

Programmentwurf – 3er Gruppe

Cost Tracker

Name: Felix, Florian

Matrikelnummer: 7662024

Name: Kosek, Nils

Matrikelnummer: 8859924

Name: Vorweg, Fabian

Matrikelnummer: 2384350

Abgabedatum: 31.05.2023

Allgemeine Anmerkungen:

- *Gesamt-Punktzahl: 60P (zum Bestehen mit 4,0 werden 30P benötigt)*
- *die Aufgabenbeschreibung (der blaue Text) und die mögliche Punktzahl muss im Dokument erhalten bleiben*
- *es darf nicht auf andere Kapitel als alleiniger Leistungsnachweis verwiesen werden (z.B. in der Form “XY wurde schon in Kapitel 2 behandelt, daher hier keine Ausführung”)*
- *alles muss in UTF-8 codiert sein (Text und Code)*
- *das Dokument muss als PDF abgegeben werden*
- *es gibt keine mündlichen Nebenabreden / Ausnahmen – alles muss so bearbeitet werden, wie es schriftlich gefordert ist*
- *alles muss ins Repository (Code, Ausarbeitung und alles was damit zusammenhängt)*
- *die Beispiele sollten wenn möglich vom aktuellen Stand genommen werden*
 - *finden sich dort keine entsprechenden Beispiele, dürfen auch ältere Commits unter Verweis auf den Commit verwendet werden*
 - *Ausnahme: beim Kapitel “Refactoring” darf von vorne herein aus allen Ständen frei gewählt werden (mit Verweis auf den entsprechenden Commit)*
- *falls verlangte Negativ-Beispiele nicht vorhanden sind, müssen entsprechend mehr Positiv-Beispiele gebracht werden*
 - ***Achtung: werden im Code entsprechende Negativ-Beispiele gefunden, gibt es keine Punkte für die zusätzlichen Positiv-Beispiele sondern 0,5P Abzug für das fehlende Negativ-Beispiel***
 - *Beispiel*
 - *“Nennen Sie jeweils eine Klasse, die das SRP einhält bzw. verletzt.” (2P)*
 - *Antwort: Es gibt keine Klasse, die SRP verletzt, daher hier 2 Klassen, die SRP einhalten: [Klasse 1], [Klasse 2]*
 - *Bewertung: falls im Code tatsächlich keine Klasse das SRP verletzt: 2P ODER falls im Code mind. eine Klasse SRP verletzt: 1P*
- *verlangte Positiv-Beispiele müssen gebracht werden – im Zweifel müssen sie extra für die Lösung der Aufgabe implementiert werden*
- *Code-Beispiel = Code in das Dokument kopieren (inkl. Syntax-Highlighting)*
- *falls Bezug auf den Code genommen wird: entsprechende Code-Teile in das Dokument kopieren (inkl. Syntax-Highlighting)*

- *bei UML-Diagrammen immer die öffentlichen Methoden und Felder angeben – private Methoden/Felder nur angeben, wenn sie zur Klärung beitragen*
- *bei UML-Diagrammen immer unaufgefordert die zusammenspielenden Klassen ergänzen, falls diese Teil der Aufgabe sind*
- *Klassennamen/Variablennamen/etc im Dokument so benennen, wie sie im Code benannt sind (z.B. im Dokument nicht anfangen, englische Klassennamen zu übersetzen)*
- *die Aufgaben sind von vorne herein bekannt und müssen wie gefordert gelöst werden – z.B. ist es keine Lösung zu schreiben, dass es das nicht im Code gibt*
 - *Beispiel 1*
 - *Aufgabe: Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten*
 - *Antwort: Es wurden keine Fake/Mock-Objekte gebraucht.*
 - *Punkte: 0P*
 - *Beispiel 2*
 - *Aufgabe: UML, Beschreibung und Begründung des Einsatzes eines Repositories*
 - *Antwort: Die Applikation enthält kein Repository*
 - *Punkte*
 - *falls (was quasi nie vorkommt) die Fachlichkeit tatsächlich kein Repository hergibt: volle Punktzahl*
 - *falls die Fachlichkeit in irgendeiner Form ein Repository hergibt (auch wenn es nicht implementiert wurde): 0P*
 - *Beispiel 3*
 - *Aufgabe: UML von 2 implementierte unterschiedliche Entwurfsmuster aus der Vorlesung*
 - *Antwort: es wurden keine Entwurfsmuster gebraucht/implementiert*
 - *Punkt: 0P*

Kapitel 1: Einführung (4P)

Übersicht über die Applikation (1P)

[Was macht die Applikation? Wie funktioniert sie? Welches Problem löst sie/welchen Zweck hat sie?]

Die Applikation „CostTracker“ kann zur Nachverfolgung von Einkäufen und Ausgaben, die ein Nutzer tätigt, verwendet werden. Dafür muss der Nutzer seine Ausgaben und Einkäufe in der Applikation hinterlegen. Diese Ausgaben und Einkäufe werden dann gespeichert und können anschließend ausgewertet werden. Dabei können die Ausgaben und Einkäufe Kategorien und Firmen zugeordnet werden, die der Nutzer selbst anlegen kann. Für die Auswertung können anschließend Übersichten erstellt werden. Dabei können Übersichten für die Ausgaben pro Tag, pro Woche, pro Monat, pro Firma und pro Kategorie erstellt werden. Die Übersicht pro Kategorie und pro Firma kann dabei durch eine Zeitspanne beschränkt werden. Diese Übersichten können einmal in der Applikation angezeigt werden oder in verschiedene Dateiformate exportiert werden. Das Anlegen der Einkäufe oder Ausgaben, Firmen oder Kategorien kann einerseits über die Konsole erfolgen. Andererseits kann auch ein Import über Dateiformate durchgeführt werden.

Starten der Applikation (1P)

[Wie startet man die Applikation? Was für Voraussetzungen werden benötigt? Schritt-für-Schritt-Anleitung]

Die Applikation wurde in Java entwickelt, im Detail mit dem JDK17. Gestartet werden kann die Applikation mittels der gängigen IDE's (Bspw. IntelliJ oder Eclipse), dafür muss die Main() in dem Verzeichnis `costtracker/src/main/java/` ausgeführt werden.

Alternativ geht es auch mit Docker, wie unten beschrieben.

Zum Starten der Applikation muss zunächst sichergestellt sein, dass Docker auf dem System installiert ist, da dort die Applikation ausgeführt werden soll. Danach muss ein Terminal geöffnet werden und zunächst ein Dockerimage gebaut werden mit dem Command `„docker build -t costtracker-image .“`. Der Command `„docker run -it costtracker-image“` muss als nächstes ausgeführt werden, um die Applikation zu starten. Ist diese gestartet wird durch Informationen im Dialog durch diesen geführt. Darüber hinaus ist zu beachten, dass bei erforderlichen Eingaben, welche nicht ausgefüllt werden, die Eingabe geschlossen wird und im Dialog zurück gegangen. Einzige Ausnahme besteht bei der Bearbeitung von Käufen, Firmen oder Kategorien, bei welchen eine leere Eingabe dazu führt, dass die bestehende Information übernommen wird.

Technischer Überblick (2P)

[Nennung und Erläuterung der Technologien (z.B. Java, MySQL, ...), jeweils Begründung für den Einsatz der Technologien]

Java

Als Programmiersprache wurde Java gewählt, da diese plattformunabhängig ist. Java ist zudem objektorientiert und durfte als Programmiersprache für dieses Projekt verwendet werden. Des Weiteren wurde die Programmiersprache im Rahme der Vorlesung „Programmieren“ behandelt, wodurch weitreichende Kenntnisse in dieser Programmiersprache im Team vorhanden waren.

H2

Als Datenbank wurde sich für die H2 Datenbank entschieden. Dabei spielte der Fakt, dass es eine Open Source Software ist, eine große Rolle. Des Weiteren kann diese Datenbank in Memory betrieben werden. Dies wollen wir in unserem Projekt benutzen.

Maven

Maven ist ein Build-Werkzeug, welches verwendet wurde, um Dependency's zuverlässig und einfach zu managen. Maven wurden genutzt, um eine lauffähige Applikation in Form eines JAR-Files zu bauen.

JSON Library

Die JSON Library org.json wurde verwendet, um bequemer Daten über die API auszutauschen.

JUnit

Um die entwickelten Funktionen zu testen, wird auf JUnit gesetzt. Junit ist das Unit Testing Tool für Java und unterstützte mit seinen einfachen Funktion bei der Entwicklung von Unit Tests.

Kapitel 2: Clean Architecture (8P)

Was ist Clean Architecture? (1P)

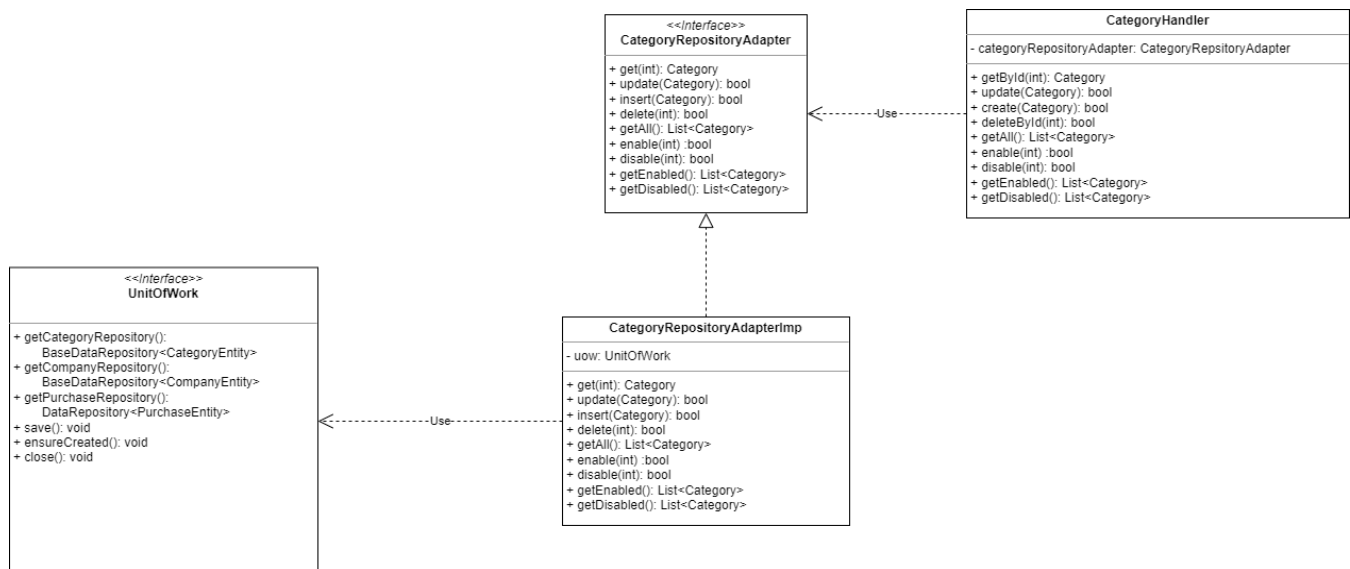
[allgemeine Beschreibung der Clean Architecture in eigenen Worten]

Clean Architecture ist eine Sammlung aus Design Entscheidungen, die während des Implementierens von Software getroffen werden. Dabei zielen die Entscheidungen darauf ab, die Software so langlebig wie möglich zu machen.

