

Programmentwurf WG-ShoppingList

Name: Rico Rauschkolb
Matrikelnummer: 9072443

Abgabedatum: 29.05.2022

Allgemeine Anmerkungen:

- *es darf nicht auf andere Kapitel als 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)*
- *sollten mündliche Aussagen den schriftlichen Aufgaben widersprechen, gelten die schriftlichen Aufgaben (ggf. an Anpassung der schriftlichen Aufgaben erinnern!)*
- *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*
 - *Beispiele*
 - *“Nennen Sie jeweils eine Klasse, die das SRP einhält bzw. verletzt.”*
 - *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: volle Punktzahl ODER falls im Code mind. eine Klasse SRP verletzt: halbe Punktzahl*
- *verlangte Positiv-Beispiele müssen gebracht werden*
- *Code-Beispiel = Code in das Dokument kopieren*

Kapitel 1: Einführung

Übersicht über die Applikation

Die Applikation ermöglicht das Verwalten von Einkaufslisten. Dies kann beispielsweise in einer Wohngemeinschaft sinnvoll sein. Die Applikation ist als eine CommandLine Anwendung aufgebaut. Potentiell kann aber jede Art des User Interface gewählt werden und die CommandLine ersetzt werden. Der Nutzer hat die Möglichkeit neue Einkaufslisten anzulegen und diesen einen Namen zu geben. Diesen Listen kann er Items hinzufügen. Dabei kann er eine Menge sowie eine Einheit angeben. Die Einkaufslisten kann er sich anzeigen und den Status der Items abfragen. Des Weiteren hat er die Möglichkeit Items abzuhaken und dadurch deren Status auf Gekauft zu setzen. Hierbei kann er einen Preis angeben. Dieser wird in der Berechnung der Gesamtkosten der Einkaufsliste berücksichtigt. Hintergrund werden die Daten in CSV-Dateien gespeichert. Dies kann jedoch durch die Architektur der Anwendung geändert werden und durch eine andere Datenhaltung ausgetauscht werden.

Wie startet man die Applikation?

Im Order indem die .jar liegt `java -r anwendung.jar`

Es kommt leider teilweise zu Problemen bei Umlauten. Dies geschieht jedoch nur wenn die jar genutzt wird. Wenn das Programm durch die Entwicklungsumgebung gestartet wird ist dies nicht der Fall.

Wie testet man die Applikation?

Im Projektordner `mvn clean test`

Kapitel 2: Clean Architecture

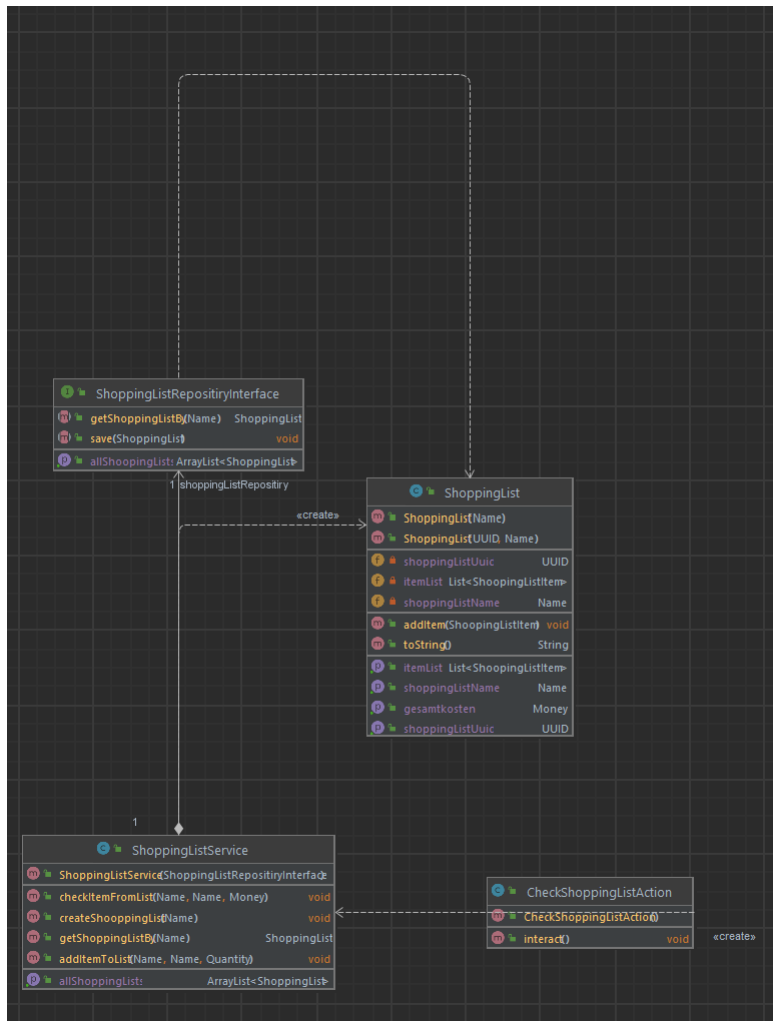
Was ist Clean Architecture?

Durch Clean Architecture soll der fachliche Kern der Anwendung unabhängig von der Datenhaltung, Frameworks oder sonstigen Bausteinen sein. Die Anwendung wird in Schichten abgebildet. Dadurch kann der Entwickler seinen Code besser strukturieren und die Einhaltung der Prinzipien wird vereinfacht. Die Schichten unterscheiden sich dabei wie schnell sie sich verändern können. Umso weiter innenliegend eine Schicht ist, umso weniger sollte sie sich verändern. Zugriffe können nur nach innen gerichtet sein. Das heißt Code kann nur auf Schichten zugreifen, die weiter innenliegend sind als seine eigene. Dies gewährleistet, dass Äußere Schichten ausgetauscht werden ohne eine Auswirkung für eine innenliegende Schicht. Durch den Aufbau der Anwendung kann dies direkt durch Dependencies definiert werden und dadurch kann dagegen nicht verstoßen werden.

Es existieren unterschiedliche Arten wie Clean Architecture umgesetzt ist. Dabei unterscheiden sich die Anzahl an Schichten und dadurch die Abstraktionsebenen. Alle haben jedoch das Ziel der Domänen Code möglichst unabhängig von Infrastruktur oder Framework Code zu halten.

Analyse der Dependency Rule

Positiv-Beispiel: Dependency Rule



ShoppingListService befindet sich im Appllication Code der Anwendung. Dieser darf somit nur auf Klassen der Domain Schicht zugreifen. Dies ist durch die Klassen ShoppingListRepository und ShoppingList dargestellt. Erstellt wird der Service durch die verschiedenen Aktionen in der Plugin Database Schicht.

Negativ-Beispiel: Dependency Rule

Ein Negativ-Beispiel ist durch die definierten Maven Dependencies nicht zu finden. Es ist ohne Anpassen der Dependencies nicht möglich aus inneren Schichten auf Äußere zuzugreifen. Dies ist beispielhaft im unterem Bild dargestellt. Dort wurde versucht aus dem Domain Code auf dem Appllication Code zuzugreifen.

