# Dokumentation Programmentwurf
# Vorlesung Advanced Software Engineering
# DHBW Karlsruhe Semester 5/6

Lukas Panni
TINF18B5

9. März 2021

# Inhaltsverzeichnis

# 1. Legacy Code

## 1.1 Abhängigkeiten brechen

Im folgenden sollen einige Beispiele aufgeführt werden, bei denen Abhängigkeiten mithilfe der in der Vorlesung behandelten Techniken beseitigt werden. Ohne diese Abhängigkeiten sind Tests leichter zu entwickeln, sodass weitere Änderungen, zum Beispiel zur Verbesserung des Designs, später einfacher und komfortabler möglich sind. Das Ziel beim Brechen der Abhängigkeiten ist es, die Testbarkeit der betroffenen Klassen beziehungsweise Methoden zu erhöhen.

### 1.1.1 ExtractInterface bei Commit 1e47e7b

#### Ausgangszustand

Die Klasse *AuthHandler* ist stark abhängig von der Android-Klasse *Activity* und kann auch nicht erstellt werden ohne eine Instanz dieser Klasse der *getInstance*-Methode zu übergeben. Die Testbarkeit ist schlecht, da eine Android-*Activity*-Instanz nicht ohne weiteres im Test-Kontext erstellt werden kann. Auch ein Fake-Objekt ist schwer zu erstellen, da eine finale Methode genutzt wird, in Ableitungen von *Activity* nicht überschrieben werden kann. Da die Klasse *AuthHandler* für die OAuth-Authentifizierung verantwortlich ist und damit wichtig für das Gesamtsystem, ist es wichtig, Tests für diese Klasse zu ermöglichen.

#### Gewählte Technik

Die *getInstance*-Methode benötigt eine Instanz der Klasse *Activity*, die im Test-Kontext nicht leicht erstellbar ist und für die auch Fake-Objekte nur schwer erstellt werden können. Da allerdings nur wenige Methoden der *Activity*-Klasse verwendet werden, und eine Änderung der *AuthHandler* Klasse nur vergleichsweise wenige Änderungen erfordert, kann in diesem Fall **ExtractInterface** angewendet werden. So wird die Abhängigkeit von *AuthHandler* zu *Activity* gelöst, indem das Interface *AuthHandlerActivity* eingeführt und der Parameter von *getInstance* angepasst wird, sodass eine Instanz vom Typ des Interfaces verwendet wird. Dabei werden auch kleinere Anpassungen an Methodenaufrufen vorgenommen, sodass Methoden des Interfaces aufgerufen werden. **ExtractInterface** ist durch IDE-Unterstützung vergleichsweise einfach und ohne große Fehleranfälligkeit durchzuführen. Außerdem wird die Abstraktion durch diese Technik verbessert.

#### Endzustand

*AuthHandler* nutzt nach dieser Änderung nur noch die Methoden des Interfaces, wodurch die Testbarkeit erhöht wird. Allerdings gibt es hier das Problem, dass *AuthHandler* die Third-Party-Klasse *AuthorizationService* verwendet, die auf eine *Activity*-Instanz angewiesen ist. Deshalb gibt es im neu eingeführten Interface eine Methode, die eine *Activity* zurückgibt. Die Abhängigkeit konnte also in diesem Fall nicht komplett aufgelöst werden. Trotzdem wurde die Testbarkeit erhöht und die Abstraktion verbessert. Diese Änderung stellt damit eine gute Basis für weitere Refactorings und Änderungen dar, wie zum Beispiel in Commit cbc5318.

### 1.1.2 Parametrize Constructor bei Commit d098b93

#### Ausgangszustand

Die Klassen *RepositoryDataRepository* und *UserContributionsRepository* haben eine Abhängigkeit zur Klasse *ResponseCache* und erzeugen im Konstruktor eine Instanz dieser Klasse und speichern diese in einer Instanzvariablen. Für den Test der Klassen ist es hilfreich, diese Instanzvariable durch eine eigene Cache-Implementierung zu nutzen. In diesem Fall lässt sich die Abhängigkeit mit Hilfe von **Parametrize Constructor** brechen.
Dazu wird ein neuer Konstruktor erzeugt, der einen zusätzlichen Parameter vom Typ der zu überschreibenden Instanzvariable entgegennimmt. In diesem Beispiel eine Instanz von *ResponseCache*. Die existierenden Konstruktoren rufen den neuen Konstruktor mit dem ursprünglichen Wert der Variable auf. Dabei bleibt die alte Funktionalität erhalten und es müssen keine Anpassungen an anderen Stellen erfolgen.

#### Endzustand

Durch das Brechen der Abhängigkeit ist es nun möglich, beim Erstellen einer Instanz vom Typ *RepositoryDataRepository* oder *UserContributionsRepository* ein *ResponseCache*-Objekt zu übergeben. Dadurch kann zum Beispiel beim Test ein Fake-Objekt übergeben werden.

### 1.1.3 ExtractInterface bei Commit da964f5

#### Ausgangszustand

Ein Objekt der Klasse *AuthHandler* wird zum Beispiel in der Klasse *GHClient* (bei da964f5 umbenannt in *GithubOAuthClient*) benötigt, um Daten über die API abrufen zu können. Aktuell ist ein Test dieser Komponente nur schwer möglich, da *AuthHandler* selbst wiederum Abhängigkeiten zu einer Third-Party Bibliothek besitzt. Deshalb ist hier das erstellen von Fake-/Mock-Objekten sehr aufwändig. Bei 1.1.1 wurde bereits die Testbarkeit der *AuthHandler*-Klasse erhöht und durch ExtractInterface an dieser Stelle soll die Testbarkeit der von *AuthHandler* abhängigen Klassen erhöht werden.