Analyse der Dependency Rule (3P)

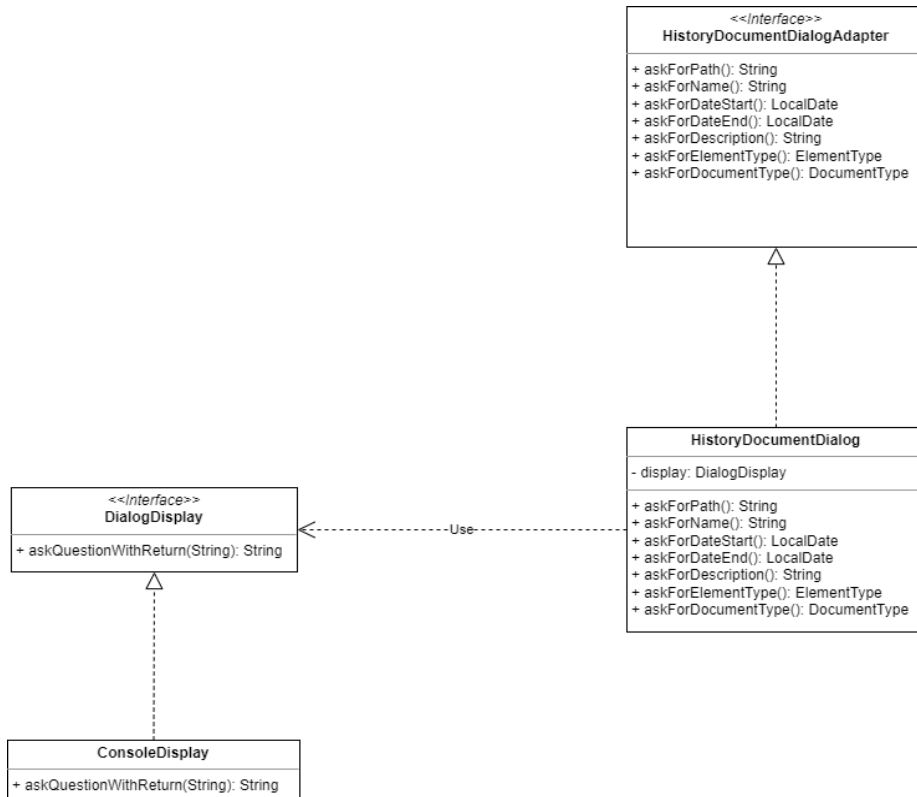
[1 Klasse, die die Dependency Rule einhält und 1 Klasse, die die Dependency Rule verletzt; jeweils UML (mind. die betreffende Klasse inkl. der Klassen, die von ihr abhängen bzw. von der sie abhängt) und Analyse der Abhängigkeiten in beide Richtungen (d.h., von wem hängt die Klasse ab und wer hängt von der Klasse ab) in Bezug auf die Dependency Rule]

Positiv-Beispiel 1: Dependency Rule



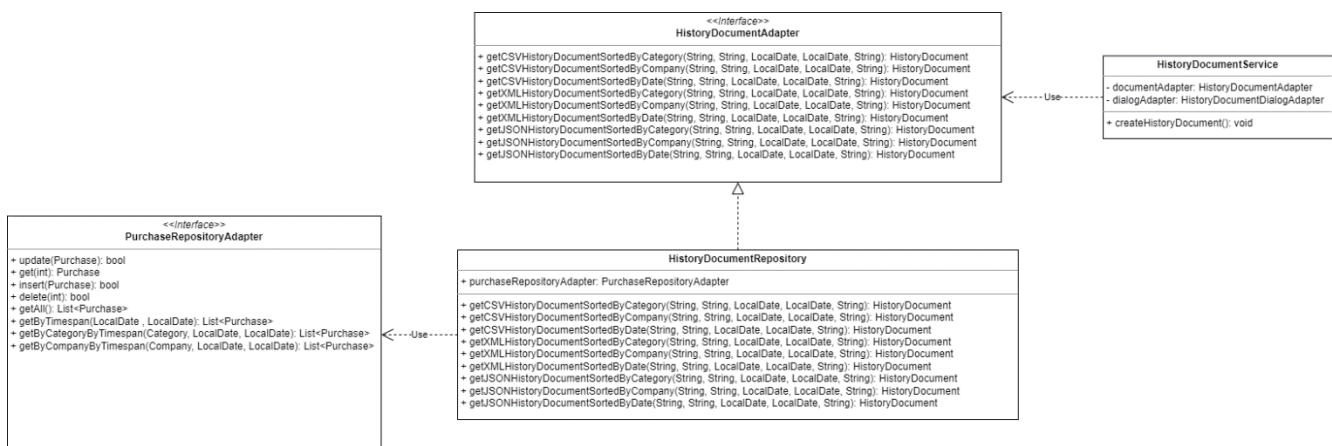
Das „CategoryRepositoryAdapter“ - Interface wird von der Klasse „CategoryRepositoryAdapterImp“ implementiert. Das Interface liegt dabei in der Applikationsschicht, während die Klasse in der Adapterschicht ist. Die Klasse benutzt das Interface „UnitOfWork“, das in der Adapterschicht liegt. Sie hängt damit von inneren Schichten ab und nutzt zur Laufzeit Klassen oder Interfaces der gleichen Schicht, die allerdings von äußeren Schichten implementiert oder erweitert werden können.

Positiv-Beispiel 2: Dependency Rule



Die Klasse „ConsoleDisplay“ implementiert das Interface „DialogDisplay“. Die Klasse ist ein Plugin, das die Darstellung per Console erfüllt. Das „DialogDisplay“ Interface liegt in der Adapterschicht. Es wird von der „HistoryDocumentDialog“ Klasse benutzt, die das „HistoryDocumentDialogAdapter“ – Interface implementiert. Die Klasse implementiert daher ein Interface aus einer tieferen Schicht. Sie wird zur Laufzeit als Implementierung des Interfaces in der Adapterschicht verwendet.

Positiv-Beispiel 3: Dependency Rule



Die Klasse „HistoryDocumentRepository“ implementiert das Interface „HistoryDocumentAdapter“. Die Klasse befindet sich auf der Adapterschicht und implementiert das Interface, das auf der Applikationsschicht liegt. Es benutzt dabei den „PurchaseRepositoryAdapter“, der auch auf der

Applikationsschicht liegt. Der HistoryDocumentService benutzt das Interface „HistoryDocumentAdapter“. Dadurch hängt er nicht von der Klasse, sondern vom Interface ab.

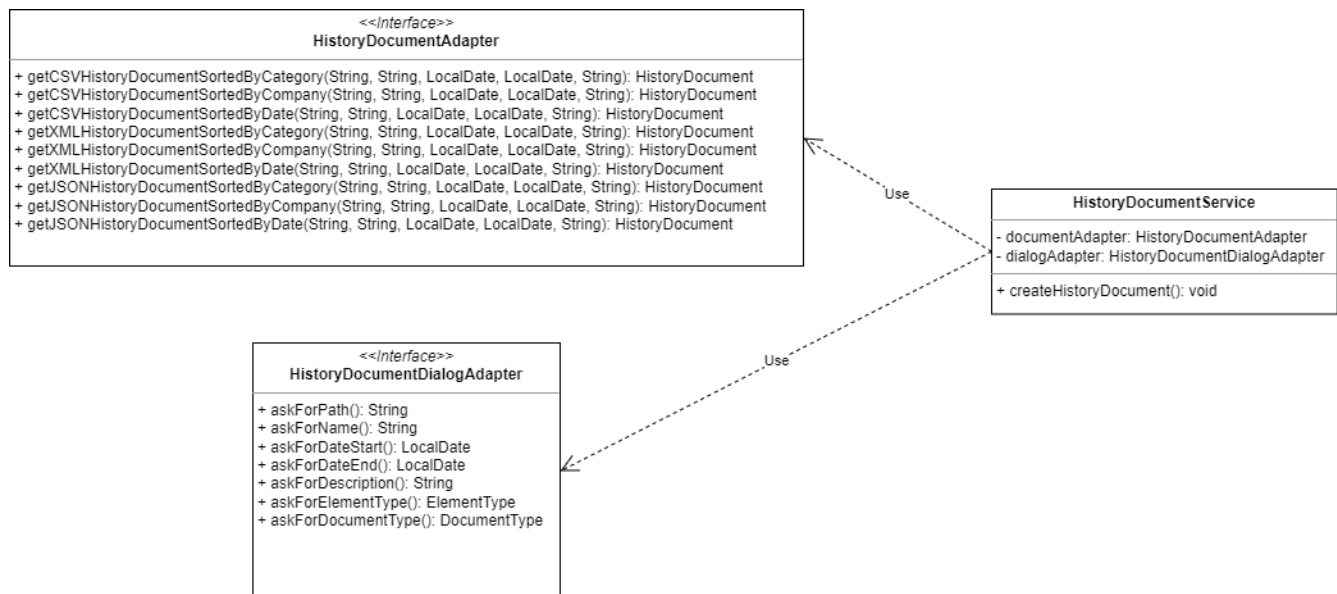
Negativ-Beispiel: Dependency Rule

Alle Abhängigkeiten zeigen nach innen.

Analyse der Schichten (4P)

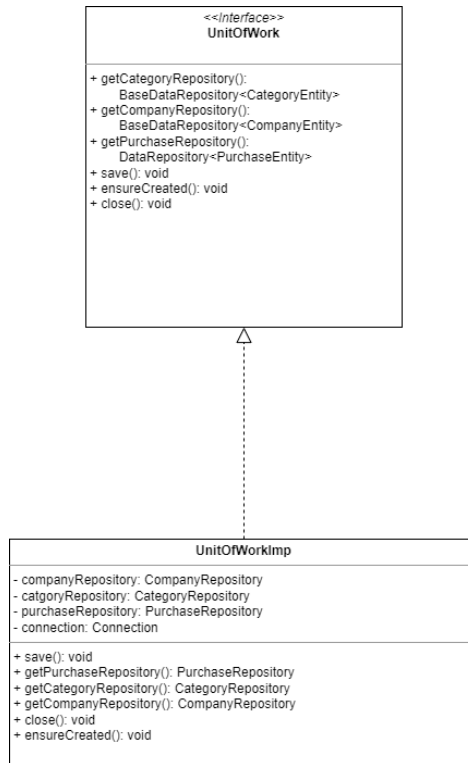
[jeweils 1 Klasse zu 2 unterschiedlichen Schichten der Clean-Architecture: jeweils UML (mind. betreffende Klasse und ggf. auch zusammenspielenden Klassen), Beschreibung der Aufgabe, Einordnung mit Begründung in die Clean-Architecture]

Schicht: Application



Die Klasse „HistoryDocumentService“ ist für die Erstellung eines Dokuments zuständig, bei dem eine Historie dargestellt wird. Sie behandelt dabei den UseCase „Erstellung eines historischen Dokuments“. Dadurch liegt sie in der Applikationsschicht. Die Klasse verwendet dabei die Interfaces „HistoryDocumentAdapter“ und „HistoryDocumentDialogAdapter“. Diese sind in der Applikationsschicht definiert und werden in der Adapterschicht ausimplementiert.