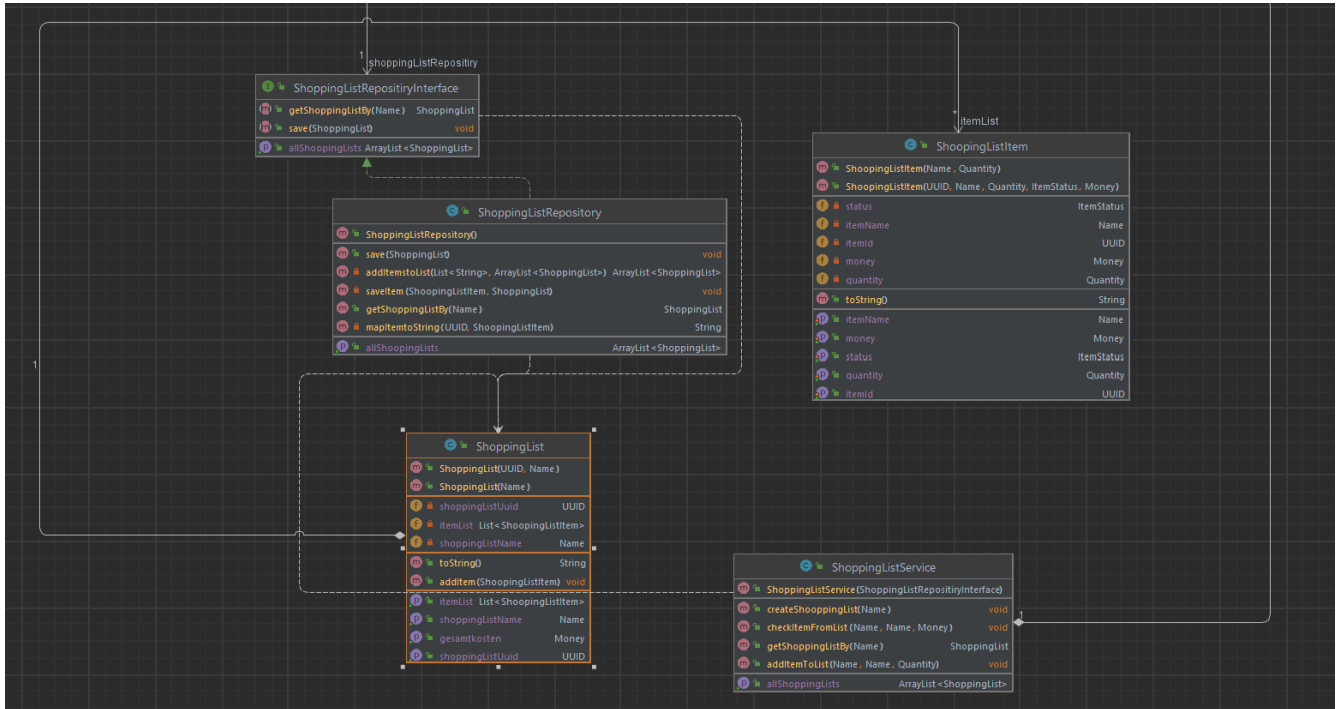
```
ShoppingListService shoppingListService = new ShoppingListService();
```

Cannot resolve symbol 'ShoppingListService'

Add dependency on module '2-cleanproject-application' Alt+Umschalt+Eingabe

More actions... Alt+Eingabe

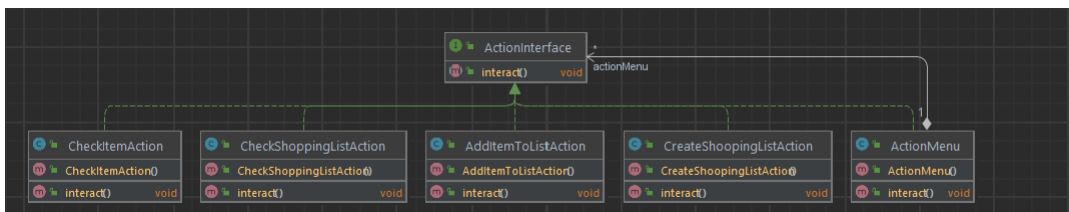
2 Positiv-Beispiel: Dependency Rule



ShoppingList greift auf keine Klassen aus höheren Schichten zu. Es wird nur aus höheren Schichten(Application Code) auf diese zugegriffen.

Analyse der Schichten

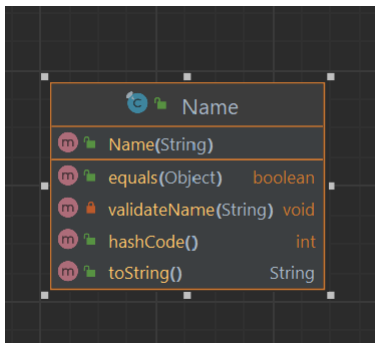
Plugin-Database Schicht: Action CheckItemAction



CheckItemAction ist neben weiteren Action Klassen dafür zuständig mit den Aktionen des Nutzers umzugehen. Dazu implementiert die Klasse das ActionInterface, dadurch wird gewährleistet dass die unterschiedlichen Aktionen einen gleichen Aufbau haben und erst zur Laufzeit entschieden wird welche Aktion durchgeführt wird. Im speziellen hat die Klasse die Aufgabe die Eingaben des Nutzers für das Abhaken eines Items entgegenzunehmen. Da es sich klar um ein von der Implementierung des User

Interface handelt, ist diese in der Plugin Database Schicht. Eine Änderung der Implementierung ist somit leicht möglich.

Domain Schicht: Name

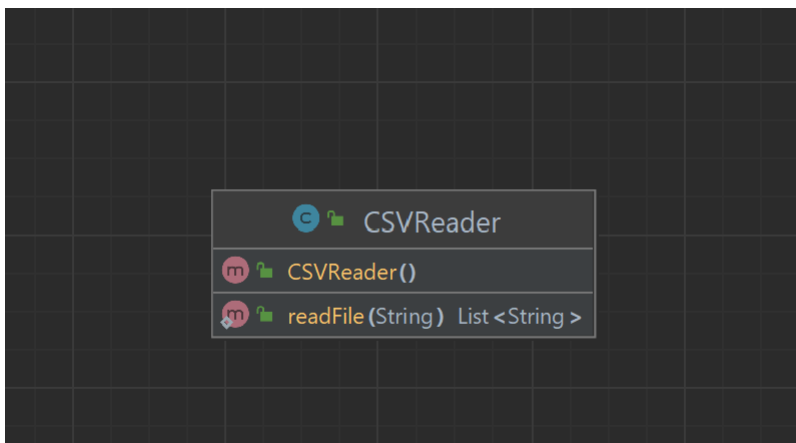


Name stellt ein Value Object dar. Die Klasse ist als Final deklariert und Objekte können sich somit nach dem Erstellen nicht mehr ändern. Um Änderungen durchzuführen muss eine neue Instanz der Klasse erstellt werden. Aufgaben der Klasse ist die überprüfen der Korrektheit von Namen und mögliche Verstöße als eine Exeption aufzuzeigen. Die Klasse befindet sich im Domain Code der Anwendung. Dadurch sollte sich die Implementierung nur sehr selten ändern. Dies könnte durch fachliche Anforderung an die Korrektheit von Namen der Fall sein.

Kapitel 3: SOLID

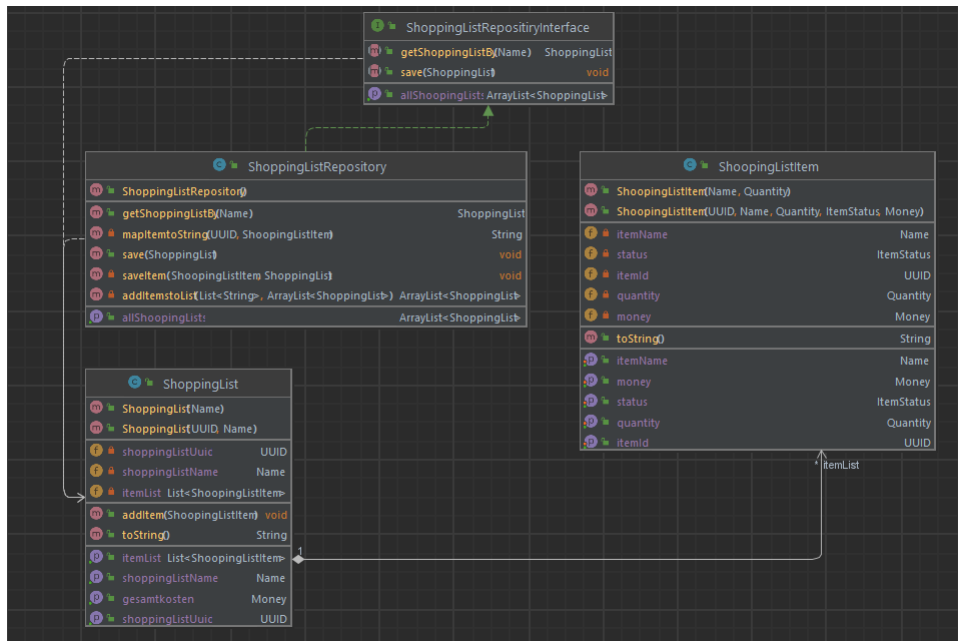
Analyse Single-Responsibility-Principle (SRP)

Positiv-Beispiel



Als Positiv Beispiel kann die Klasse CSVReader genannt werden. Diese Klasse hat die Aufgabe den Inhalt einer CSV Datei zu lesen und der Inhalt als Liste zurückzugeben. Dies ist die einzige Aufgabe der Klasse.