#### Gewählte Technik

Es wurde **ExtractInterface** gewählt um Abhängigkeiten zu *AuthHandler* zu beseitigen. Dazu wurde das Interface *AuthenticationHandler* eingeführt wird, das in der alten *AuthHandler*-Klasse (zur besseren Verständlichkeit umbenannt zu *GithubOAuthHandler*) implementiert wird. Um den Vorgang zu ermöglichen wurde zuvor die Singleton-Eigenschaft entfernt (13efee8) und die Methoden und ihre Verwender wurden angepasst (473384c). Anschließend kann das Interface erstellt werden und bei allen Verwendern der Ursprünglichen Klasse, die dies erlauben, die Abhängigkeit zur Konkreten Klasse durch eine Abhängigkeit zum Interface ersetzt werden.

#### Endzustand

In der Klasse *GithubOAuthClient* konnte die Abhängigkeit zu *GithubOAuthHandler* (vormals *AuthHandler* vollständig beseitigt werden. Dadurch werden Tests der Klasse mit Fake-/Mock-Implementierungen des neuen Interfaces *AuthenticationHandler* möglich. Außerdem wird die Abstraktion verbessert und die konkrete Authentifizierungs-Klasse könnte leichter ausgetauscht werden, was zum Beispiel benötigt wird, wenn ein anderes OAuth-Framework eingesetzt werden sollte.

### 1.1.4 Parametrize Method bei Commit d8c995c

#### Ausgangszustand

Die Klasse *TimeSpanFactory*, die verwendet wird um *TimeSpan*-Objekte zu erzeugen hat eine starke Abhängigkeit zur Java-Klasse *java.util.Calendar*. In den Methoden von TimeSpanFactory wird jeweils eine Calendar-Instanz benötigt. Um diese zu erhalten wird die Methode *Calendar.getInstance()* verwendet. Da die verwendete Calendar-Instanz das Ergebnis der Methoden beeinflusst muss die Instanz für Tests ausgetauscht werden können. Dies ist aber aktuell nicht möglich.

#### Gewählte Technik

Um die Abhängigkeit der Methoden zu *Java.util.Calendar* zu beseitigen wurde die Technik **Parametrize Method** gewählt. Bei dieser Technik wird für die Methode, bei der die Abhängigkeit gebrochen werden muss, eine neue Methode mit einem zusätzlichen Parameter angelegt. Dieser Parameter ersetzt dann die Variable, die im Test benötigt wird. Die eigentliche Funktionalität der alten Methode wird in die neue Methode verschoben und die alte Methode ruft diese mit dem ursprünglichen Wert der Variablen auf. So sind keine Änderungen an bestehendem Code notwendig und trotzdem wird ein Test der Methode ermöglicht.

#### Endzustand

Die Technik Parametrize Method wurde für mehrere Methoden in TimeSpanFactory angewendet um die Abhängigkeit zu lösen, sodass Tests nun möglich sind. Durch die Möglichkeit eine Calendar-Instanz zu übergeben können die Methoden einfach getestet werden.

### 1.1.5 Subclass and Override Method bei Commit 13676e5

#### Ausgangszustand

Die Klasse *GithubOAuthClient* verwendet eine Third-Party-Library um GraphQL-Anfragen an die GitHub-API durchzuführen. Diese Abhängigkeit macht Tests vergleichsweise schwierig. Um also Tests durchführen zu können ist es notwendig, diese Abhängigkeit zu lösen.

#### Gewählte Technik

Um diese Abhängigkeit zu lösen wurde die Technik **Subclass and Override Method** gewählt. Diese Technik erlaubt es, eine Methode, die im Test nicht sichtbar ist, zu ändern. So kann zum Beispiel der Rückgabewert der Methode für den Test festgelegt werden. Dafür wird zunächst die bestehende Methode umbenannt und eine neue Methode mit der gleichen Signatur der eben umbenannten Methode angelegt. Diese neue Methode muss für Ableitungen der Klasse sichtbar sein und kann dort überschrieben werden. In der Basisklasse werden alle Aufrufe der alten Methode durch Aufrufe der neuen Methode ersetzt, die selbst wiederum die alte Methode aufruft.
Konkret wurde die private Methode *GithubOAuthClient.getGraphqlClient* umbenannt zu *GithubOAuthClient.getGraphqlClientInternal* und eine neue protected Methode *GithubOAuthClient.getGraphqlClient* erstellt. So ist es möglich in Ableitungen der Klasse *GithubOAuthClient* das Verhalten der Methode *GithubOAuthClient.getGraphqlClient* zu verändern.

#### Endzustand

Durch Subclass and Override Method wurde es möglich gemacht, die Klasse *GithubOAuthClientTestImplementation* zu erstellen, die *getGraphqlClient* so überschreibt, dass eine Mock-Instanz zurückgegeben werden kann. So wird die Abhängigkeit zur Third-Party-Library reduziert und die Entwicklung von Tests fällt leichter.

# 2. Entwurfsmuster

In diesem Kapitel soll exemplarisch die Verwendung von klassischen Entwurfsmustern gezeigt und begründet werden. Außerdem wird die Verwendung der Entwurfsmuster jeweils mit einem UML-Klassendiagramm verdeutlicht.
Allgemein sind Entwurfsmuster wiederverwendbare und generalisierte Lösungen für häufige Problemstellungen. Durch die Generalisierung müssen Entwurfsmuster teilweise für konkrete Anwendungsfälle angepasst werden, sodass die endgültige Lösung vom eigentlichen Entwurfsmuster abweichen kann. Deshalb werden in den folgenden Abschnitten insbesondere auch die Abweichungen vom Muster in Reinform dokumentiert.