Schicht: Plugin



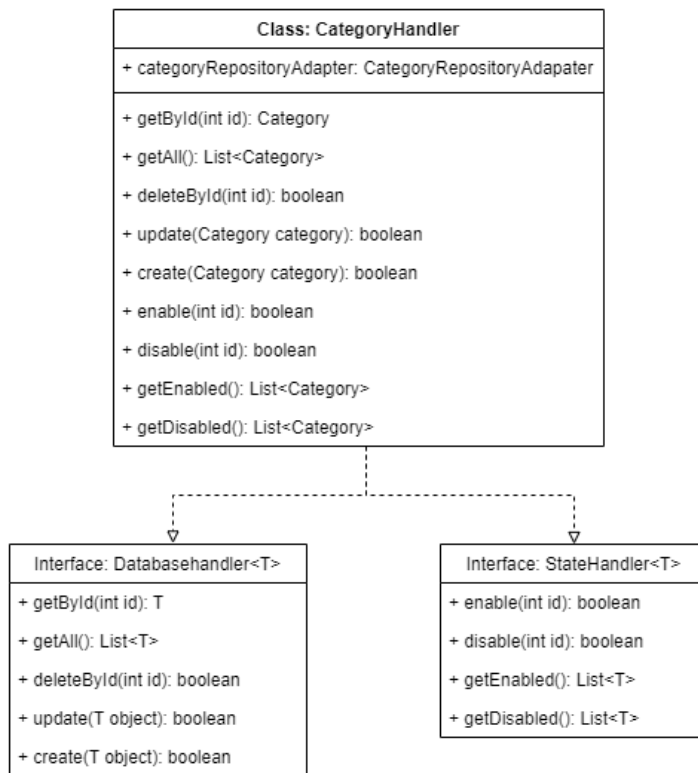
Die Klasse „UnitOfWorkImp“ ist für die Einhaltung des UnitOfWork patterns zuständig. Hier überwacht es die Einhaltung bei der verwendeten Datenbank h2. Daher ist sie in der Plugin Schicht, da sie zum Datenbank Framework gehört. In der Adapterschicht wird ein Interface „UnitOfWork“ angeboten, das diese Klasse implementiert. So kann ohne Abhängigkeit zur Pluginschicht aus der Adapterschicht persistiert werden.

Kapitel 3: SOLID (8P)

Analyse SRP (3P)

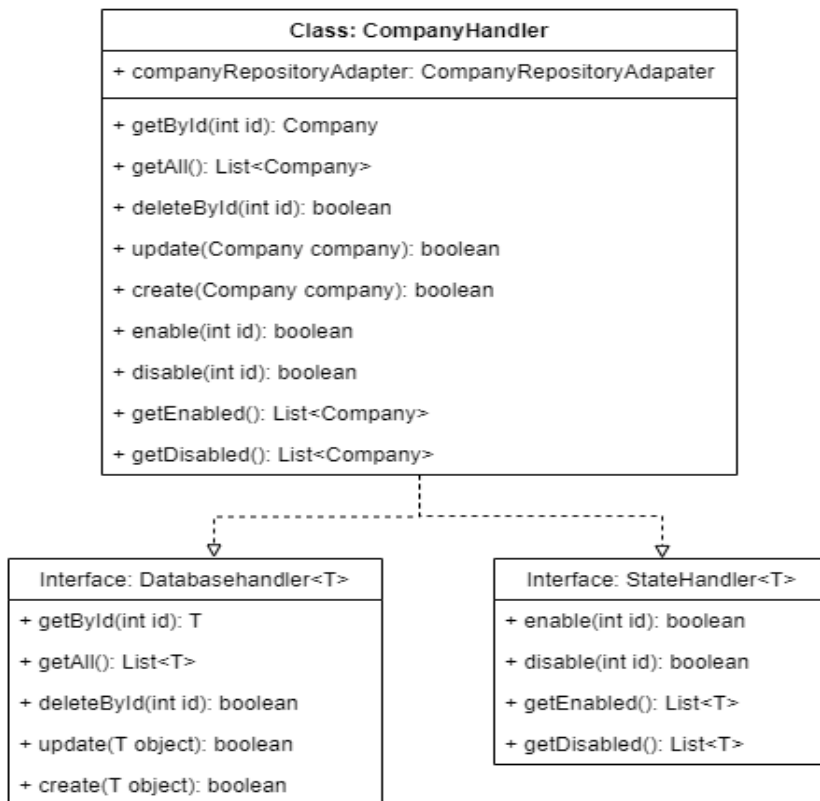
[zwei Klassen als positives Beispiel und eine Klasse als negatives Beispiel für SRP; jeweils UML und Beschreibung der Aufgabe bzw. der Aufgaben und möglicher Lösungsweg des Negativ-Beispiels (inkl. UML)]

Positiv-Beispiel 1



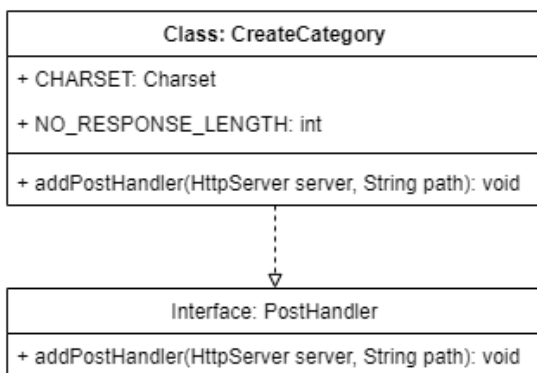
Die Klasse **CategoryHandler** kümmert sich um alles, was zum Verwalten/speichern einer Kategorie benötigt wird. Dafür werden die beiden Interfaces **DatabaseHandler** und **StateHandler** implementiert. Erfüllt ist das SRP dadurch, dass sich hier nur um die Kategorie gekümmert wird und die Implementierung dieser als abgeschlossen gilt.

Positiv-Beispiel 2



Die Klasse **CompanyHandler** kümmert sich um alles, was zum Verwalten/speichern einer Company benötigt wird. Dafür werden die beiden Interfaces **DatabaseHandler** und **StateHandler** implementiert. Erfüllt ist das SRP dadurch, dass sich hier nur um die Kategorie gekümmert wird und die Implementierung dieser als abgeschlossen gilt.

Negativ-Beispiel

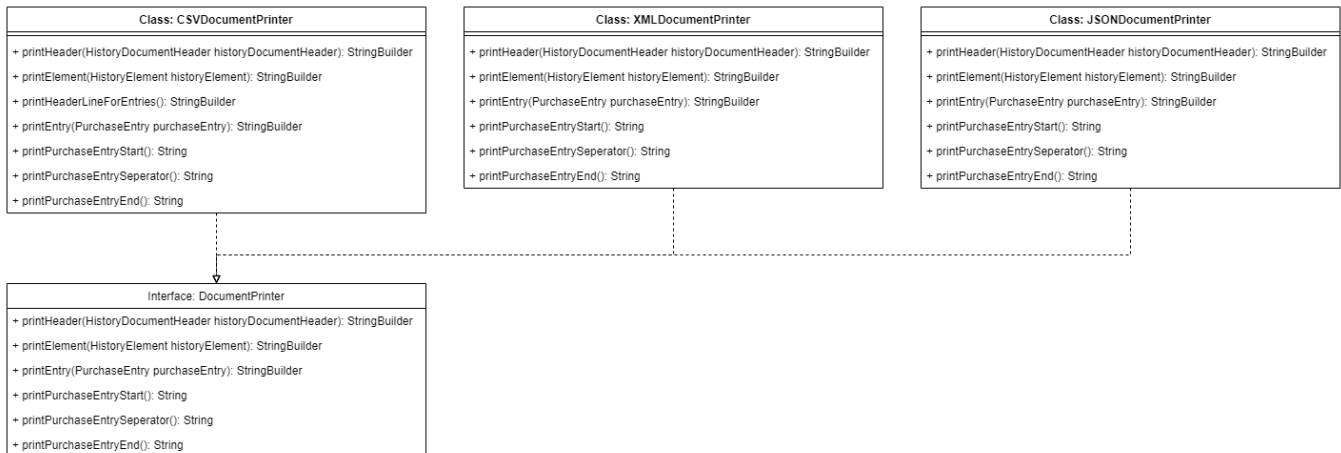


In diesem Negativbeispiel wird die Klasse **CreateCategory** behandelt. Diese kümmert sich um das Erstellen einer neuen Kategorie über die API. Innerhalb der Methode wird der **ResponseHeader** und **ResponseBody** zusammengesetzt, der **RequestBody** sowohl der **RequestHeader** ausgewertet und geprüft, ob der aufgerufene Pfad valide ist. Dadurch ist das SRP hier nicht gegeben und könnte durch ein aufsplitten in einzelne Funktion gelöst werden.

Analyse OCP (3P)

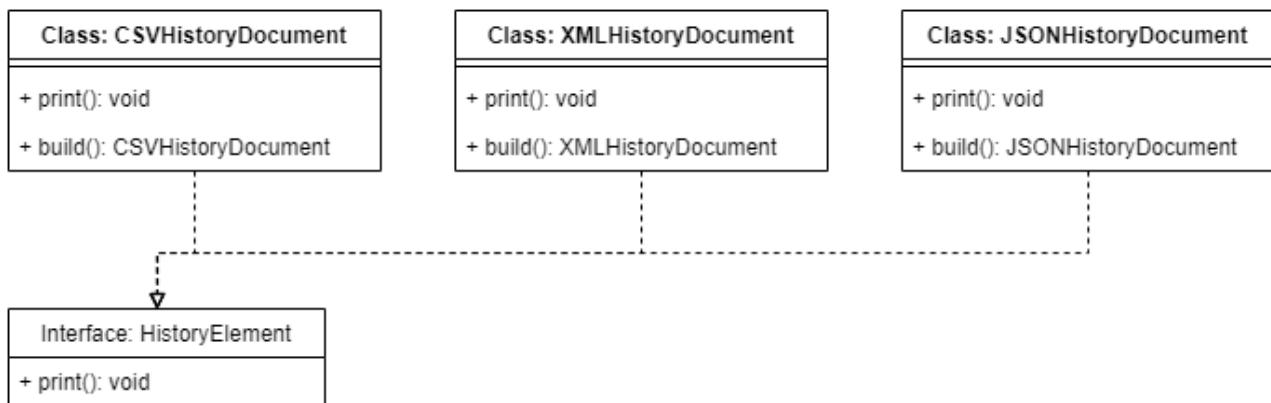
[zwei Klassen als positives Beispiel und eine Klasse als negatives Beispiel für OCP; jeweils UML und Analyse mit Begründung, warum das OCP erfüllt/nicht erfüllt wurde – falls erfüllt: warum hier sinnvoll/welches Problem gab es? Falls nicht erfüllt: wie könnte man es lösen (inkl. UML)?]