Negativ-Beispiel

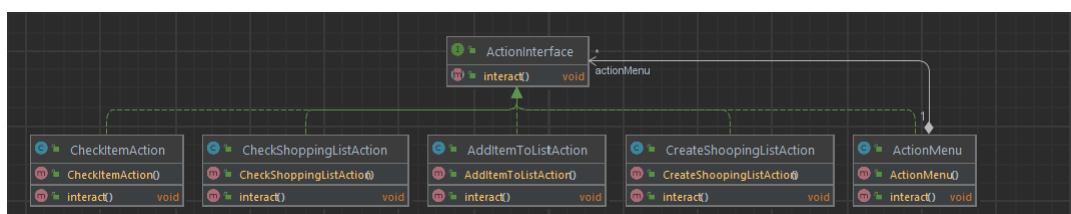


Als Negativ Beispiel kann die Klasse ShoppingListRepository genannt werden. Aufgaben dieser Klasse ist die konkrete Implementierung des ShoppingListRepositoryInterface. Diese würde prinzipiell kein Problem darstellen. Jedoch ist durch die Implementierung durch CSV Dateien und der Aufteilung ist Shopping List und ShoppingListItem zusätzliche Komplexität hinzugekommen. Deshalb hat die Klasse nun sowohl die Aufgabe ShoppingList in ein Format zum abspeichern zu konvertieren. Aber gleichzeitig auch die ShoppingList Items. Somit verwaltet diese Klasse sowohl ShoppingList als auch ShoppingListItem.

Um dieses Problem zu lösen könnte ein separates Repository für ShoppingListItem erstellt werden. Dieses wäre dann nur für das Speichern und Laden der Items zuständig. Wichtig ist dabei das dies nicht im Domain Code definiert wird. Da dies ein Implementierungsdetail ist und ein direkter Zugriff auf Items einer ShoppingList nicht möglich sein sollte.

Analyse Open-Closed-Principle (OCP)

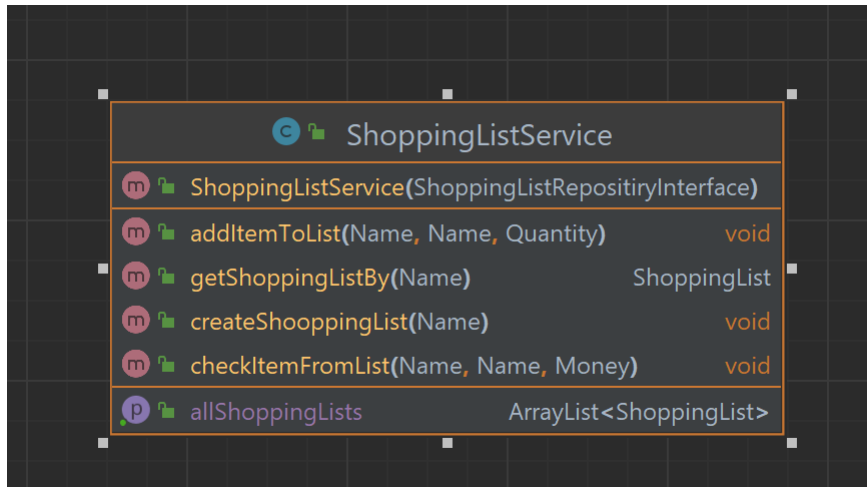
Positiv-Beispiel



Ein Positiv Beispiel ist die Implementierung der User Aktionen. Diese sollen über das ActionMenu verfügbar sein. Dazu werden sie dort in einer Hashmap gespeichert. Um dies zu ermöglichen implementieren alle Aktion das ActionInterface. Wählt der User nun eine Aktion aus wird die korrekte Aktion aus der Hashmap gewählt und in der interact Methode durchgeführt. Dadurch kann sich die

Implementierung der interact Methode in den einzelnen Aktionen ändern, jedoch ändert sich dadurch nicht der andere bestehende Code. Gleich verhält es sich mit neuen Aktionen die in die Hashmap eingefügt werden können.

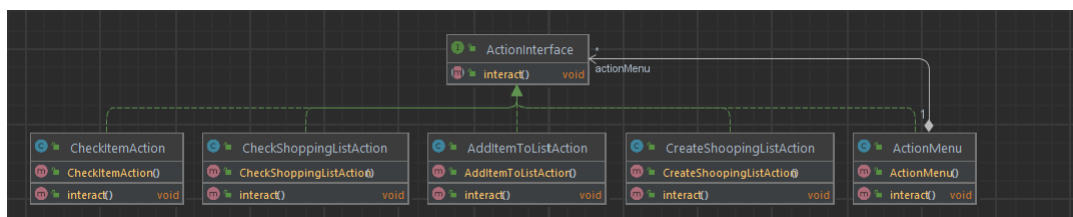
Negativ-Beispiel



Ein negativ Beispiel findet sich mit dem ShoppingListService. Bei Änderungen an der Implementierung des Services müssen alle Klassen die diesen instanziiieren auch geändert werden. Dies trifft auf die verschiedenen User Aktionen zu. Diese greifen in ihrer interact Methode direkt auf den Service zu. Gelöst werden könnte dies durch ein Interface und somit einen weiteren Abstraktionsebene. Anstatt eine direkte Instanz des Service würden die Aktionen nur das Interface nutzen. Somit könnte sich die korrekte Implementierung ändern ohne Auswirkungen nach außen.

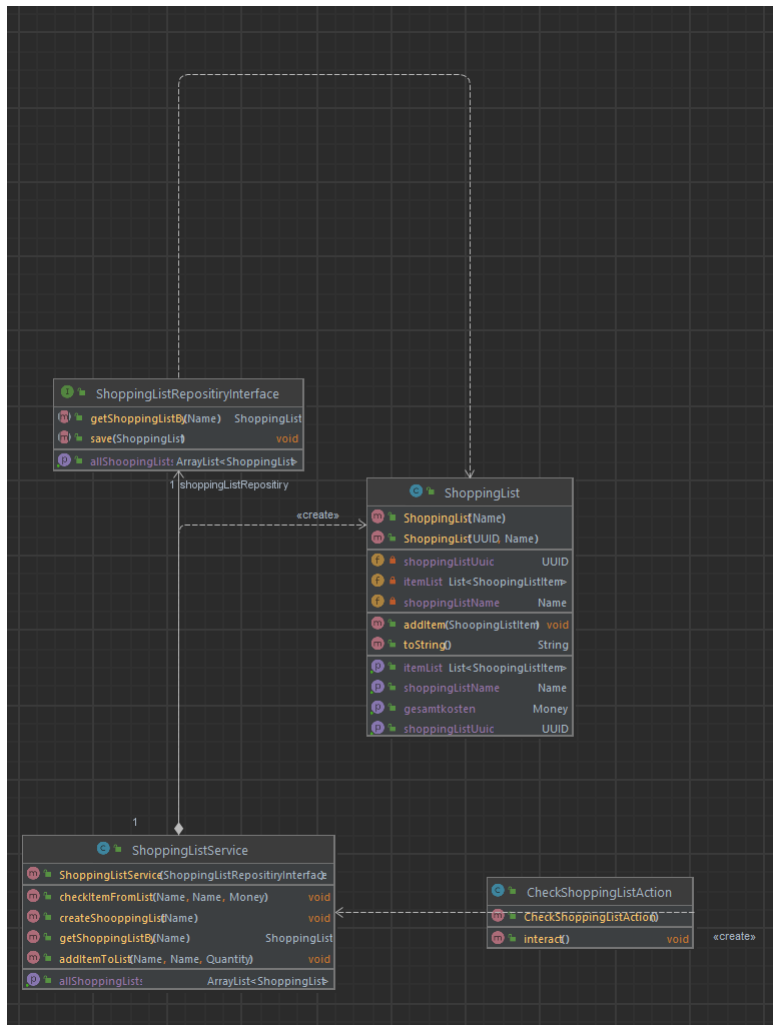
Dependency-Inversion-Principle (DIP)

Positiv-Beispiel



An statt, dass das ActionMenu direkt von den Implementierungen der Aktionen abhängt wird das Dependency-Inversion-Principle angewendet. Dadurch hängt sowohl das Menu als auch die konkrete Implementierung von dem Interface ab. Dadurch wird erreicht, dass wenn sich die Implementierung der Aktionen ändert dies keine Auswirkung auf das ActionMenu hast.