## 2.1 Composite

Das **Composite** ist ein Strukturmuster, das es dem Verwender erlaubt, ein komplexes Objekt genauso zu verwenden wie ein einfaches Objekt. Ein komplexes Objekt kann dabei aus einem oder mehreren Objekten bestehen, wobei ein solches Kind selbst wieder ein komplexes Objekt sein kann. Im Gegensatz dazu besteht ein einfaches Objekt nicht aus mehreren Kindobjekten. So können zum Beispiel auch Baumstrukturen abgebildet werden. Der große Vorteil des Musters ist, dass der Verwender nicht unbedingt wissen muss, ob es sich um ein einfaches oder komplexes Objekt handelt [1, pp. 142–155].
Das hier aufgeführte Anwendungsbeispiel umfasst das Interface *ClientDataCallback* und die Klasse *ClientDataCallbackComposite*, die das Interface implementiert. Das Interface stellt eine Callback-Methode bereit, die ein Objekt vom Typ *ResponseData* entgegennimmt. Das ClientDataCallbackComposite wurde eingeführt, um es zu ermöglichen, mehrere Callback-Methoden mit einem Funktionsaufruf ausführen zu können. Dies wird insbesondere benötigt, um die ResponseData-Objekte mit denen die Callback-Methoden aufgerufen werden im lokalen Cache ablegen zu können. Die Funktionalität, ein Objekt in den Cache einzufügen wird demnach als einfache Callback-Methode übergeben und kann mit einer anderen Callback-Methode zusammen in einem Objekt gehalten werden.
Da das Composite-Muster dem Decorator-Muster strukturell sehr ähnlich ist, wurde die Klasse ClientDataCallbackDecorator auch erst mit Commit 2198284 in ClientDataCall-

backComposite umbenannt, um klarer zu machen, was der Zweck dieser Klasse ist. Ein Decorator wird eher dazu verwendet einem Objekt zur Laufzeit neues Verhalten hinzuzufügen, anstatt ein komplexes Objekt aus mehreren Einzelobjekten zusammenzusetzen [1, pp. 155-169]. Da der Zweck der Anwendung des Musters, gezeigt in Abbildung 2.1, ist, beliebig viele Callbacks mit nur einem Methodenaufruf ausführen zu können, wurde hier die Bezeichnung Composite gewählt. Durch die sehr große strukturelle Ähnlichkeit der Muster könnte man in diesem einfachen Fall aber auch für die Benennung Decorator argumentieren.

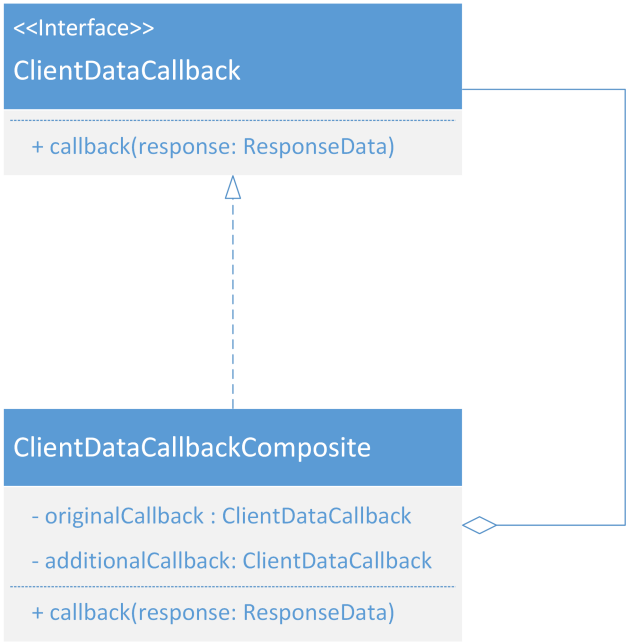


Abbildung 2.1: Umsetzung des Composite Patterns

## 2.2 Strategy

Das **Strategy**-Muster ist ein Verhaltensmuster, das dazu dient, Algorithmen austauschbar zu machen. Der Hauptvorteil dabei ist, dass die Algorithmen zur Laufzeit beliebig ausgetauscht werden können. Realisiert wird das Muster, indem eine Funktionalität, die mit mehreren Algorithmen umgesetzt werden kann, und die eigentlichen Algorithmen voneinander getrennt werden. Dabei wird für jeden Algorithmus eine Klasse erzeugt, die von einer gemeinsamen Basisklasse erbt, oder ein gemeinsames Interface implementiert. So können die verschiedenen Algorithmen in gleicher Art und Weise verwendet werden. Im ursrpünglichen Muster gibt es eine Kontextklasse, die mehrere der Implementierungen enthalten kann, oder diese zumindest kennen muss, um dann zur Laufzeit zu entscheiden, welche Implementierung tatsächlich genutzt wird [1, pp. 343–351].
Abbildung 2.2 zeigt ein UML-Klassendiagramm der Umsetzung dieses Musters. Zur besseren Verständlichkeit der Abbildung wurde darauf verzichtet, alle Parameter und Rückgabetypen vollständig anzugeben.
Die verschiedenen Algorithmen, oder auch Strategien, werden von der abstrakten Klasse *ContributionCountRanking* abgeleitet. Dadurch müssen die abgeleiteten Klassen die Methode *rank* implementieren. Diese Methode nimmt eine Liste aus ContributonCount-Objekten entgegen und gibt auch eine solche Liste mit bestimmter Rangfolge (Sortierung) zurück. Wie diese Rangfolge bestimmt wird, unterscheidet sich zwischen den konkreten Implementierungen.
Für den konkreten Anwendungsfall wurde das Muster leicht abgewandelt. Anstatt einer Kontextklasse, ist im Diagramm nur ein abstrakter Aufrufer eingezeichnet. Dieser kann für eine beliebige Klasse stehen, die die Funktionalität der ContributionCountRanking-Implementierungen verwenden möchte. Im UML-Diagramm ist zusätzlich die Klasse *ContributionCountAnalyzer* abgebildet, deren Methoden *getTop1* und *getTopK* jeweils ein ContributionCountRanking Objekt als Parameter erhalten. So kann diese Klasse Top-K Anfragen durchführen, wobei die Sortierweise vom Aufrufer vorgegeben wird.