Positiv-Beispiel 1



Hier wird das OCP erfüllt, durch die Implementierung des DocumentPrinter Interface, welches generisch gehalten ist, um neue DocumentPrinter zu implementieren. Wie in dem Diagramm zu erkennen, sind drei DocumentPrinter implementiert worden. Nämlich der CSV-, XML- und JSONDocumentPrinter. Sinnvoll ist das an dieser Stelle, damit schnell und beliebig neue DocumentPrinter implementiert werden können, umso den Wünschen der Nutzer zu entsprechen.

Positiv-Beispiel 2



In diesem Beispiel wird OCP erfüllt, durch die Nutzung des Interface HistoryElement. Dieses wird genutzt um beliebige HistoryDocument (sei es CSV-, XML- oder JSONHistoryDocument) umzusetzen. Ähnlich zu dem vorherigen Beispiel lassen sich damit neue Klassen erstellen (Offen für Erweiterung), aber keine Modifikation an den Klassen durchführen (geschlossen für Modifikation). Alle Modifikation der Funktionen, müssen durch das Interface festgelegt werden. Nötig und sinnvoll hier OCP anzuwenden ist es, da die HistoryDocument zusammen mit DocumentPrinter arbeiten und dadurch für beide OCP erfüllt sein muss.

Negativ-Beispiel

Class: Dialogue
+ editor: Editor + adder: Adder + deactivator: Deactivator + activator: Activator
+ talk(): void + interaction1(): boolean + editCompanyOrCategory(): void + historyAction(): void + timespanAction(History history): void

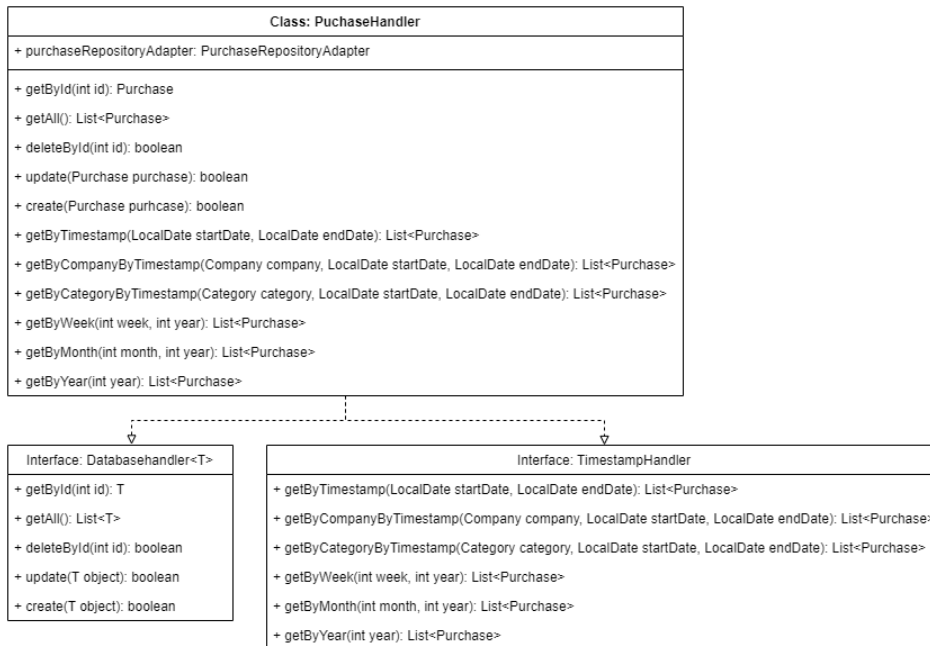
Bei diesem Beispiel ist OCP nicht zu finden, da wenn neue Interaktionen implementiert werden sollen, muss die ganze Klasse und die darauf aufbauenden Klassen angepasst werden. Dadurch entsteht ein hoher Workload. Gelöst werden kann das, indem man neue Interfaces erstellt, welche die einzelnen Interaktionen/Dialoge beinhalten, um unabhängiger zu werden.

Analyse [LSP/ISP] (1P)

[jeweils eine Klasse als positives und negatives Beispiel für entweder LSP oder ISP; jeweils UML und Begründung, warum hier das Prinzip erfüllt/nicht erfüllt wird; beim Negativ-Beispiel UML einer möglichen Lösung hinzufügen]

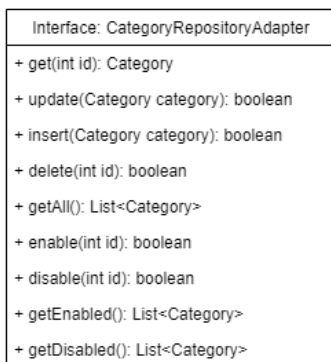
[Anm.: es darf nur ein Prinzip ausgewählt werden; es darf NICHT z.B. ein positives Beispiel für LSP und ein negatives Beispiel für ISP genommen werden]

Positiv-Beispiel



Hier ist ISP erfüllt da durch das Nutzen der zwei Interfaces wissen getrennt bzw. in logische Bereiche gruppiert wurde. Die Klasse kümmert sich in diesem Beispiel um das Verwalten/managen der Purchases.

Negativ-Beispiel

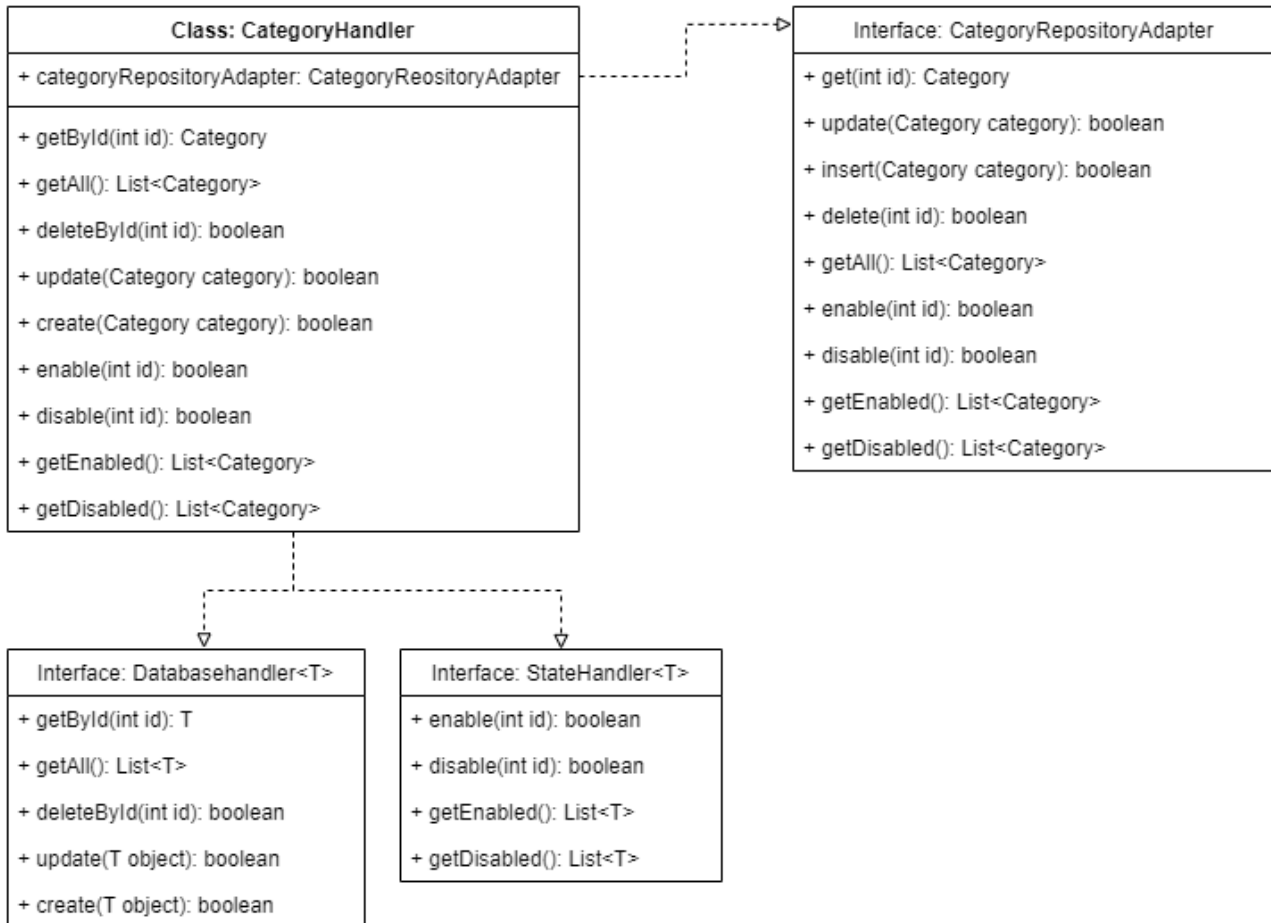


Hier ist das ISP nicht erfüllt/nicht gegeben, da alle Funktionen in einem Interface zusammengefasst sind. Sinnvoll wäre es hier diese aufzusplitten in verschiedene Interfaces, um eine besser Struktur zu erhalten und so spezifischer zu werden.