Negativ-Beispiel

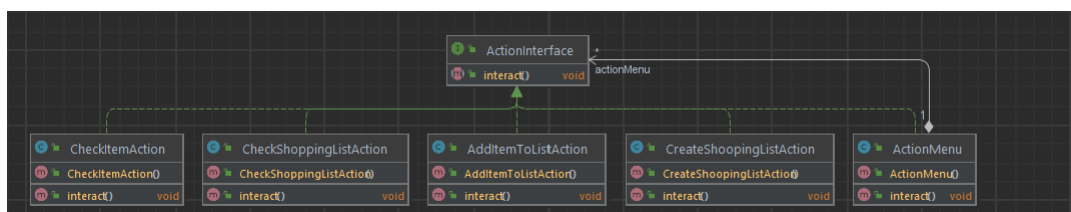


Im Gegensatz dazu hängt das `ShoppingListRepositoryInterface` direkt an der konkreten Implementierung des `ShoppingListServices`. Dadurch ist das Dependency-Inversion-Prinzip dort nicht eingehalten. Um dieses einzuhalten darf eine Klasse nicht direkt von der konkreten Implementierung einer anderen abhängen.

Kapitel 4: Weitere Prinzipien

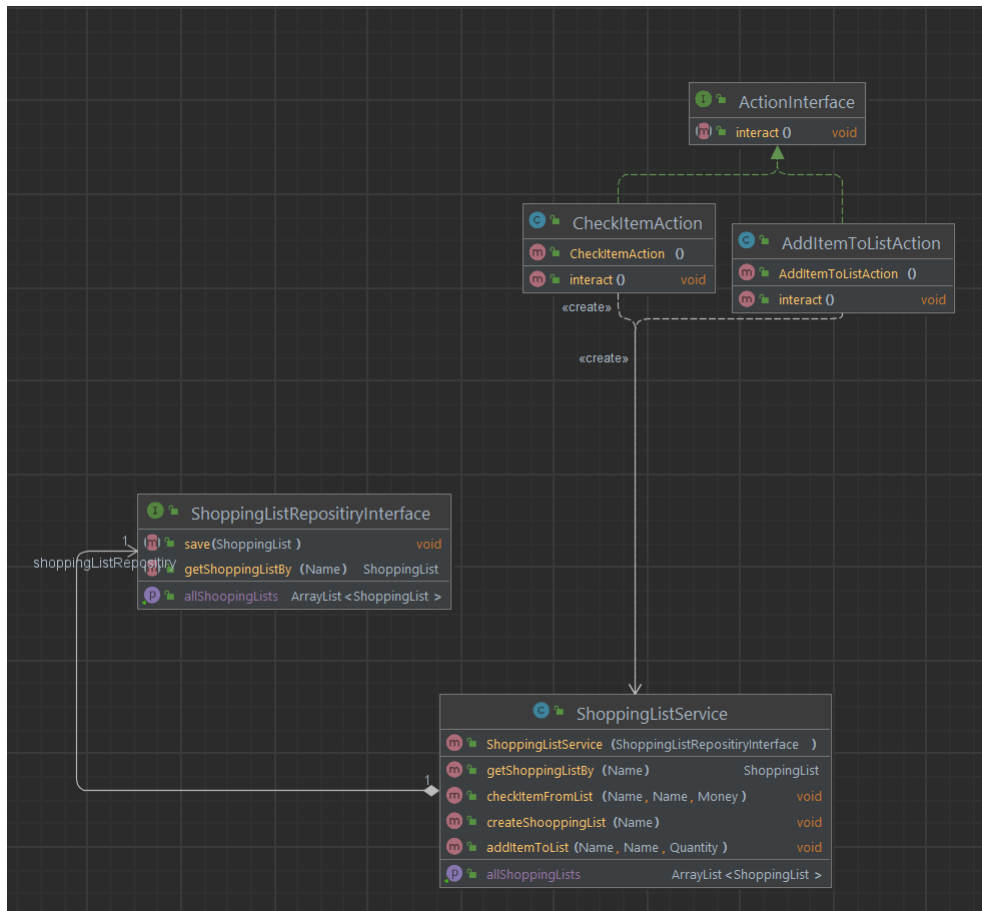
Analyse GRASP: Geringe Kopplung

Positiv-Beispiel



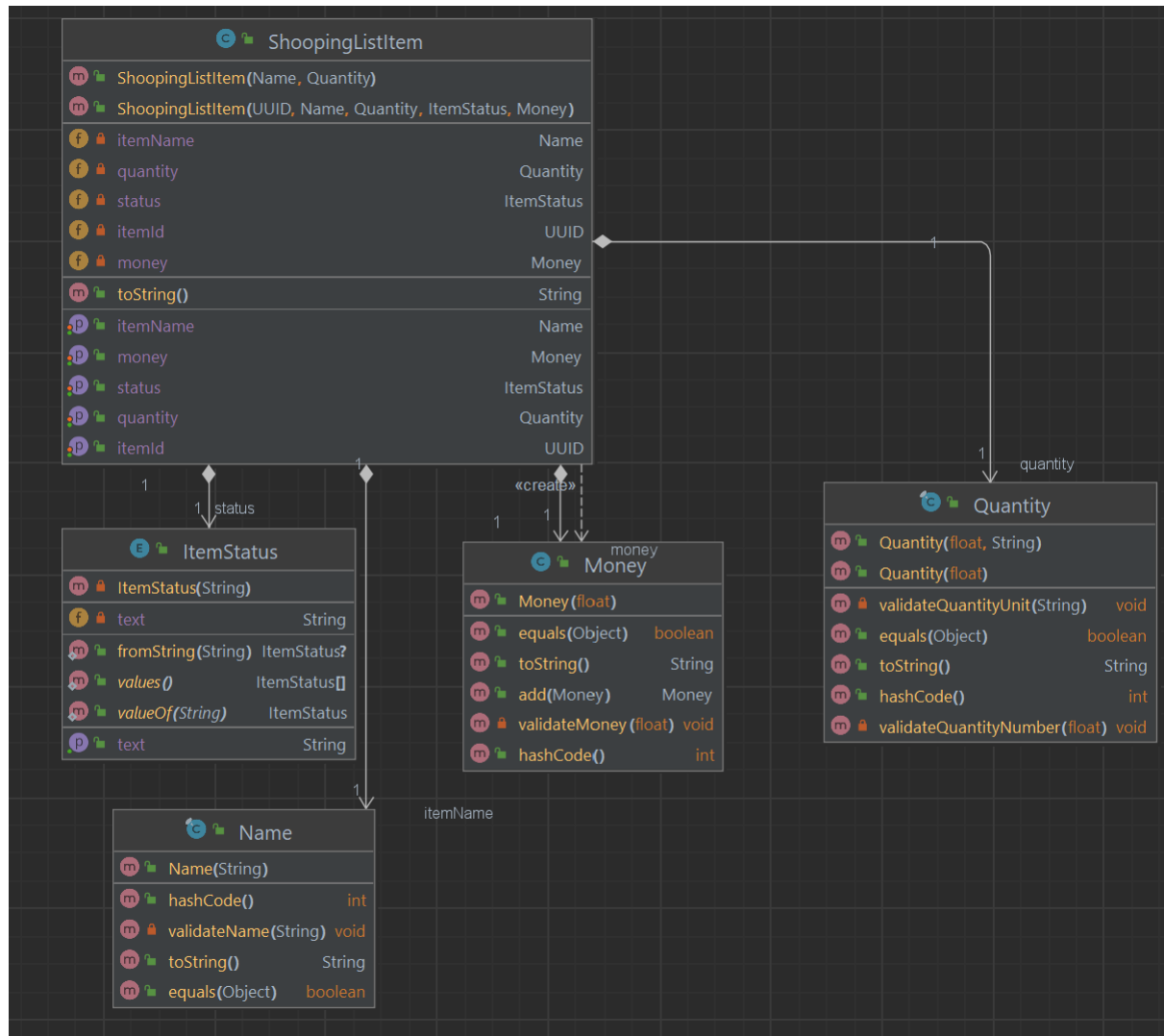
An dieser Stelle kann die Umsetzung der Aktionen im User Interface genannt werden. Die Geringe Kopplung wird erzielt indem alle Aktionen das selbe Interface implementieren. Sie unterscheiden sich in der konkreten Implementierung. Die Kopplung zwischen Action Menu und den einzelnen Aktionen ist somit gering.

