# Refactoring

Lukas Panni
TINF18B5

DHBW Karlsruhe
Vorlesung Advanced Software Engineering Semester 5/6

# Inhaltsverzeichnis

# 1 Code Smells

Im Folgenden sollen bekannte Code Smells im Code identifiziert werden. Im Abschnitt Refactorings werden Refactorings beschrieben, die einige der Identifizierten Code Smells beheben sollen.

## 1.1 Duplicated Code

*Duplicated Code* beschreibt, dass der gleiche beziehungsweise sehr ähnlicher Code an mehreren Stellen des Systems vorkommt. Dadurch ist der Wartungsaufwand vergleichsweise hoch, da bei jeder Änderung potentiell mehrere Stellen angepasst werden müssen. Das kann auch dazu führen, dass sich die Funktionalität der einzelnen Stellen im Laufe der Zeit minimal unterscheidet, wodurch das Verhalten des Systems inkonsistent wird. Um Duplicated Code zu reduzieren muss der doppelte Code ausgelagert werden und kann dann an verschiedenen Stellen wiederverwendet werden.

- TimeSpanDetails: Click-Listener-Code wird vier mal in gleicher Form (bis auf eine Variable) verwendet. Gelöst mit ExtractMethod bei Commit 0c0b357
- *UserContributionRepository.userContributionsTimeSpan* und *RepositoryDataRepository.repositorySummary* sind, bis auf die verwendeten Datentypen sehr ähnlich. Da beide Klassen von der gleichen Basisklasse erben wäre die Auslagerung einer Methode, in die Basisklasse eine denkbare Lösung.
  Ein erster Schritt zur Verringerung des doppelten Codes ist durch das Refactoring ExtractMethod bei Commit 3b1eb5b bereits umgesetzt.

Ansonsten konnten keine weiteren duplizierten Code-Teile gefunden werden.

## 1.2 Long Method & Large Class

Der Code Smell *Long Method* zeichnet sich durch sehr lange Methoden aus, wobei die Länge, ab der eine Methode als zu lang betrachtet wird, von Projekt zu Projekt variabel sein kann. Lange Methoden erschweren das Verständnis des Codes, was wiederum die Wartbarkeit und auch die Erweiterbarkeit einschränkt. Als Lösung kann die Lange Methode in mehrere kürzere Methoden aufgeteilt werden.

Ähnlich wie Long Method beschreibt *Large Class*, Klassen, die vergleichsweise viele Code-Zeilen beinhalten. Dies kommt häufig vor, wenn mehrere Verantwortlichkeiten in einer Klasse untergebracht werden. Auch hier kann die Verständlichkeit des Codes erschwert werden. Das Aufteilen der Klasse in mehrere Klassen, ist eine sinnvolle Lösung für diesen

Code Smell.

Um sowohl lange Methoden als auch große Klassen zu finden wurde das Code-Statistik Plugin *Statistic* für Android Studio verwendet. Als Obergrenze für die Länge von Klassen werden 200 Zeilen und für Methoden 50 Zeilen Code festgelegt. Es konnte festgestellt werden, dass keine Klasse mehr als 100 Code-Zeilen enthält und keine Methode die 50 erlaubten Zeilen erreicht.

## 1.3 Shotgun Surgery

*Shotgun Surgery* beschreibt, dass für vergleichsweise kleine Funktionale Änderungen Anpassungen an vielen Stellen notwendig sind und deutet auf schlechte Struktur und eine Verflechtung von Verantwortlichkeiten hin. Durch eine Umstrukturierung des Codes, sodass jede Klasse nur eine Verantwortlichkeit hat, kann dies behoben werden.

Dieser Code-Smell ist vergleichsweise schwer zu entdecken, wenn man danach sucht. Stattdessen kann dieser Code-Smell bei Anpassungen des Codes entdeckt werden. Bisher wurde dieser Code-Smell bei keiner Änderung entdeckt.