Analyse [DIP] (1P)

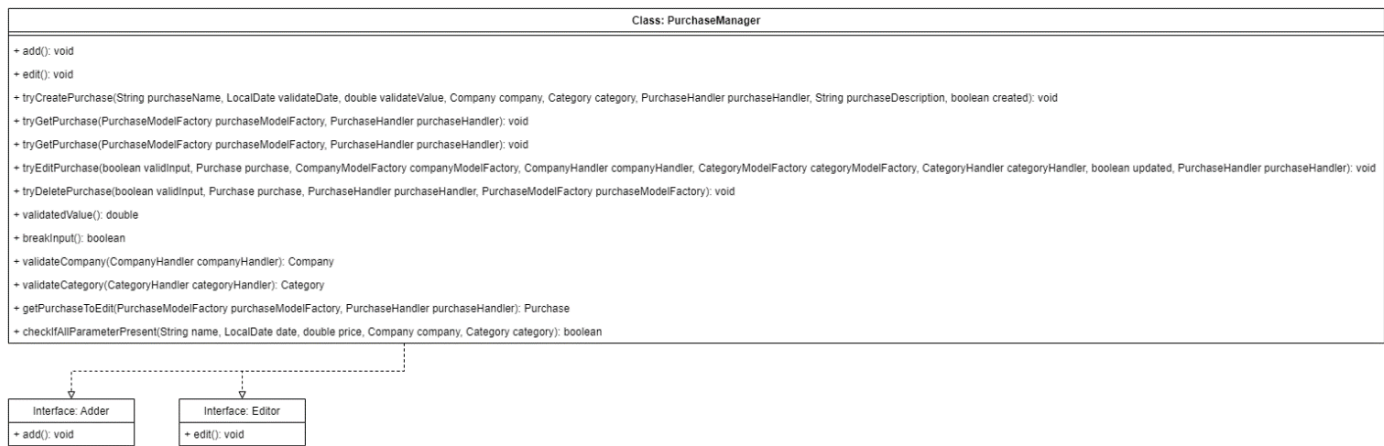
[jeweils eine Klasse als positives und negatives Beispiel für DIP; jeweils UML und Begründung, warum hier das Prinzip erfüllt/nicht erfüllt wird; beim Negativ-Beispiel UML einer möglichen Lösung hinzufügen]

Positiv-Beispiel



In diesem Beispiel wird DIP erfüllt, da hier mit dem DependencyContainer gearbeitet wird. Dieser stellt sicher, dass die Dependencies richtig geladen und genutzt werden können. Hierbei kann ein beliebiger RepositoryAdapter verwendet werden.

Negativ-Beispiel

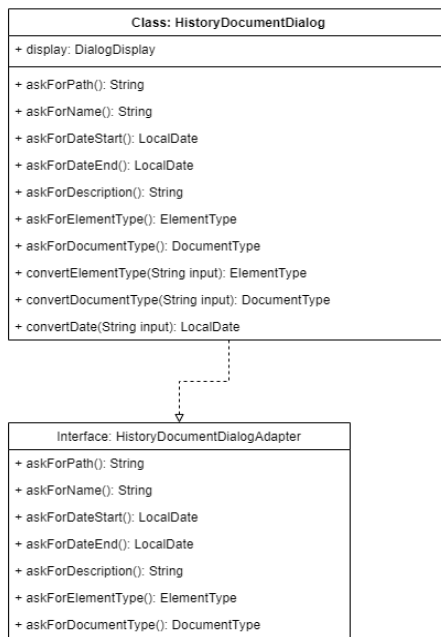


Hier wird das DIP nicht erfüllt, da man sich innerhalb der Funktionen `add()` und `edit()` abhängig von dem `CompanyHandler`, dem `CategoryHandler` und dem `PurchaseHandler` macht. Problematisch ist das, da man auf diese baut, anstelle auf den `DependencyContainer` der für das `DependencyInjection` gemacht ist.

Kapitel 4: Weitere Prinzipien (8P)

Analyse GRASP: Geringe Kopplung (3P)

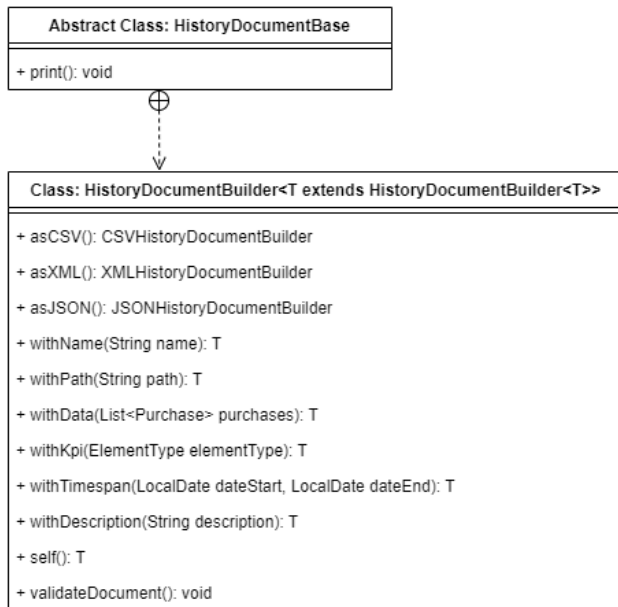
[eine bis jetzt noch nicht behandelte Klasse als positives Beispiel geringer Kopplung; UML mit zusammenspielenden Klassen, Aufgabenbeschreibung der Klasse und Begründung, warum hier eine geringe Kopplung vorliegt]



In diesem Beispiel liegt geringe Kopplung vor, da hier neben dem Einsatz des DependencyContainer auch ein Interface genutzt wird. Dadurch entsteht eine einfache Anpassbarkeit bei Bedarf und auch eine bessere Testbarkeit. Die Lesbarkeit wird hierbei durch das Verwenden des DialogDisplay verbessert.

Analyse GRASP: Polymorphismus (1,5P)

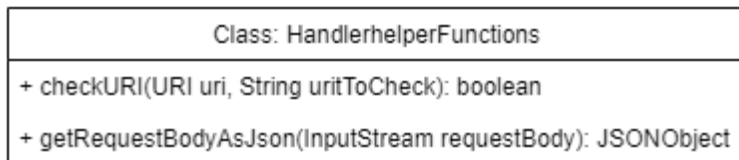
[eine Klasse als positives Beispiel entweder von Polymorphismus oder von Pure Fabrication; UML Diagramm und Begründung, warum es hier zum Einsatz kommt]



Hier liegt Polymorphismus vor, da vorher in dem `HistoryDocumentBuilder` über mehrere `if`-Statements auf den `DocumentType` (CSV, XML oder JSON) geprüft werden musste. Durch das Anpassen der Klasse können nun die einzelnen Files erzeugt werden.

Analyse GRASP: Pure Fabrication (1,5P)

[eine Klasse als positives Beispiel entweder von Polymorphismus oder von Pure Fabrication; UML Diagramm und Begründung, warum es hier zum Einsatz kommt]



Die Klasse HandlerHelperFunctions weißt kein Domainwissen auf und ist somit als allein stehende Klasse zu betrachten die nützliche Hilfsfunktionen bereitstellt. In diesem Beispiel sind das die Funktionen checkURI (die eine gegebene URI überprüfen) und getRequestBodyAsJson (die einen InputStream in ein JSONObject parsen).

DRY (2P)

[ein Commit angeben, bei dem duplizierter Code/duplizierte Logik aufgelöst wurde; Code-Beispiele (vorher/nachher) einfügen; begründen und Auswirkung beschreiben – ggf. UML zum Verständnis ergänzen]

Konnte nicht mehr bearbeitet werden, es ist aber duplizierte Code zu finden, der ersetzt werden kann durch bessere Abstraktion der Klassen bzw. Methoden.

Kapitel 5: Unit Tests (8P)

10 Unit Tests (2P)

[Nennung von 10 Unit-Tests und Beschreibung, was getestet wird]

Unit Test	Beschreibung
CategoryUnitTest#testFromEntity	Dieser Unit Test testet, ob sich ein CategoryEntity zu einem Category BuisnessObject mappen lässt.
CategoryUnitTest#testToEntity	Dieser Unit Test testet, ob sich ein Category BuisnessObject in ein CategoryEntity mappen lässt.
CompanyUnitTest#testFromEntity	Dieser Unit Test testet, ob sich ein CompanyEntity zu einem Company BuisnessObject mappen lässt.
CompanyUnitTest#testToEntity	Dieser Unit Test testet, ob sich ein Company BuisnessObject in ein CompanyEntity mappen lässt.
PurchaseUnitTest#testFromEntity	Dieser Unit Test testet, ob sich ein PurchaseEntity zu einem Purchase BuisnessObject mappen lässt.
PurchaseUnitTest#testToEntity	Dieser Unit Test testet, ob sich ein Purchase BuisnessObject in ein PurchaseEntity mappen lässt.
UnitOfWorkUnitTest#TestGetPurchaseRepository	Dieser Unit Test testet, ob sich das PurchaseRepository über die UnitOfWork ansprechen lässt.
UnitOfWorkUnitTest#TestGetCompanyRepository	Dieser Unit Test testet, ob sich das CompanyRepository über die UnitOfWork ansprechen lässt.
UnitOfWorkUnitTest#TestGetCategoryRepository	Dieser Unit Test testet, ob sich das CategoryRepository über die UnitOfWork ansprechen lässt.
DependencyFactoryUnitTest#testRegisterScoped	Dieser Unit Test testet, ob sich mithilfe einem Dependency Container ein registerScoped des PurchaseRepositoryAdapter erstellen lässt.























































ATRIP: Automatic (1P)

[Begründung/Erläuterung, wie 'Automatic' realisiert wurde]

Automatic wurde in diesem Projekt mithilfe der GitHub Actions realisiert. Genauer eine CI-Pipeline, die bei jedem Pull-Request und bei jedem Merge in den Master getriggert wird und somit die Unit Tests ausführt. Wichtig ist das, damit nicht vergessen wird die Tests auszuführen beim Entwickeln von neuen Features. Somit können Bugs und Fehler frühzeitig identifiziert werden.

ATRIP: Thorough (1P)

[Code Coverage im Projekt analysieren und begründen]

▼  src/main/java		32.7 %	4,171	8,565	12,736
>  costtracker.plugin.ui		0.1 %	3	3,138	3,141
>  costtracker.plugin.api.routes.purchase		0.0 %	0	1,122	1,122
>  costtracker.plugin.api.routes.company		0.0 %	0	914	914
>  costtracker.plugin.api.routes.category		0.0 %	0	902	902
>  costtracker.adapter.repositoryadapters		10.9 %	72	587	659
>  costtracker.application.handlers		0.0 %	0	339	339
>  costtracker.adapter.document		25.7 %	105	303	408
>  costtracker.domain.businessobjects		46.1 %	245	286	531
>  costtracker.plugin.api		7.7 %	23	275	298
>  costtracker.application.in		16.7 %	45	224	269
>  costtracker.plugin.api.enums		0.0 %	0	99	99
>  (default package)		0.0 %	0	83	83
>  costtracker.plugin.db.repositories		92.2 %	930	79	1,009
>  costtracker.adapter.entities		79.7 %	236	60	296
>  costtracker.application.out		90.6 %	326	34	360
>  costtracker.plugin.db.unitofwork		64.9 %	63	34	97
>  costtracker.domain.in		94.6 %	524	30	554
>  costtracker.application.converter		76.9 %	90	27	117
>  costtracker.adapter.mappers		94.7 %	107	6	113
>  costtracker.plugin.api.helper		88.5 %	46	6	52
>  costtracker.domain.dependencyinjection		93.0 %	66	5	71
>  costtracker.domain.in.csv		98.9 %	344	4	348
>  costtracker.domain.in.json		99.2 %	497	4	501
>  costtracker.domain.in.xml		99.0 %	412	4	416
>  costtracker.domain.in.enums		100.0 %	21	0	21
>  costtracker.plugin.console.dialog		100.0 %	16	0	16

Die Code Coverage beträgt 32%. Die Api und die Ui wurde durch Integrationstests getestet. Daher fehlen hier automatisierte Tests. Dadurch leidet die Code Coverage in den PlugIns. In der Domäne wurden die Klassen mehr getestet, da sie der wichtigste Bestandteil sind. Die Business Objekte haben dabei eine eher schlechtere Coverage, da sie Getter und Setter Methoden haben, die nicht verwendet wurden. Des Weiteren wurden im „Builder“ Pattern der Business Objects nicht auf alle Fehlerfälle bei der Objekterzeugung geprüft.