Das doch recht komplexe Strategy-Muster wurde in diesem Fall verwendet, um es zu ermöglichen, die Sortierweise anzupassen. Die Sortierweise kann allerdings auch einfacher über Java-Comparator Objekte beeinflusst werden. Trotzdem wurde hier das Strategy-Muster gewählt, da es deutlich mehr Flexibilität bietet und geplante zukünftige Erweiterungen um deutlich komplexere Sortierweisen sehr einfach ermöglicht.

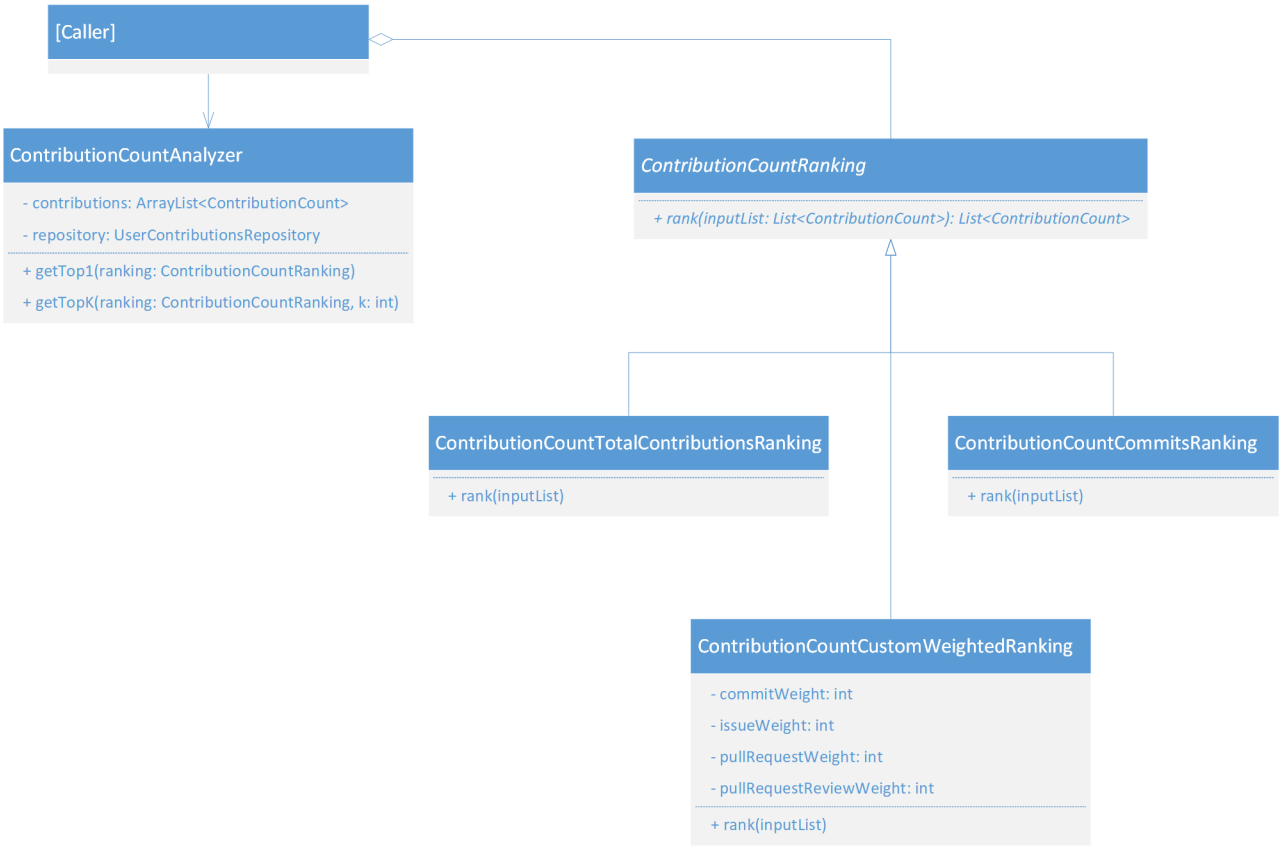


Abbildung 2.2: Umsetzung des Strategy-Patterns

# 3. Refactoring

## 3.1 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.

### 3.1.1 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. Die folgende Liste enthält Beispiele für Duplicated Code im vorliegenden Projekt. Die meisten Stellen wurden bereits behoben.

- 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.
- Der Konstruktor in *ContributionCount* enthielt mehrere gleiche Codeteile, die in Commit 34691d7 beseitigt wurden

- Auch in der Klasse *TimeSpanFactory* wurden mehrere Methoden gefunden, die zu Teilen aus dem gleichen Code bestehen. Dies wurde ebenfalls in Commit 34691d7 beseitigt.