Negativ-Beispiel



Als Negativ Beispiel kann die Kopplung zwischen **UserService** und **ShoppingListService** genannt werden. Die einzelnen Aktionen des **UserService** instantiieren direkt den **ShoppingListService**. Somit ist dort eine hart Kopplung zwischen dem **Userinterface** und dem **Application Code**. Dies könnte aufgelöst werden indem der **Service** ein Interface implementiert. Dadurch würde eine weitere Abstraktionsebene eingefügt und die Kopplung verringert.

Analyse GRASP: Hohe Kohäsion



Als Beispiel für hohe Kohäsion kann das `ShoppingListItem` genutzt werden. Dabei besteht es aus unterschiedlichen Attributen, welche in anderen Klassen ausgelagert sind. Diese bilden zusammen eine feste Einheit als `ShoppingListItem` und dadurch besteht eine hohe Kohäsion.

Don't Repeat Yourself (DRY)

Vorher: [gitignore update again · Rico2000/ASEShoppingList@fb76544 \(github.com\)](#)

Nacher: [Refactor dont repeat yourself · Rico2000/ASEShoppingList@8961fb3 \(github.com\)](#)

Wie im Nachher Commit zu sehen waren an zwei Stellen in zwei unterschiedlichen Klassen zweimal der exakt selbe Code zu finden. Dieser Code dient dazu den `StringInput` durch Regexp aufzuteilen und in zwei Teile zu unterteilen. Anstatt zweimal den selben Code zu haben wurde eine Helper Klasse erstellt, welche eine statische Methode beinhaltet. Es wurde sich für eine Helper Klasse entschieden da der Code nicht klar einer Klasse zuzuordnen ist. Der Code ist stark abhängig von der Implementierung und in diesem Fall von dem CLI Interface.

Kapitel 5: Unit Tests

10 Unit Tests

Unit Test	Beschreibung
TestMoney testNumberIsNegative	Testet ob man Money mit negativen Zahlen erstellen kann. Erwartet wird ein IllegalQuantityException.
TestMoney testAddMoney	Testet dass addieren von Money und ob das Ergebnis korrekt ist.
TestName testNameIsTooShort	Testet ob man zu kurze Namen erstellen. Erwartet wird ein IllegalArgumentException
TestName testNameContainsIllegalCharacters	Testet ob man Namen mit illegalen Characters(Sonderzeichen Leerzeichen) erstellen kann. Erwartet wird eine IllegalArgumentException
TestName testNamePositiveCases	Testet positive Fälle für das Erstellen eines Namens
TestName testNameEquals	Testet ob zwei Namen mit gleichem String gleich sind
TestQuantity testNegativeAndZeroValuesThrowException	Testet ob man Quantity mit Null oder Negativen Werten erstellen kann. Erwartet wird eine IllegalQuantityException
TestQuantity testTooLongUnit	Testet ob man zu lange Units erstellen kann. Erwartet wird eine IllegalUnitException
TestQuantity testUnitContainsIllegalCharacters	Testet ob die Einheit illegale Characters(Sonderzeichen Zahlen) enthält. Erwartet wird eine IllegalUnitException.
TestQuantity testPositiveCases	Testet die positiven Fälle für das erstellen kann.
TestShoppingListService testCreateShoppingList	Testet ob das Erstellen einer ShoppingList funktioniert. Es wird die Anzahl an ShoppingListen überprüft.
TestShoppingListService testAddItemToList	Testet das Hinzufügen eines Items zu einer bestehenden ShoppingListe. Es wird überprüft ob das korrekte Item in der Liste ist

ATRIP: Automatic

Um die automatische Testausführung zu gewährleisten bietet Maven die Möglichkeit alle Tests auf einmal durchlaufen zu lassen mit dem Command `mvn clean test`. Dabei kann überprüft werden ob diese erfolgreich sind oder nicht. Dies sollte nach jedes nennenswerten Änderung gemacht werden.

```
[INFO] -----
[INFO] Reactor Summary for x-cleanproject 0.0.1-SNAPSHOT:
[INFO]
[INFO] x-cleanproject ..... SUCCESS [ 0.145 s]
[INFO] 4-cleanproject-plugin ..... SUCCESS [ 0.652 s]
[INFO] 3-cleanproject-domain ..... SUCCESS [ 1.588 s]
[INFO] 2-cleanproject-application ..... SUCCESS [ 0.626 s]
[INFO] 1-cleanproject-adapters ..... SUCCESS [ 0.038 s]
[INFO] 0-cleanproject-plugin-database ..... SUCCESS [ 0.294 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3.489 s
[INFO] Finished at: 2022-05-28T15:47:57+02:00
[INFO] -----
```

ATRIP: Thorough

Der Test `testNameIsShort` deckt alle möglichen Fälle ab wie ein Name zu kurz sein könnte. Ein Name muss aus mindestens 3 Zeichen bestehen. Somit werden alle möglichen Fälle bei denen eine Exception geworfen werden sollte abgedeckt.

```
@Test
public void testNameIsToShort() {
    assertThrows(IllegalArgumentException.class, () -> new Name("de"));
    assertThrows(IllegalArgumentException.class, () -> new Name("e"));
    assertThrows(IllegalArgumentException.class, () -> new Name(""));
}
```

Der `testNameContainsIllegalCharacters` deckt nicht alle möglichen Fälle ab die er abdecken sollte. Es würde neben Zahlen und Leerzeichen noch andere Characters geben, die nicht verwendet werden dürfen. Dazu würden Sonderzeichen zählen. Durch ein iteratives Vorgehen bei der Erstellung hätte dies verhindert werden können.

```

@Test
public void testNameContainsIllegalCharacters() {
    assertThrows(IllegalArgumentException.class, () -> new Name("d12"));
    assertThrows(IllegalArgumentException.class, () -> new Name(" "));
    assertThrows(IllegalArgumentException.class, () -> new Name("Ric0"));
}

```

ATRIP: Professional

Positiv:

Die Verwendung von BeforeEach verhindert, dass man nicht in jedem Test den gleichen Code wiederholen muss. Es kann somit ein Allgemeines Setup für jeden Test aufgebaut werden. Es gibt auch noch weitere Methoden wie BeforeAll um die Duplikate zu reduzieren. Im unteren Anwendungsfall wird ein Mock Objekt erstellt und der zu Testende Service. Diese müsste sonst bei jedem Test hinzugefügt werden.

```

9 usages
static ShoppingListRepositoryInterface mock;
5 usages
static ShoppingListService service;

@BeforeEach
public void setUpBeforeClass(){
    mock = new MockShoppingListRepository();
    service = new ShoppingListService(mock);
}