ATRIP: Professional (1P)

[1 positives Beispiel zu 'Professional'; Code-Beispiel, Analyse und Begründung, was professionell ist]

```
package costtracker.ut.db;

import java.sql.Connection;
import java.sql.Date;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class DatabaseTestHelper {

    private Connection connection;

    public DatabaseTestHelper(Connection connection) {
        this.connection = connection;
    }

    int createCompany(String name, String location) throws SQLException {
        int id = 0;
        String sql = "Insert into company" + "(name, location, isenabled)" + "values"
            "(?, ?, true)";
        PreparedStatement stmt = this.connection.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
        stmt.setString(1, name);
        stmt.setString(2, location);
        stmt.executeUpdate();
        ResultSet generatedKeys = stmt.getGeneratedKeys();
        if(generatedKeys.next()) {
            id = generatedKeys.getInt(1);
        }
        return id;
    }

    int createCategory(String name) throws SQLException {
        int id = 0;
        String sql = "Insert into category" + "(name, isenabled)" + "values"
            "(?,true)";
        PreparedStatement stmt = this.connection.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
        stmt.setString(1, name);
        stmt.executeUpdate();
        ResultSet generatedKeys = stmt.getGeneratedKeys();
        if(generatedKeys.next()) {
            id = generatedKeys.getInt(1);
        }
        return id;
    }

    int createPurchase(String name, String description, Date date, double price, int categoryid, int companyid) throws SQLException {
        int id = 0;
        String sql = "Insert into purchase" + "(name, description, date, price, category, company)" + "values (?, ?, ?, ?, ?, ?)";
        PreparedStatement stmt = this.connection.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
    }
```

```

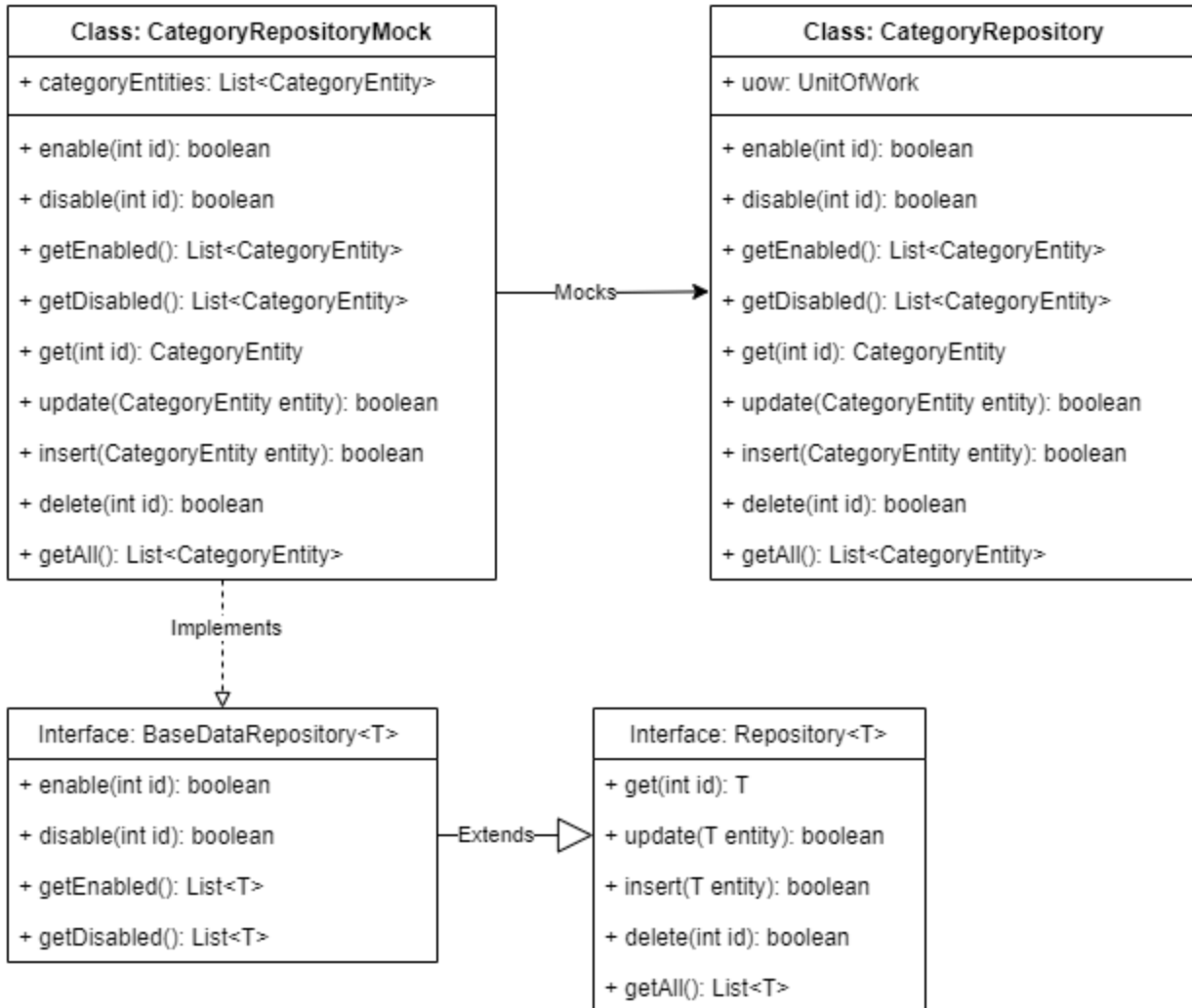
        stmt.setString(1, name);
        stmt.setString(2, description);
        stmt.setDate(3, date);
        stmt.setDouble(4, price);
        stmt.setLong(5, categoryid);
        stmt.setLong(6, companyid);
        stmt.executeUpdate();
        ResultSet generatedKeys = stmt.getGeneratedKeys();
        if(generatedKeys.next()) {
            id = generatedKeys.getInt(1);
        }
        return id;
    }
}

```

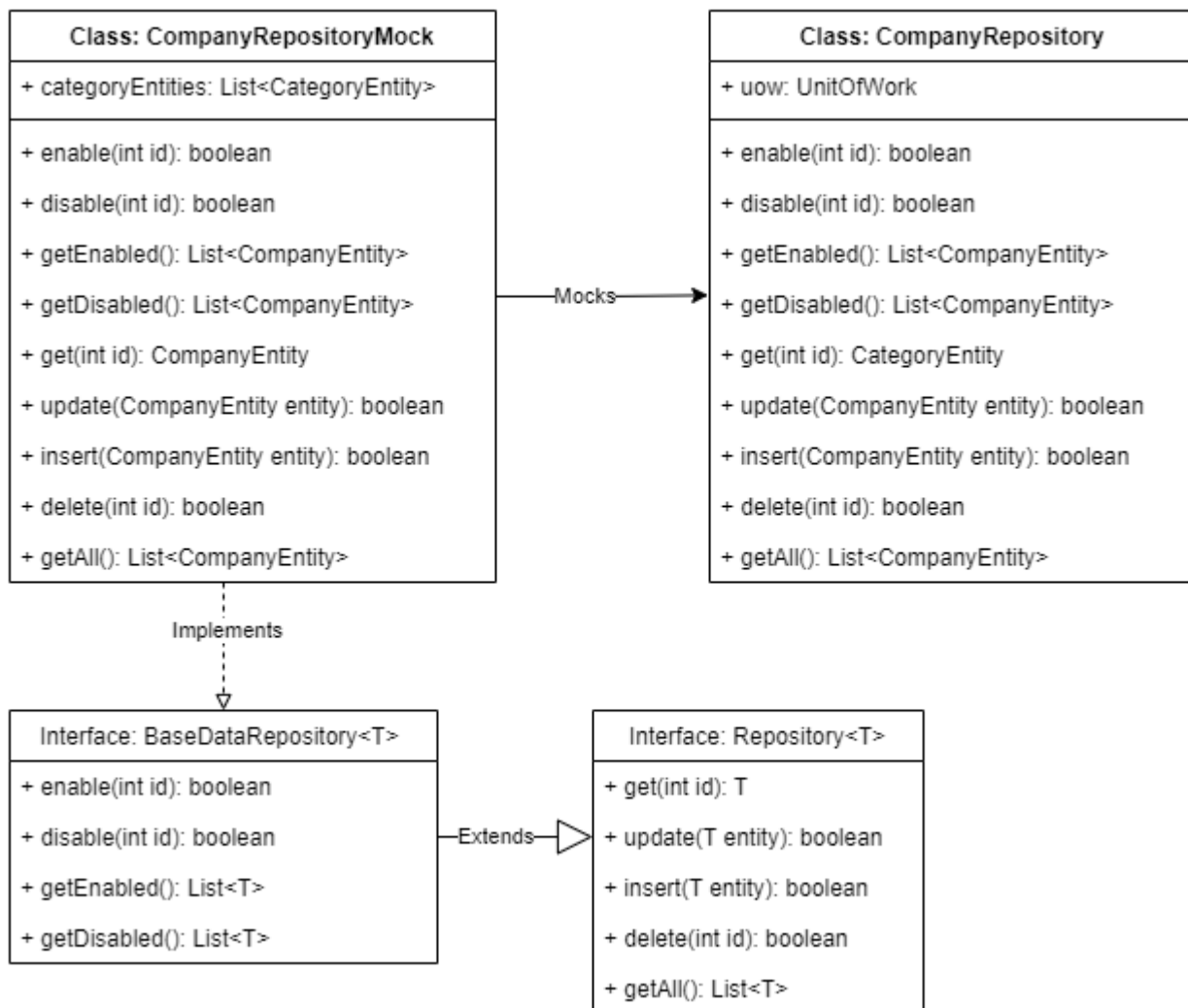
Die oben zusehende TestHelperKlasse, gilt als Professionell, da diese im Rahmen der Test Erstellung implementiert wurde, um die Funktionen besser testen zu können, um so nah wie möglich an die Produktiv Umgebung zu gelangen. Zusammenfassend ist das Ziel davon gewesen nah am Produktiv Code zu sein.