Durch die Code-Inspection von AndroidStudio konnten nicht alle der aufgeführten Dopplungen identifiziert wurden. Mit Hilfe von IntelliJ IDEA Ultimate konnten weitere Dopplungen identifiziert und beseitigt werden. Nach Beseitigung der genannten Dopplungen konnten mit der Inspection-Rule *Duplicated code fragment* keine weiteren Dopplungen im Produktivcode gefunden werden. Da nur wenige Dopplungen gefunden wurden und nur noch eine Kopplung weiterhin besteht, kann abschließend gesagt werden, dass der Code weitgehend frei von unnötigen Duplizierungen ist.

### 3.1.2 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. Um große Klassen oder Methoden zu finden, muss zunächst eine Obergrenze für die Methodenlänge fesgelegt werden. Legt man zusätzlich eine maximale Anzahl von Methoden pro Klasse fest, lässt sich eine maximale Klassenlänge ableiten. Im Buch *Refactoring in Large Software Projects* wird angegeben, dass Methoden im Durchschnitt nicht länger als 30 Zeilen sein sollten [2, p. 31]. Außerdem wird beschrieben, dass eine Klasse weniger als 30 Methoden und damit auch weniger als 900 Zeilen umfassen sollte. Die Längenvorgabe für Methoden soll auch hier verwendet werden. Eine Klasse mit 900 Zeilen Code, ist aber meiner Meinung nach zu unübersichtlich für den praktischen Gebrauch. Stattdessen sollte eine Klasse maximal etwa 200 Zeilen Code umfassen. So kann der Code der Klasse noch schnell erfasst und verstanden werden und gleichzeitig wird eine zu starke Fragmentierung in viele einzelne Datein vermieden.

Aus diesem Grund sollen alle Klassen kleiner als 200 Zeilen Code sein.
Abbildung 3.1 zeigt die Länge der längsten Klassen im Projekt. Keine der Klassen erreicht die maximale Anzahl an Zeilen. Um die Länge der Methoden festzustellen wurden alle Klassen mit einer Länge von über 30 Zeilen untersucht. Die Längste gefundene Methode besteht aus 28 Code-Zeilen und ist damit knapp kürzer als erlaubt.

| Source File | Total Lines | Source Code Lines |
|---|---|---|
| GithubOAuthHandler.java | 128 | 96 |
| GithubOAuthClient.java | 109 | 94 |
| RepositoryDetailsCard.java | 108 | 89 |
| MockApolloQueryCallFactory.java | 109 | 85 |
| ProgressCard.java | 99 | 78 |
| RepositoryListFragment.java | 99 | 78 |

Abbildung 3.1: Größte Klassen des Projekts

### 3.1.3 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.

### 3.1.4 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.

### 3.1.5 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.

## 3.2 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.

### 3.2.1 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));
[...]
```

### 3.2.2 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));
```

### UML Vorher

Abbildung 3.2 zeigt das UML-Klassendiagramm vor dem Refactoring.