```

Negativ:

```

@Test
public void testAddItemToList()
    throws IllegalArgumentException, ShoppingListNotFoundException, IllegalQuantityException, ShoppingListItemAlreadyExistsException {
    mock.save(new ShoppingList(new Name("Test")));
    service.addItemToList(new Name("Test"), new Name("Eier"), new Quantity(number: 10));
    assertEquals(mock.getAllShoppingLists().get(0).getItemList().size(), actual: 1);
    assertEquals(mock.getAllShoppingLists().get(0).getItemList().get(0).getItemName(), new Name("Eier"));
    assertEquals(mock.getAllShoppingLists().get(0).getItemList().get(0).getQuantity(), new Quantity(number: 10));
}

```

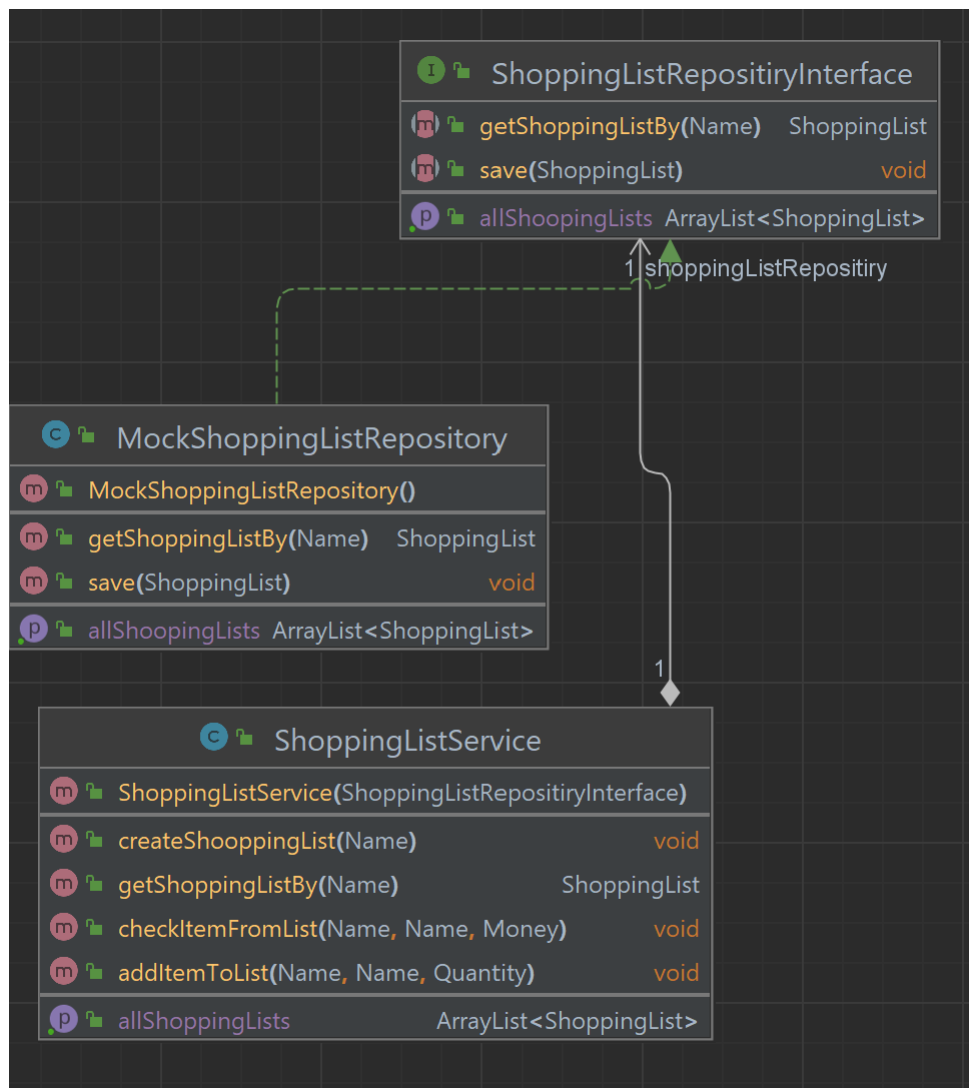
Bei Tests sollte der selbe Qualitätsstandard gelten wie bei produktiven Code. Deshalb sollte der Code lesbar und strukturiert sein. Bei dem oben aufgeführten Test ist dies nicht der Fall. Es ist nicht ersichtlich was passiert und was die Assert Methoden genau machen. Man könnte das Setup des Tests in Methoden auslagern. Des Weiteren könnte man der tatsächliche Wert davor einer Variable hinzufügen.

Code Coverage

Class, %	Method, %	Line, %
45,8% (11/24)	41,9% (36/86)	24,9% (101/406)

Im Projekt konnte mit den erstellten Test eine Line Coverage von 45 %, Method Coverage von 45 % und eine Class Coverage von 45%. Dabei ist zu beachten, dass es nicht sinnvoll ist manche Teile des Projektes zu testen. Dazu zählen die Implementierungen der Repositories, sowie die CSV Reader und CSVWriter. Diese können nicht getestet werden da diese direkt mit der Datenhaltung interagieren. Um diese zu testen müsste man Testdateien erstellen und die Tests an diese anpassen

Fakes und Mocks



Im Programm konnte nur an einer Stelle ein Mock sinnvoll zum Testen genutzt werden. Um die Datenhaltung durch ein Mock zu ersetzen wurde das `ShoppingListRepositoryInterface` implementiert. Die Funktionalität wurde durch einfache Listen ersetzt und dadurch die Korrektheit gewährleistet. Somit kann die Funktionalität des `ShoppingListService` nun separat getestet werden.

Eine weitere Möglichkeit wäre wenn man die Verwaltung von ShoppingListItem aus dem ShoppingListRepository herauszieht könnte man auch dieses durch ein Mock ersetzen.

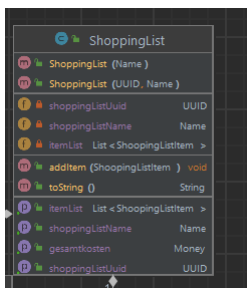
Kapitel 6: Domain Driven Design

Ubiquitous Language

Bezeichnung	Bedeutung	Begründung
ShoppingList	Eine Einkaufliste mit Namen und Items	Klare Aussage über was eine Einkaufliste ist. (Englische Bezeichnung)
ShoppingListItem	Ein Bestandteil einer Einkaufliste	Klar definiert, dass es zu einer Shoppinglist gehört und ein Bestandteil davon ist.
ItemStatus	Gibt an ob ein Item noch - offen oder bereits gekauft ist	
Quantity	Anzahl und Einheit (z.b 500 Gramm 500 Stück)	Gibt klare Aussage über den Inhalt des Value Objects

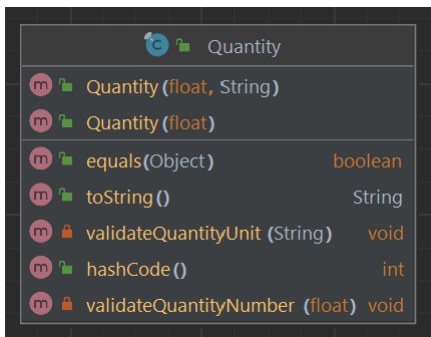
Entities

Eine Entity ist ein Daten Objekt. Es besteht aus Value Object und primitiven Datentypen. Die Gleichheit zweier Entities muss über ein Attribut gewährleistet werden. In diesem Fall durch eine UUID. Eine Entity ist eine Einheit von Attributen die zusammen verwaltet wird.