## 1.4 Switch Statements

Die Verwendung von *Switch Statements* fördert Fehler durch die unintuitive Syntax und verleitet oft dazu das gleiche Switch-Statement an mehreren Stellen einzusetzen. Durch das übermäßige Verwenden von Switch Statements wird die Wartbarkeit und auch die Erweiterbarkeit des Codes eingeschränkt. Switch Statements können in Objektorientiertem Code häufig durch die Verwendung von Polymorphie reduziert werden.

Im gesamten Code konnten keine Switch-Statements entdeckt werden. Außerdem wurde auch das Alternativkonstrukt (lange if-else Verkettungen) nicht entdeckt.

## 1.5 Code Comments

*Code Comments* die beschreiben, was der Code an dieser Stelle tut, deuten häufig darauf hin, dass der Code an dieser Stelle unverständlich geschrieben ist.

Durch die Suche nach Kommentaren und eine Beurteilung der Kommentare in Bezug auf diesen Code-Smell ergab den folgenden Kommentar in den Klassen *UserContributionsRepository* und *RepositoryDataRepository*, der beschreibt, was der Code an dieser Stelle tut, was auf unverständlichen Code hindeutet.

```
//Wrap callback to add response to cache
ClientDataCallback decoratedCallback = new
   ClientDataCallbackDecorator(callback, response ->
   cache.put(repository, response));
```

Der Code an dieser Stelle ist ohne den Kommentar schwer verständlich. Um die Verständlichkeit des Codes zu erhöhen wurde das Refactoring ExtractMethod bei Commit 3b1eb5b angewendet und die extrahierte Methode in die Basisklasse verschoben.

# 2 Refactorings

Die hier beschriebenen Refactorings sollen das Design des Systems verbessern, die Wartbarkeit und Erweiterbarkeit verbessern und auch die Verständlichkeit erhöhen. Dadurch soll es unter anderem einfacher werden Fehler zu finden und zu beheben sowie neue Funktionen hinzuzufügen.

## 2.1 ExtractMethod bei Commit 0c0b357

In der Klasse *TimeSpanDetails* wurde sehr ähnlicher Code für einen Click-Listener an vier unterschiedlichen Stellen verwendet. Durch die Auslagerung in die Methode *getClickListener* kann der Duplicated Code vermieden werden. Durch die Einführung des Parameters *resource* kann die extrahierte Methode flexibel an allen Stellen wiederverwendet werden. Dadurch wird außerdem die Lesbarkeit des Codes erhöht. Auch eventuelle spätere Änderungen am Verhalten der Methode müssen so nur an einer Stelle durchgeführt werden.

**Code Vorher:**

```
[...]
view.findViewById(R.id.to_commit_repos)
    .setOnClickListener(v -> Navigation.findNavController(view)
    .navigate(R.id.action_1, getArguments()));
view.findViewById(R.id.to_issue_repos)
    .setOnClickListener(v -> Navigation.findNavController(view)
    .navigate(R.id.action_2, getArguments()));
[...]
```

**Code Nacher:**

```
[...]
view.findViewById(R.id.to_commit_repos)
    .setOnClickListener(getClickListener(view, R.id.action_1));
view.findViewById(R.id.to_issue_repos)
    .setOnClickListener(getClickListener(view, R.id.action_2));
[...]
```

## 2.2 ExtractMethod bei Commit 3b1eb5b

In den Klassen *UserContributionsRepository* und *RepositoryDataRepository* wurde doppelter und zusätzlich schlecht verständlicher Code verwendet. Um die Verständlichkeit des Codes zu erhöhen wurde eine Methode extrahiert und so benannt, dass die Funktion leicht verständlich ist. Die extrahierte Methode wurde in die Basisklasse verschoben, da diese Methode in allen Ableitungen der Basisklasse verwendet wird. Diese Änderung ist vergleichsweise klein erhöht jedoch die subjektive Verständlichkeit enorm.