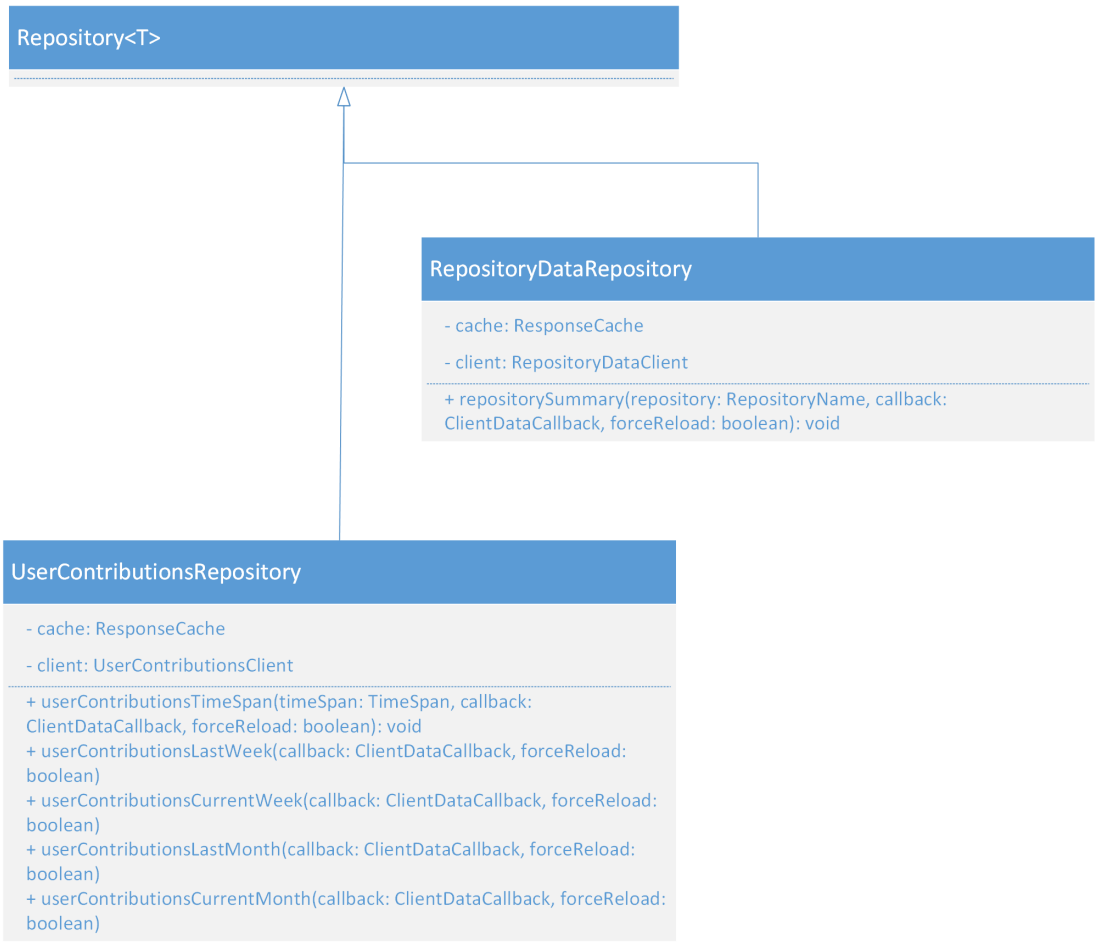


Abbildung 3.2: UML vor Refactoring

### UML Nacher

Abbildung 3.3 zeigt das UML-Klassendiagramm nach dem Refactoring.

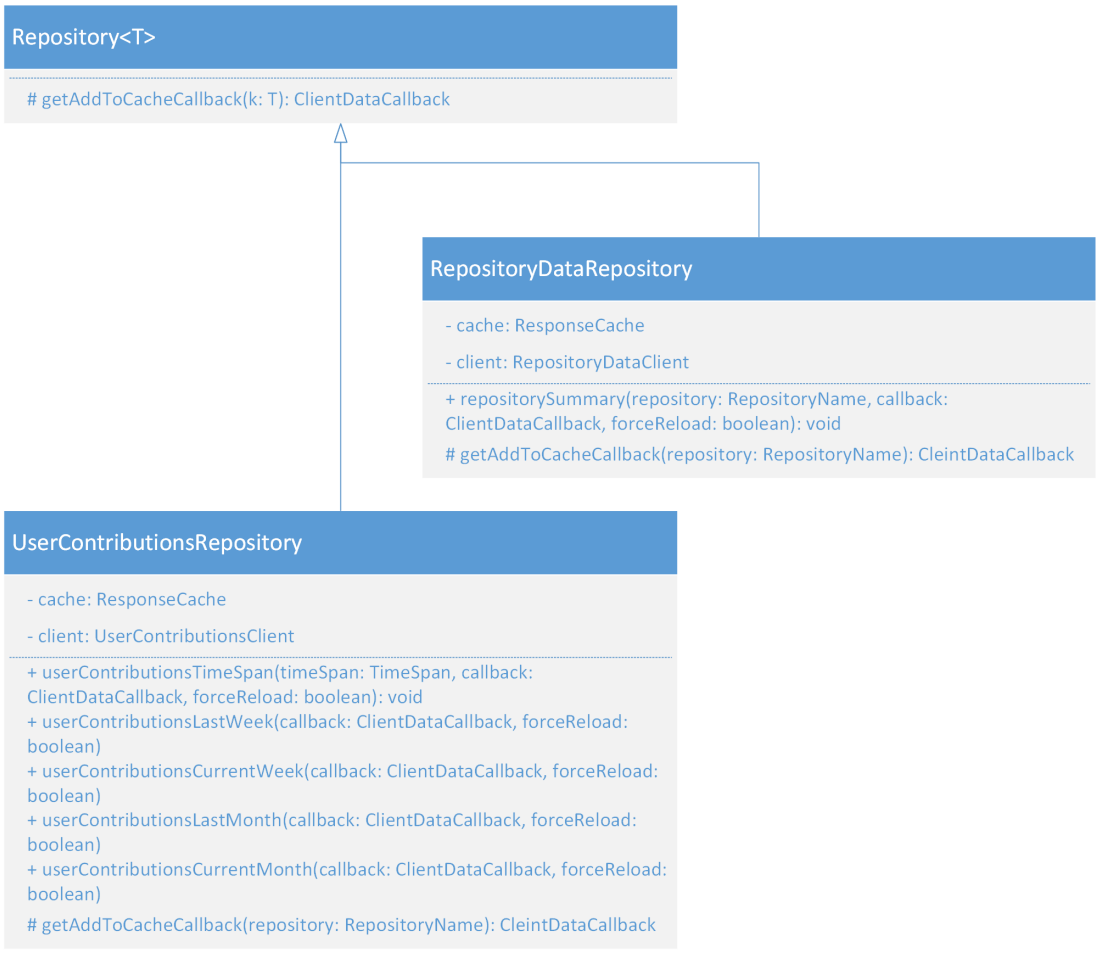


Abbildung 3.3: UML nach Refactoring

### 3.2.3 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.

### 3.2.4 RenameMethod bei Commit 650235a

In den beiden Klassen *RepositoryDataRepository* und *UserContributionsRepository* werden Methodennamen verwendet, die den Zweck der Methode nur schwer verständlich machen. Um dies zu beheben wurde das Refactoring **ExtractMethod** angewendet.

#### RepositoryDataRepository

Die Methode *RepositoryDataRepository.repositorySummary* wurde umbenannt in *RepositoryDataRepository.loadRepositoryData*, da dieser Name die Funktion der Methode deutlich besser beschreibt. Der alte Methodenname beschreibt nur den Zusammenhang mit einer Zusammenfassung über ein Repository, zeigt aber nicht deutlich genug, dass in dieser Methode Daten zu einem übergebenen Repository geladen werden. Auch war aufgrund des Methodennamens die Funktion des Parameters *forceReload* unklar. Mit dem neuen Methodennamen sollen der Code deutlich leichter verständlich sein.