Value Objects

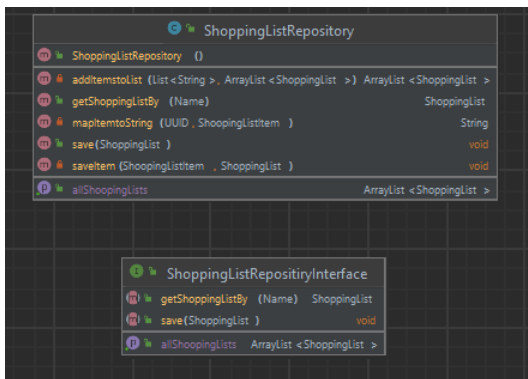
Value Object ist ein unveränderbares Objekt, welches möglichst einfache Datentypen ersetzen soll. Wenn der Wert des Value Object sich ändern soll muss ein neues erstellt werden. Dafür sind die Klassen als final deklariert. Ein Vorteil von Value Objects ist, dass sie die Korrektheit direkt im Konstruktor überprüfen können und dadurch die Korrektheit gesichert ist.



Quantity ist durch eine Anzahl und eine Einheit gekennzeichnet. Die Einheit ist dabei optional. Die Anzahl ist ein float, dieser darf nicht negativ oder null sein. Die Einheit darf keine Sonderzeichen oder Zahlen enthalten. Die Korrektheit dieser Vorgaben wird in den Methoden `validateQuantityUnit` und `validateNumber` überprüft. Damit nach dem Erstellen keine Veränderungen an dem Objekt sowie dessen Attributen vorgenommen werden können, ist sowohl die Klasse wie auch alle Attribute als `final` gekennzeichnet.

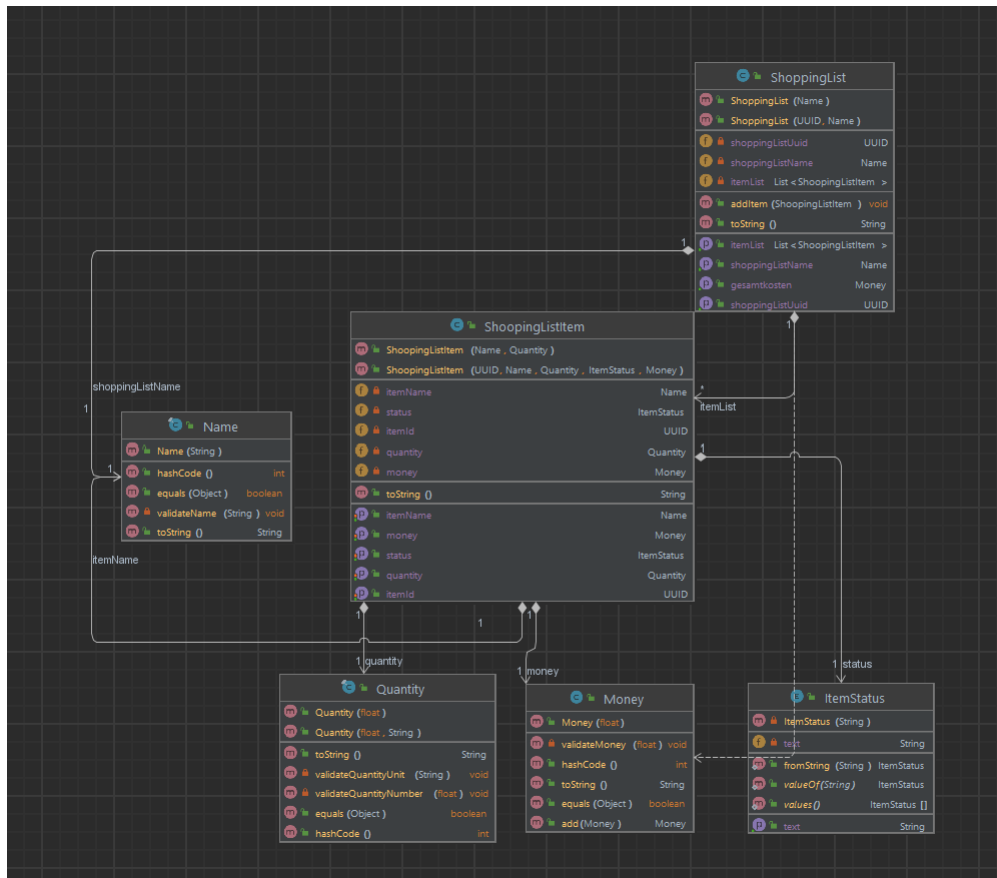
Repositories

Repositories sind dazu da Daten aus dem Speicher zu lesen und der dem Business Code bereitzustellen. Aufgaben des Repositories können das hinzufügen löschen aggregieren oder aktualisieren von Objekten sein. Da die Implementierung stark davon abhängt mit was für einer Technologie Daten persistiert werden, wird meist ein Interface für die Definition der Schnittstellen genutzt. Dieses wird im Domänen Code definiert und in der Plugin Database Schicht implementiert.



Das `ShoppingListRepository` implementiert das `ShoppingListRepositoryInterface`. Es definiert die Methoden und greift über den `CSVReader` oder `CSVWriter` auf die Daten zu. Es kann neue `ShoppingList` hinzufügen, nach diesen filtern oder alle `ShoppingList` zurückgeben.

Aggregates



Im gegebenen Programm wurden keine konkreten Aggregate verwendet.

Jedoch könnte die ShoppingList als eine Art Aggregate gesehen werden. Auf ShoppingListItem kann nur über ShoppingList zugegriffen werden. Ein direkter Zugriff auf Items ist nicht möglich. Die ShoppingList könnte man somit als Aggregate Root bezeichnen. Durch diese hat man Zugriff auf ShoppingListItem und deren Value Objects. Es existiert des Weiteren nur ein ShoppingListRepositoryInterface dadurch kennt der Domänen Code nur diesen Zugriffspfad.

Kapitel 7: Refactoring

Code Smells

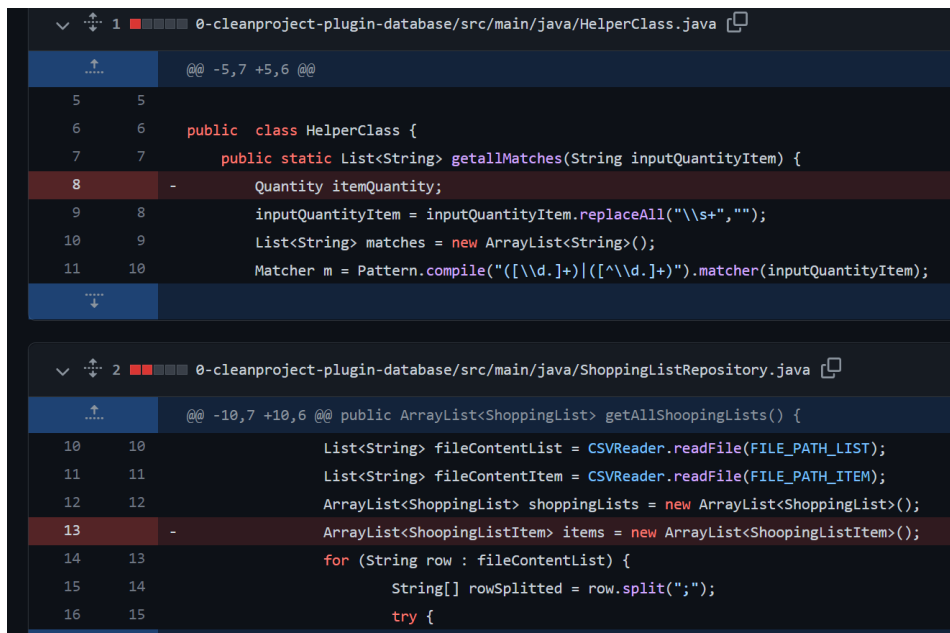
Unused Imports:

[delete unused imports · Rico2000/ASEShoppingList@19ef572 \(github.com\)](https://github.com/Rico2000/ASEShoppingList@19ef572)

```
3 0-cleanproject-plugin-database/src/main/java/AddItemToListAction.java
...
1 - import java.util.ArrayList;
2   1 import java.util.List;
3   2 import java.util.Scanner;
4   - import java.util.regex.Matcher;
5   - import java.util.regex.Pattern;
6   3
```

