konstruktor stellt gültigen Zustand ein
- 3. equals() und hashCode() sind überschrieben
- 4. Keine Setter oder andere Methoden, durch die Felder geändert werden
- 5. Methoden liefern als Rückgabe nur
 - a. Unveränderliche (immutable) Objekte
 - b. Defensive Kopien (beispielsweise bei Collections)
- 6. Klasse ist final deklariert (keine Vererbung)

Refactoring:

Es gibt verschiedene Gründe, warum während eines Projektes Refactoring betrieben werden sollte. Das wichtigsten Punkte sind hierbei, dass der Code sowohl verständlicher als auch wartbarer wird. Des Weiteren können Fehler im Coding während des Refactoring oftmals gefunden werden, die zuvor nicht bemerkt wurden.

Als Grundlagen, um herauszufinden wo es sinnvoll ist, Refactoring anzuwenden, werden sogenannte Code Smells identifiziert. Ein Code Smell ist die Bezeichnung für eine Stelle im Coding, die nach der Meinung des Entwicklers verbessert werden sollte. Typische Gründe für Code Smells sind duplizierter Code oder lange Methoden.

Ein Beispiel, wo direkt zu Beginn des Projektes Refactoring durchgeführt wurde, ist der untenstehende Commit. Es konnte schnell identifiziert werden, dass die Berechnung der Transaktionsgebühr in den einzelnen Modifikationen berechnet wurde. Hierdurch wiederholte sich der Code sehr häufig und die Methode wurde unnötig lang, was zu einer schlechteren Lesbarkeit und Verständnis führte. Des Weitere hätte eine Änderung der Berechnung der Transaktionsgebühr zu extrem vielen Anpassungen im Coding geführt. Aus diesem Gründe hatte ich mich dazu entschlossen, dass hier auf jeden Fall Refactoring durchgeführt werden muss. Es wurden hierzu die Logik zur Berechnung der Transaktionsgebühr in eine eigene Methode ausgelagert (extract method)

Link zum ersten Commit:

<https://github.com/schlueterkai/FeeCalculator/commit/6b0093f3ed53b3449b9ad033c36b6af29f055a41>

Ein weitere Stelle im Quellcode bei, der ein Code Smell identifiziert werden konnte, war bei Einlesen der Csv-Dateien. Das Problem bei der Methode war, dass diese sehr lang war und dadurch sehr unverständlich wurde. Des Weiteren war die Funktionalität verschiedener Code Stellen nur schwer verständlich und es konnten keine Gruppierung, die zusammengehören einfach bestimmt werden. Deshalb hatte ich mich entschieden das Einlesen der Zeilen aus der Csv-Datei, das Erstellen einer Liste von Transaktionen aus einer Liste von String, als auch das Erstellen einer Transaktion aus einer Zeile in eine eigene Methode auszulagern. Dadurch wurde der Code deutlich strukturierter und lesbarer. Des Weiteren ist es hiermit deutlich einfacher Fehler zu bestimmen und neue Funktionalitäten, wie das Einlesen einer neue Spalte aus der Csv-Datei hinzuzufügen.

Link zum zweiten Commit:

<https://github.com/schlueterkai/FeeCalculator/commit/02fd887df4af9e175e838e20e85da636f4c3233c>

Neben die hier beschriebenen Beispielen wurden an vielen weiteren Stellen des Projektes Refactoring betrieben. Zur Übersichtlichkeit wird hierauf allerdings nicht genauer eingegangen.

Entwurfsmuster:

Ich hatte mich entschieden das Erbauer Entwurfsmuster für das Erstellen von Transaktionen mit einzunehmen, da diese Objekte optionale Felder besitzen. Aus diesem Grund gab es zuvor viele verschiedene Arten von Konstruktoren. Des Weiteren wird die Anzahl der optionalen Felder mit zukünftigen Features zu nehmen. So gibt es bereits Pläne in Zukunft bspw. ebenso den Absender und Empfänger aufzunehmen, damit eine detaillierte Analyse betrieben werden kann. Hierbei kommt trotzdem hinzu, dass die Kernfunktionalität (Berechnen der Transaktionsgebühren) weiterhin auch ohne die zusätzlichen und optionalen Informationen möglich ist.

Ein weiter Vorteil der Verwendung des Erbauer Entwurfsmuster ist die gut lesbare und anwenderfreundliche Schnittstellen, die das Verständnis für Einsteiger und Business-User vereinfacht.

Link zum Commit:

<https://github.com/schlueterkai/FeeCalculator/commit/f98e4a5483a16ded2d5c18cfde61a2a66fe191cc>