#### UserContributionsRepository

In dieser Klasse wurden mehrere ähnliche Methoden umbenannt:

- *UserContributionsRepository.userContributionsTimeSpan* umbenannt in *UserContributionsRepository.loadUserContributionsInTimeSpan*
- *UserContributionsRepository.userContributionslastWeek* umbenannt in *UserContributionsRepository.loadUserContributionsInLastWeek*
- *UserContributionsRepository.userContributionsCurrentWeek* umbenannt in *UserContributionsRepository.loadUserContributionsInCurrentWeek*
- *UserContributionsRepository.userContributionslastMonth* umbenannt in *UserContributionsRepository.loadUserContributionsInLastMonth*
- *UserContributionsRepository.userContributionsCurrentMonth* umbenannt in *UserContributionsRepository.loadUserContributionsInCurrentMonth*

Wie bereits in der Klasse *RepositoryDataRepository* soll der Code durch die Umbenennungen besser verständlich sein, indem die neuen Methodennamen die Funktion der Methoden besser beschreiben.
In Commit 9385bb4 wurde zusätzlich eine Umbenennung in der Klasse *GithubOAuthClient* vorgenommen um die Konsistenz zu den neuen Methodennamen der Klasse *RepositoryDataRepository* zu steigern. Konkret wurde die Methode *GithubOAuthClient.repositoryData* zu *GithubOAuthClient.loadRepositoryData* umbenannt. Durch diese kleine Änderung wird auch der Zusammenhang von *GithubOAuthClient.loadRepositoryData* und *RepositoryDataRepository.loadRepositoryData* besser deutlich.

### UML Vorher

Abbildung 3.4 zeigt das UML-Klassendiagramm vor der Umbenennung der Methoden.

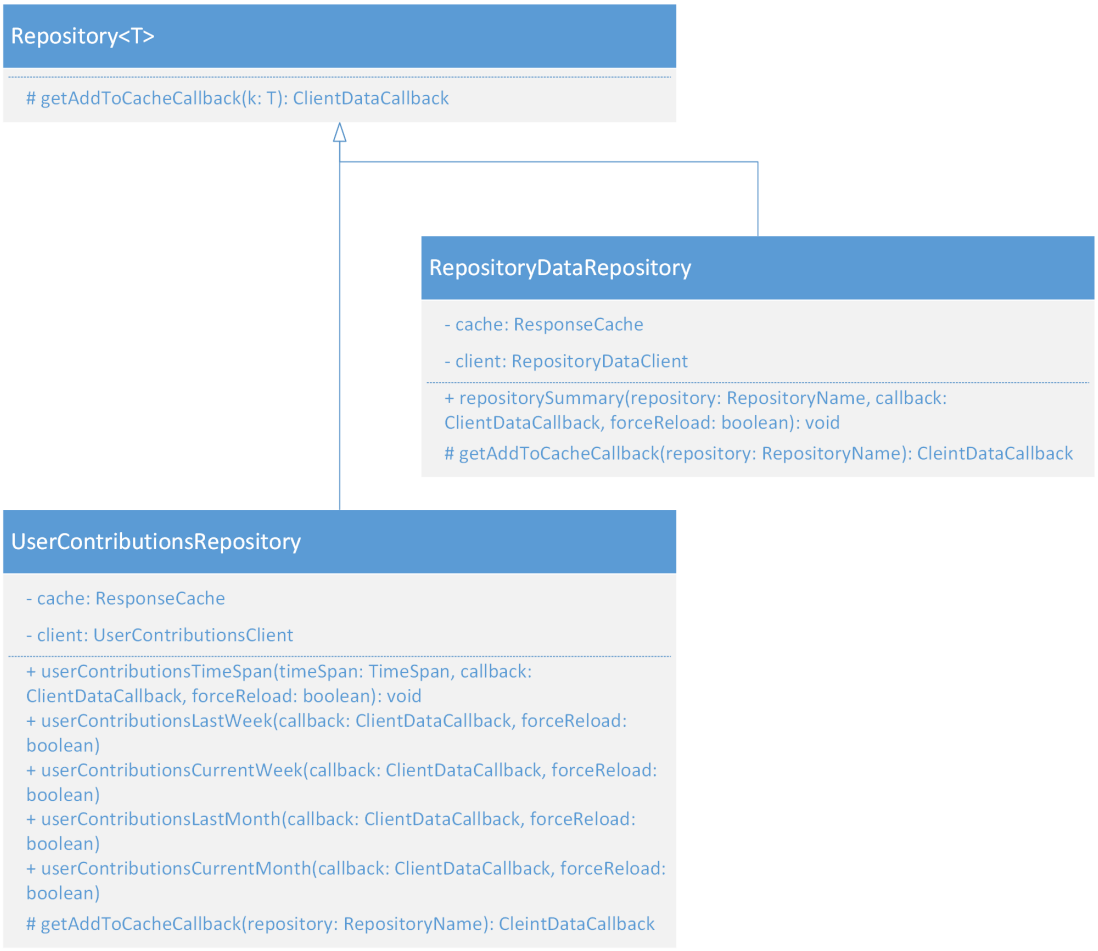


Abbildung 3.4: UML vor Refactoring

### UML Nacher

Abbildung 3.5 zeigt das UML-Klassendiagramm nach der Umbenennung der Methoden in *UserContributionsRepository* und *RepositoryDataRepository*.