Fakes und Mocks (3P)

[Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten (die Fake/Mocks sind ohne Dritthersteller-Bibliothek/Framework zu implementieren); zusätzlich jeweils UML Diagramm mit Beziehungen zwischen Mock, zu mockender Klasse und Aufrufer des Mocks]



Das Mock Objekt (Klasse) `CategoryRepositoryMock` wurde erstellt, um das `CategoryRepository` testen zu können. Da bei dem `CategoryRepository` eine `UnitOfWork` genutzt wurde, musste diese ebenfalls gemockt werden, was ebenfalls im Code zu finden ist. Bei diesem Beispiel ist zu sehen, dass alle Funktionen gemockt werden, somit ist der Mock auch vollständig. Wichtig ist das, damit die Tests aussagekräftig sind. Anderenfalls könnte man sich sonst nicht auf die Tests verlassen und die Funktionen des `CategoryRepository` abtesten.



Hierbei ist es ähnlich zu dem anderen Mock Objekt, dies wird ebenfalls dafür verwendet um ein Repository zu mocken. In diesem Fall ist es das `CompanyRepository`. Genutzt wird es um ähnlich zu dem anderen Mock Objekt die Funktionalität des `CompanyRepository` überhaupt testen zu können. Ohne diesen Mock könnte nicht sichergestellt werden, ob die Implementierten Funktionen richtig funktionieren.

Kapitel 6: Domain Driven Design (8P)

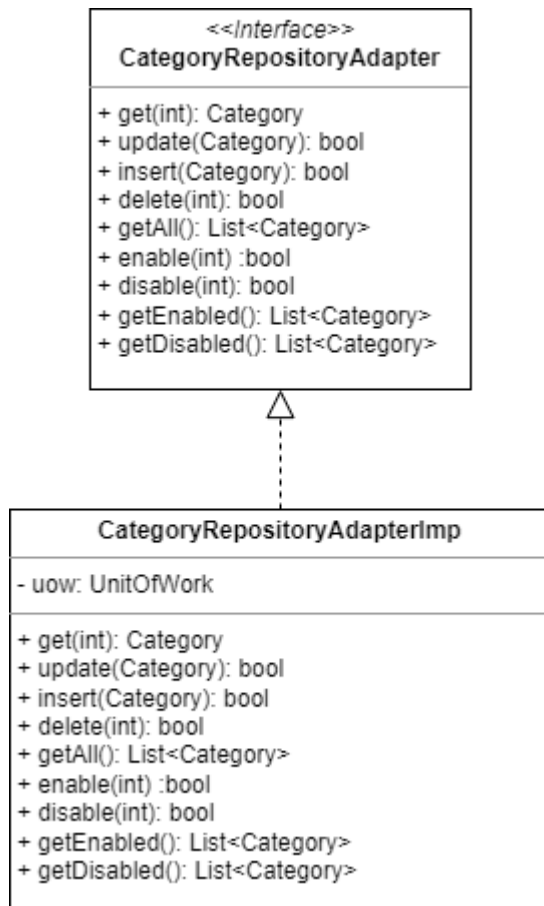
Ubiquitous Language (2P)

[4 Beispiele für die Ubiquitous Language; jeweils Bezeichnung, Bedeutung und kurze Begründung, warum es zur Ubiquitous Language gehört]

Bezeichnung	Bedeutung	Begründung
Purchase/Einkauf	Ein Eintrag eines Einkaufs/ einer Ausgabe	Diese Bezeichnung wurde am Anfang des Projekts als der Eintrag festgelegt, den die Benutzer tätigen können.
Category/Kategorie	Eine Einordnung des Einkaufs/ der Ausgabe in eine bestimmte Kategorie (z.B. Essen, Getränke)	Diese Bezeichnung wurde als Filter festgelegt, nachdem die Einkäufe gruppiert werden können. Er wurde zum Anfang des Projektes festgelegt
Company/Firma	Die Firma, bei der der Einkauf oder die Ausgabe getätigt wurde	Die Firma ist ein weiterer Filter für die Einkäufe. Der Begriff wurde zu Beginn des Projekts festgelegt. Dabei soll nur die Firma, bei der der Einkauf getätigt wurde, genannt werden, nicht die Firma, die Produkte herstellt.
Price/Kaufpreis	Der Kaufpreis des Einkaufs/ der Ausgabe	Festgelegte Größe für den Geldbetrag. Ist als Kennzahl festgelegt.

Repositories (1,5P)

[UML, Beschreibung und Begründung des Einsatzes eines Repositories; falls kein Repository vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist – NICHT, warum es nicht implementiert wurde]



Es wurde ein Repository erstellt, das die Erstellung und Speicherung der Entity „Category“ übernimmt. Ein Repository wird in unserer Implementierung im Datenbankkontext bereits verwendet, daher trägt das Repository den Namen „CategoryRepositoryAdapter“. Dieses übernimmt die Persistierung und Erstellung von der Entity.

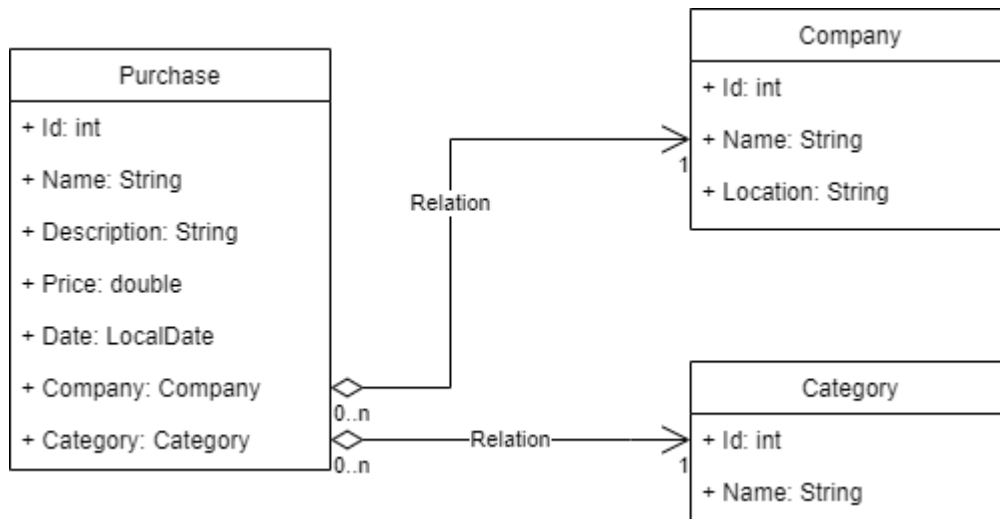
Aggregates (1,5P)

[UML, Beschreibung und Begründung des Einsatzes eines Aggregates; falls kein Aggregate vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist – NICHT, warum es nicht implementiert wurde]

Als Entities oder Value Objects stehen in unserer Implementierung nur die Entities „Purchase“, „Company“ und „Category“ zur Verfügung. Diese zu einem Aggregate zusammenzufassen wäre nicht sinnvoll, da die „Category“ und die „Company“ sich nicht mit der „Purchase“ verändern, und somit nicht in einem Repository verändert werden können.

Entities (1,5P)

[UML, Beschreibung und Begründung des Einsatzes einer Entity; falls keine Entity vorhanden: ausführliche Begründung, warum es keine geben kann/hier nicht sinnvoll ist– NICHT, warum es nicht implementiert wurde]



Es wurde die Entity „Purchase“ verwendet. Die Purchase ist eine Entity, da sie im Laufe der Anwendung verändert werden kann. Sie kann durch den Anwender bearbeitet werden und hat immer einen festen Lebenszyklus. Dieser beginnt entweder durch Erstellen einer neuen Purchase oder durch Abfragen einer Purchase aus der Datenbank. Die Purchase repräsentiert einen Einkauf in der Domäne.

Value Objects (1,5P)

[UML, Beschreibung und Begründung des Einsatzes eines Value Objects; falls kein Value Object vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist– NICHT, warum es nicht implementiert wurde]

Wir haben keine Value Objects verwendet, da wir die ID Generierung/Erzeugung bei unseren Datensätzen, über die Datenbank realisiert haben.

Kapitel 7: Refactoring (8P)

Code Smells (2P)

[jeweils 1 Code-Beispiel zu 3 unterschiedlichen Code Smells aus der Vorlesung; jeweils Code-Beispiel und einen möglichen Lösungsweg bzw. den genommen Lösungsweg beschreiben (inkl. (Pseudo-)Code)]

[CODE SMELL 1] Long Method

<https://github.com/Nilsksk/Costracker/commit/c5f4518ef31bbcc2e2f1202f6b735894ccfedf6e#diff-83b6fddc102a4191732a5eec24c01319a17283a1a8ff8c1ef8a27cbabf956773>

Wie in dem Commit zu erkennen, wurde die Funktion `getPurchase()` verkleinert und die Funktionalität in die Klasse `PurchaseConverter` ausgelagert. In dieser Klasse wurden verschiedene Funktionen geschaffen, um Purchases zu konvertieren. Die Funktionen sind `convertFrom()`, `buildPurchase()`, `getCompanyFrom()`, `getCategoryFrom()`. Diese Aufteilung erhöht nicht nur die Lesbarkeit, sondern auch das Nachvollziehen der einzelnen Funktion und deren Aufgaben.

[CODE SMELL 2] Long Parameter List

```
private void tryCreatePurchase(String purchaseName, LocalDate validatedDate, double validatedValue, Company company, Category category, PurchaseHandler purchaseHandler, String purchaseDescription, boolean created) {
```

Diese Funktion zeigt den Code Smell Long Parameter List. Das Problem welches dabei ist, ist das der Methodenaufruf sehr lang ist und dadurch schwer zu verstehen ist.

Gelöst werden kann das indem man das Purchase Objekt direkt übergibt, anstelle der einzelnen Werte (`purchaseName` und `purchaseDescription`). Ebenfalls kann der Parameter `created` weggelassen werden, da dieser hier falsch angelegt wurde. Die Lösung könnte wie folgt aussehen:

```
private void tryCreatePurchase(LocalDate validatedDate, double validatedValue, Purchase purchase, Company company, Category category, PurchaseHandler purchaseHandler)
```

Verbessert wird dadurch das Problem zwar nur bedingt, aber es ist schonmal kompakter.