Dead Code:

[unused code deleted · Rico2000/ASEShoppingList@42ad43a \(github.com\)](#)



```
0-cleanproject-plugin-database/src/main/java/HelperClass.java
@@ -5,7 +5,6 @@
5 5
6 6 public class HelperClass {
7 7 public static List<String> getallMatches(String inputQuantityItem) {
8 - Quantity itemQuantity;
9 8 inputQuantityItem = inputQuantityItem.replaceAll("\\s+", "");
10 9 List<String> matches = new ArrayList<String>();
11 10 Matcher m = Pattern.compile("([\\d.]+)|([^\d.]+)").matcher(inputQuantityItem);

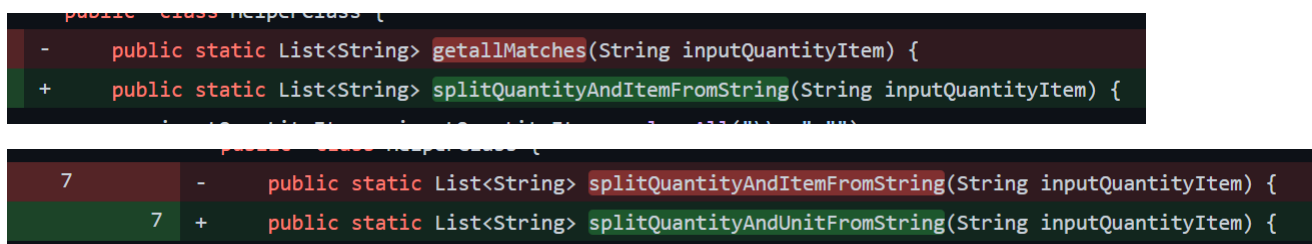
0-cleanproject-plugin-database/src/main/java/ShoppingListRepository.java
@@ -10,7 +10,6 @@ public ArrayList<ShoppingList> getAllShoopingLists() {
10 10 List<String> fileContentList = CSVReader.readFile(FILE_PATH_LIST);
11 11 List<String> fileContentItem = CSVReader.readFile(FILE_PATH_ITEM);
12 12 ArrayList<ShoppingList> shoppingLists = new ArrayList<ShoppingList>();
13 - ArrayList<ShoopingListItem> items = new ArrayList<ShoopingListItem>();
14 13 for (String row : fileContentList) {
15 14 String[] rowSplitted = row.split(";");
16 15 try {
```

2 Refactorings

Rename Method

[Refactoring done · Rico2000/ASEShoppingList@1c83451 \(github.com\)](#)

[fix method name · Rico2000/ASEShoppingList@bb4b453 \(github.com\)](#)



```
public class HelperClass {
- public static List<String> getallMatches(String inputQuantityItem) {
+ public static List<String> splitQuantityAndItemFromString(String inputQuantityItem) {

public class HelperClass {
7 - public static List<String> splitQuantityAndItemFromString(String inputQuantityItem) {
7 + public static List<String> splitQuantityAndUnitFromString(String inputQuantityItem) {
```

Hier wurde eine Methode unbenannt welche keine sprechende Namen hatte. Ziel dieser Methode ist das Splitten einen Strings aufgrund einer Regex. Der erste Teil sollte eine Quantatity repräsentieren, der zweite die Unit. Durch die Umbenennung ist nun klarer was die Methode macht.

Extract Method

[Refactoring done · Rico2000/ASEShoppingList@1c83451 \(github.com\)](#)

Extract this nested try block into a separate method.

 Code Smell **+1**

1 Nesting + 1

alt + ↑ ↓ to navigate issue locations

```
List<String> matches = HelperClass.getAllMatches(inputQuantityItem);
Quantity itemQuantity;

try {
    if (matches.size() == 2) {
        itemQuantity = new Quantity(Float.parseFloat(matches.get(0)), matches.get(1));
    } else {
        itemQuantity = new Quantity(Float.parseFloat(matches.get(0)));
    }
    shoppingListService.addItemToList(shoppingListName, itemName, itemQuantity);
}
catch(Exception e) {
    e.printStackTrace();
}
```

Hier wurde ein Nested Try Catch Block in eine separate Methode ausgelagert. Dadurch ist der Code nun besser lesbar. Des Weiteren ist der Code der extrahiert wurde unabhängig von dem restlichen und stellt eine separate Funktionalität dar. Daher ist die Extraktion in eine Methode eine sinnvolle Verbesserung

Kapitel 8: Entwurfsmuster

Singleton:

Singleton hat das Ziel das von einer Klasse nicht mehr als ein Objekt erstellt werden kann. Dies wird erreicht indem dass Objekt in der Klasse selbst erzeugt wird und als statische Instanz aufrufbar ist. Das Singleton Pattern macht die Klasse in der Anwendung global zugänglich. Es stellt ein sehr einfaches aber dennoch nützliches Pattern dar um die mehrfach Instanziierung von Klassen zu verhindern.

```

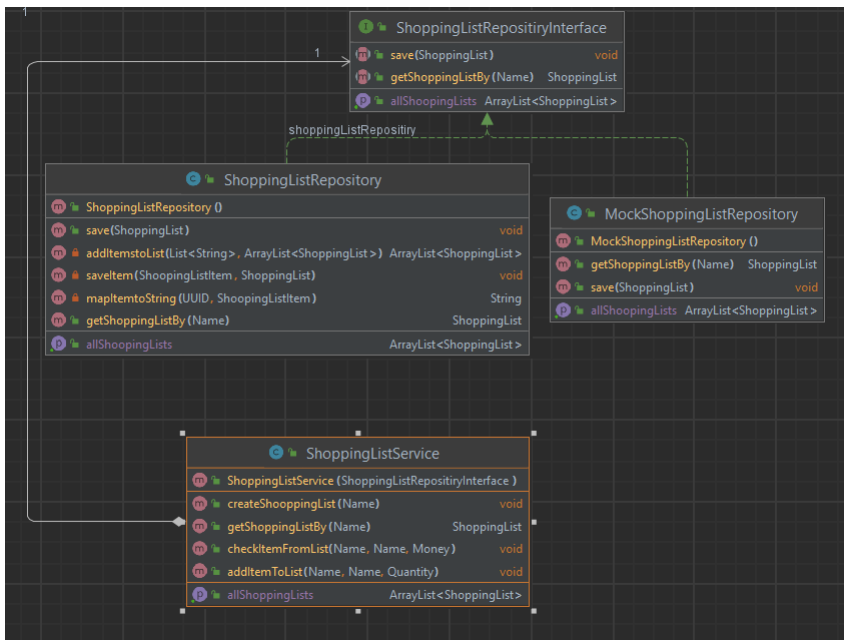
public class CommandLineLogger {
    private static final Logger LOGGER = Logger.getLogger(CommandLineLogger.class.getName());
    1 usage
    private static final CommandLineLogger instance = new CommandLineLogger();
    1 usage
    private CommandLineLogger(){}
    16 usages
    public void log(String text) {
        System.out.println(text);
    }
    16 usages
    public static CommandLineLogger getInstance(){
        return instance;
    }
}

```

CommandLogger ist dazu da die CommandLine Ausgaben darzustellen. Diese Klasse wird an vielen Stellen des Programms verwendet. Aufgrund dessen wird bei einem ersten Aufruf eine Instanz erstellt, welche fortlaufend verwendet wird.

Proxy Pattern:

Im Proxy-Muster repräsentiert eine Klasse die Funktionalität einer anderen Klasse. Diese Art von Entwurfsmuster fällt unter Strukturmuster.



Im Falle der Anwendung ist dies durch das **ShoppingListRepository** und dessen Interface realisiert. Der **ShoppingListService** greift nur auf das Interface zu. Der Service kennt dadurch die genau Implementierung nicht. Das Interface agiert als ein Proxy für das eigentliche Repository. Dadurch wird die Komplexität vor der Außenwelt verborgen.