**Code Vorher:**

```
ClientDataCallback decoratedCallback = new
   ClientDataCallbackDecorator(callback, response ->
   cache.put(repository, (RepositoryDataResponse) response));
```

**Code Nacher:**

```
ClientDataCallback decoratedCallback = new
   ClientDataCallbackDecorator(callback,
   getAddToCacheCallback(repository));
```

### 2.2.1 UML Vorher

Abbildung 1 zeigt das UML-Klassendiagramm vor dem Refactoring.

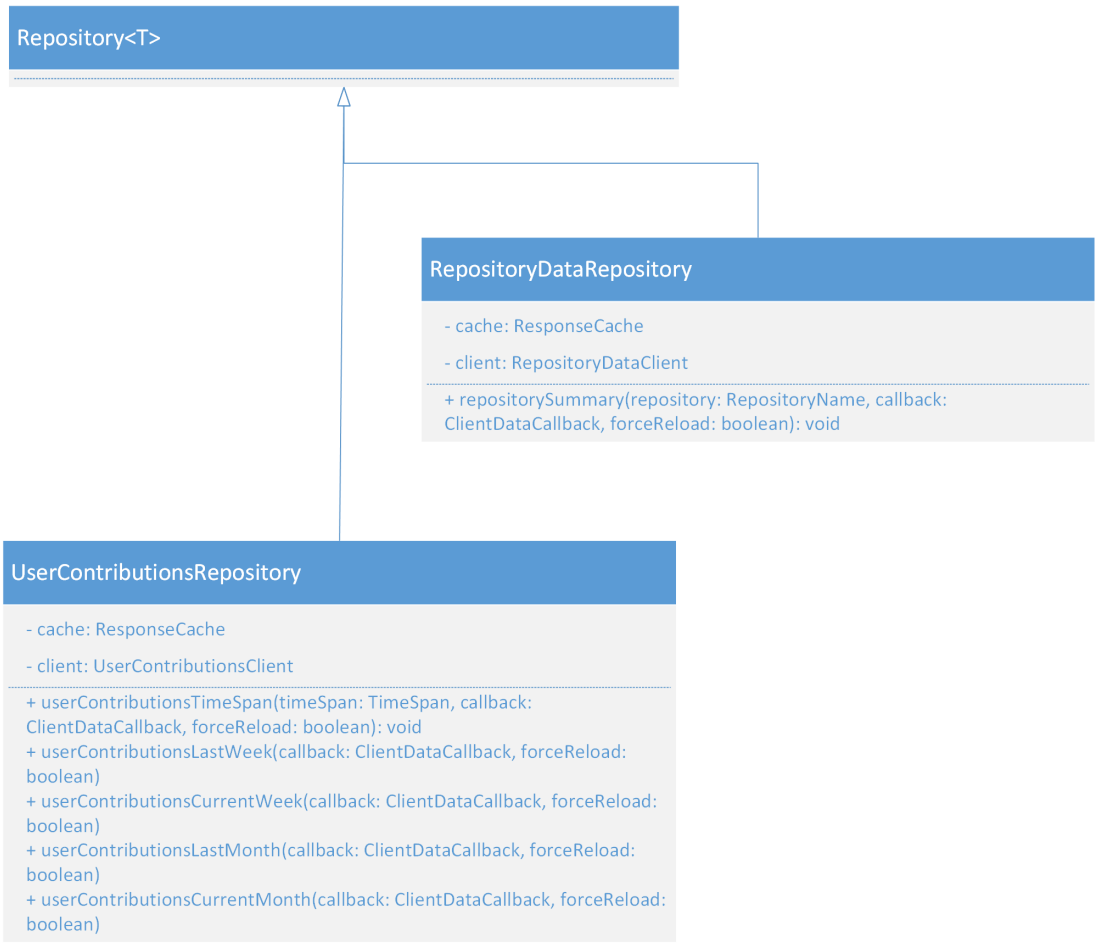


Abbildung 1: UML vor Refactoring

### 2.2.2 UML Nacher

Abbildung 2 zeigt das UML-Klassendiagramm nach dem Refactoring.