[CODE SMELL 3] Large Class

Die Klasse Dialogue ist sehr lang, geschuldet an den vielen verschiedenen Optionen, die beim Navigieren innerhalb der Applikation benötigt werden. Um das zu verbessern, könnte man die verschiedenen Überprüfungen auslagern, um eine bessere Lesbarkeit und Flexibilität zu erhalten. Aussehen würde die Struktur dann so:

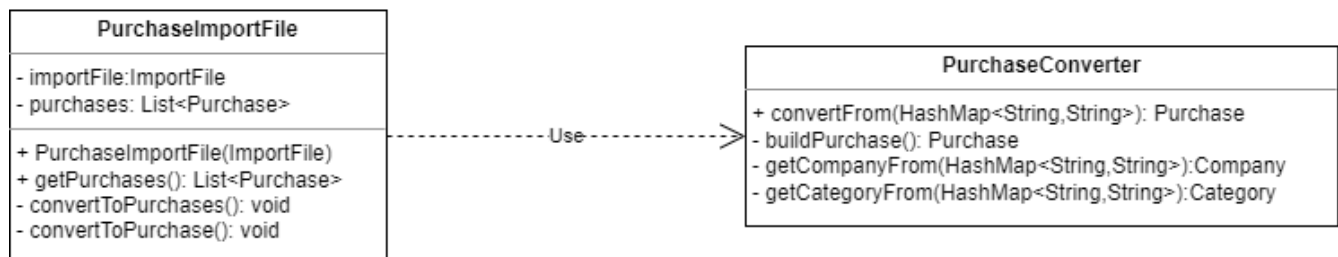
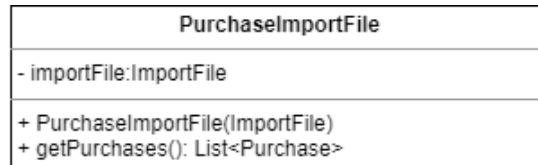
DialogueChecker

```
    checkInteraction1()  
    checkEditCompanyOrCategory()  
    checkHistoryAction()  
    checkTimespanAction()
```

Diese Funktionen würden dann innerhalb der Funktionen `interaction1()`, `editCompanyOrCategory()`, `historyAction()` und `timespanAction()` aufgerufen werden und somit die Klasse Dialogue verkleinern bzw. lesbarer machen.

3 Refactorings (6P)

[3 unterschiedliche Refactorings aus der Vorlesung anwenden, begründen, sowie UML vorher/nachher liefern; jeweils auf die Commits verweisen]



Die Methode `getPurchases` des `PurchaseImportFile` war eine lange Methode die unübersichtlich war. Aus diesem Grund wurde ein Refactoring durchgeführt, bei dem die Methode in kleinere Methoden durch das Extrahieren von Methoden verkleinert wurde.

(Commit:

<https://github.com/Nilsksk/Costtracker/commit/c5f4518ef31bbcc2e2f1202f6b735894ccfedf6e>)

DialogueHelper
<ul style="list-style-type: none"> + fillWhitespaces(int): String + startDialogue(String): void + inputDialogue(String): void + changeDialogue(String, String): void + returnInput(): String + getIntDialogue(String): int + mapTold(String): int + print(String): void + println(String): void + submitEntry(): boolean + saveData(String, String): boolean + validateCreation(boolean, String, String): void + interactQuestion(String): int + isValidInput(int, int): boolean + validateDeleteOrDeactivation(String): boolean + validateEnable(String): boolean

DialogueHelper
<ul style="list-style-type: none"> + addWhitespaces(int): String + printStartDialogueWith(String): void + printInputDialogueWith(String): void + printChangeDialogueWith(String, String): void + returnInputOfScanner(): String + printGetIdDialogueWith(String): int + mapToldFrom(String): int + print(String): void + printLine(String): void + submitEntry(): boolean + printSaveDataDialogueWith(String, String): boolean + printCreationDialogueWith(boolean, String, String): void + printInteractQuestionWith(String): int + inputIsGreaterThanZeroAndBelowMaxInput(int, int): boolean + printDeleteOrDeactivateDialogueWith(String): boolean + printEnableDialogueWith(String): boolean

Der DialogueHelper wurde einem Refactoring unterzogen. Dabei wurden die Methoden umbenannt, sodass sie besser ausdrücken, was in den Methoden gemacht wird. Dies war davor nicht ersichtlich und führte zu Verwirrung.

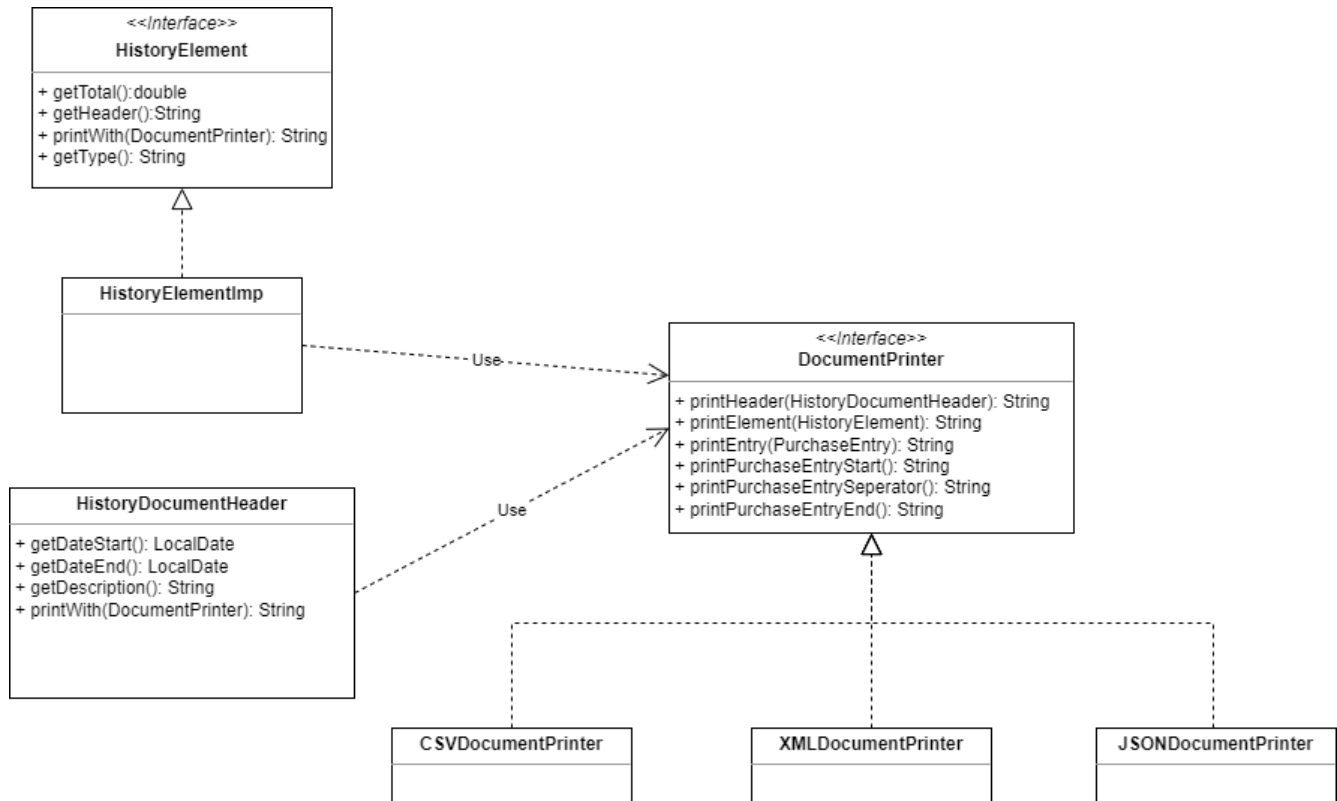
(Commit:

<https://github.com/Nilsksk/Costtracker/commit/fe9306a97011cacb863ad52065efdce5290e8a6>)

Kapitel 8: Entwurfsmuster (8P)

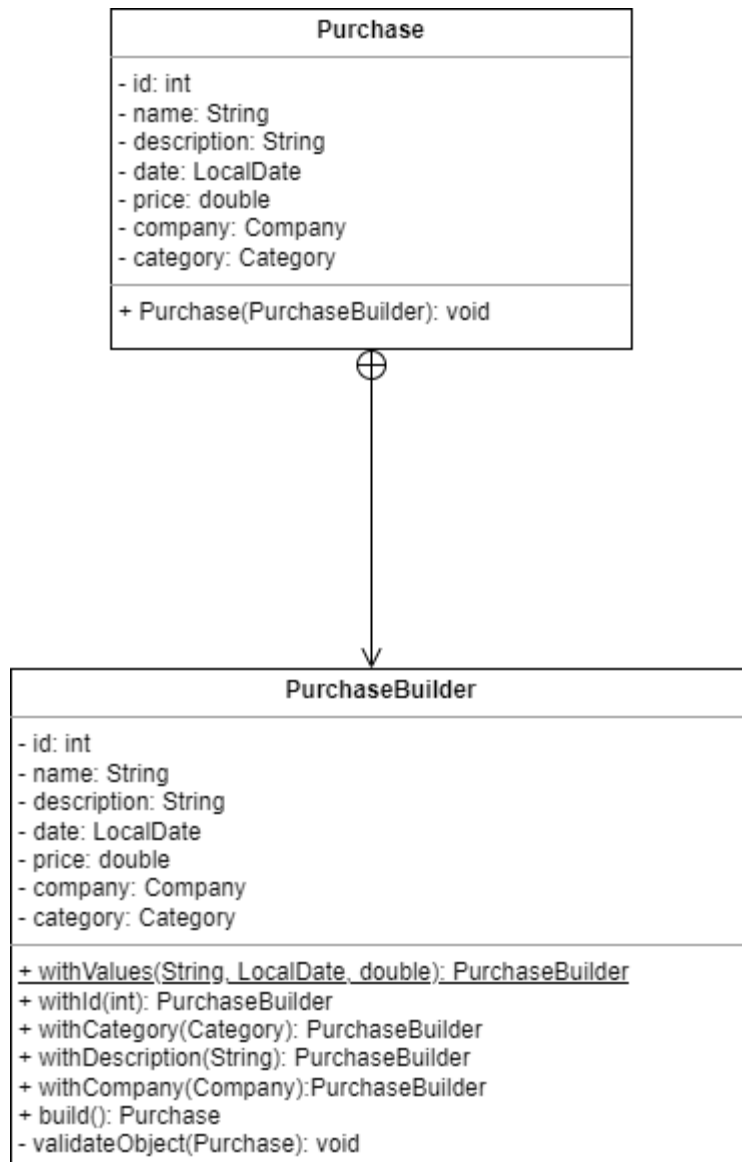
[2 unterschiedliche Entwurfsmuster aus der Vorlesung (oder nach Absprache auch andere) jeweils sinnvoll einsetzen, begründen und UML-Diagramm]

Entwurfsmuster: Strategy (4P)



Es wurde das „Strategy“ Entwurfsmuster verwendet, um ein Komponenten eines Dokument zu drucken. „Drucken“ bedeutet die Komponenten des Dokuments in ein gewünschtes „String“-Format umwandeln. Dabei gibt es ein Interface „DocumentPrinter“. Dieses kann von verschiedenen Klassen implementiert werden. Diese Klassen können dann verschiedene „Printer“ für verschiedene Dateiformate implementieren. Hierdurch wird eine Konditionalstruktur vermieden und das OCP umgesetzt, da für eine neues Datenformat nur eine neue Klasse angelegt werden muss, die das Interface „DocumentPrinter“ implementiert.

Entwurfsmuster: Builder (4P)



Für die Klasse **Purchase** wurde das Builder Pattern eingeführt. Die Klasse verfügt über viele Felder, die zuvor durch einen langen Konstruktor initialisiert wurden. Durch das Builder Pattern kann die Klasse jetzt einfacher und fehlerfreier erzeugt werden.