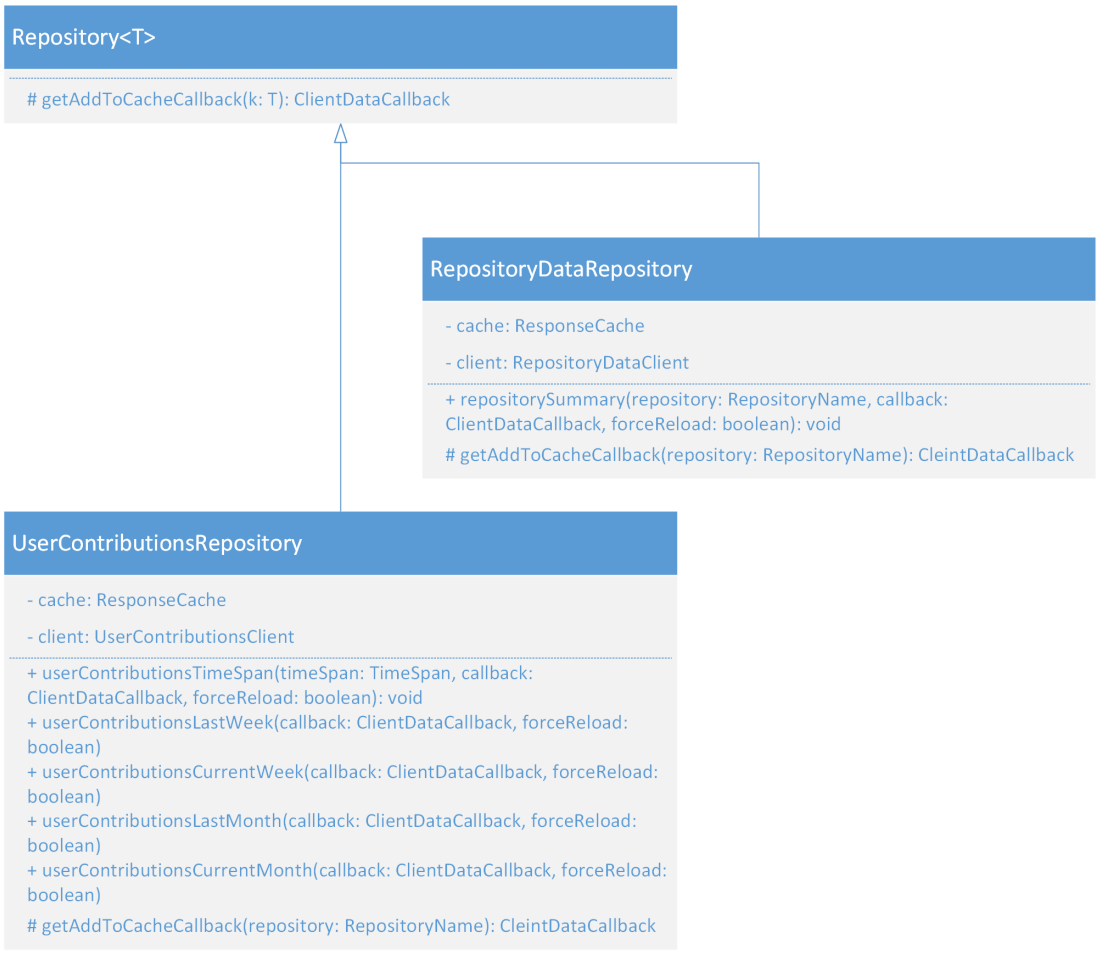


Abbildung 2: UML nach Refactoring

## 2.3 ExtractMethod bei Commit 2eedf85

Um die Singleton-Eigenschaft der Klasse *AuthHandler* entfernen zu können, muss ein Teil der *getInstance()* Methode in eine andere Methode übertragen werden. Dafür wird das Refactoring **ExtractMethod** angewendet. So kann kann das Verhalten wiederverwendet werden. Teile des Verhaltens der *getInstance()* Methode sind dadurch noch erhalten, auch wenn die ursprüngliche Methode in einem späteren Commit entfernt wird.