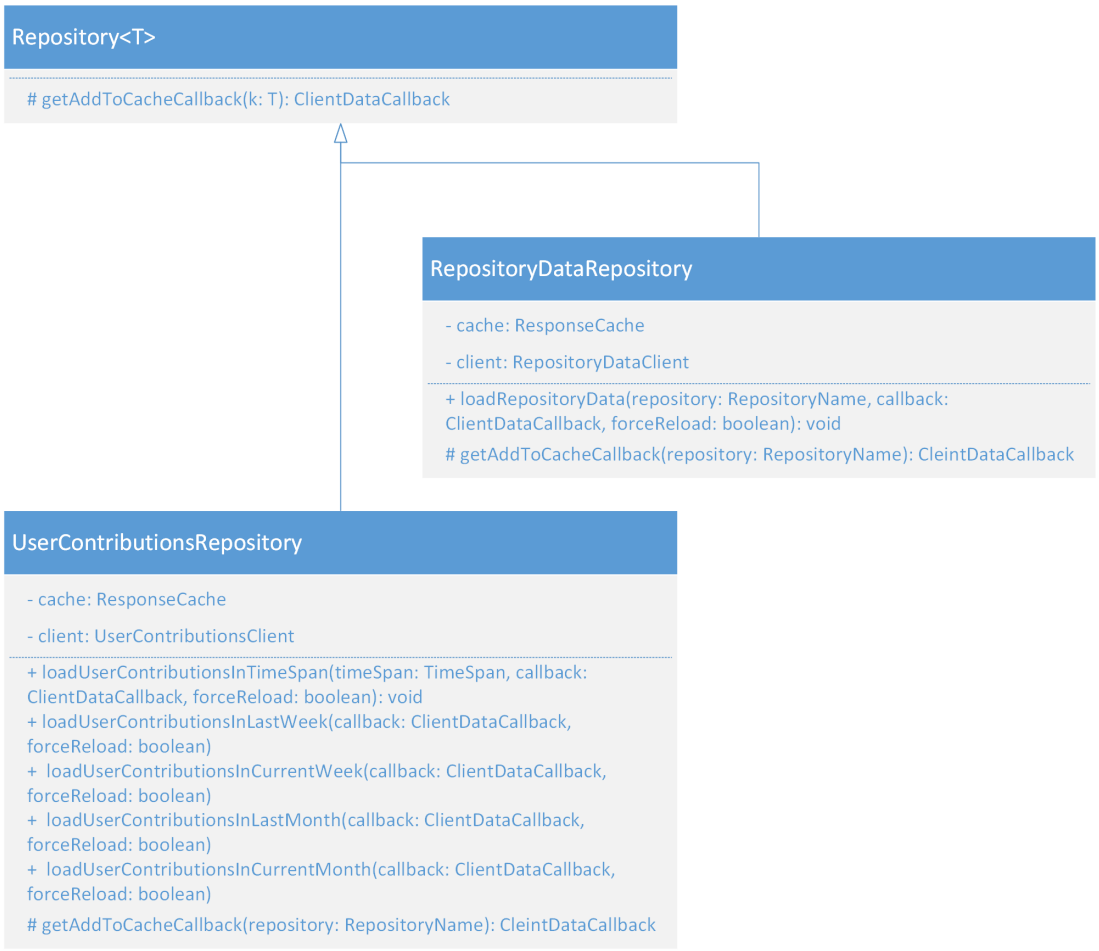


Abbildung 3.5: UML nach Refactoring

# 4. Programming Principles

## 4.1 SOLID

**SOLID** steht für fünf Prinzipien, die bei der Softwareentwicklung eingehalten werden sollten. Im Folgenden werden diese Prinzipien kurz erklärt und deren Einhaltung beziehungsweise Verwendung in diesem Projekt analysiert und begründet.

### 4.1.1 Single Responsibility Principle (SRP)

Das **Single Responsibility Principle** besagt, dass jede Klasse genau eine Aufgabe erfüllen sollte und es deshalb auch nur einen Grund geben sollte, eine Klasse zu ändern. Das Prinzip lässt sich ebenso auf andere Struktureinheiten des Codes, wie zum Beispiel Methoden oder auch Module übertragen. Dabei gilt auch, dass die *eine Aufgabe* auf Modulebene abstrakter ist als auf Klassenebene und sich aus Aufgaben mehrerer Klassen zusammensetzen kann.
Vorteile des Single Responsibility Principle sind unter anderem eine bessere Wiederverwendbarkeit, Wartbarkeit und Testbarkeit.

#### Analyse und Begründung für SRP

Bei der Betrachtung der Projektstruktur wird bereits deutlich, dass die Aufteilung des Codes in mehrere Packages bereits der Umsetzung des SRP dient. Das *analysis*-Package enthält dementsprechend nur Code, der für das Durchführen von Analysen geladener Daten verwendet wird. Die hier enthaltene Klasse *ContributionCountAnalyzer* hat die Funktion, Daten für eine Analyse zu laden und die eigentliche Analyse zu starten. Die eigentliche Analyse, die in diesem Fall ein Ranking der geladenen Daten nach verschiedenen Kriterien ist, ist in der Klasse *ContributionCountRanking* und ihren Ableitungen implementiert. Jede der Ableitungen ist für das Ranking nach bestimmten Kriterien verantwortlich. Prinzipiell wäre es auch möglich gewesen, auf die Ableitungen zu verzichten und stattdessen die verschiedenen Ranking-Implementierungen in der Basisklasse umzusetzen. Um das SRP nicht zu verletzen und eine spätere Erweiterung zu erleichtern (siehe

auch 4.1.2), wurde allerdings die Lösung gewählt, mehrere Ableitungen für jeweils eine Ranking-Implementierung zu erstellen.
Zum Beispiel enthält das *auth*-Package ausschließlich Code, der für die Authentifizierung (kurz: Auth) verantwortlich ist. Hier finden sich Interfaces und dazugehörige Implementierungen, die nur für die Authentifizierung eines Nutzers verwendet werden. Betrachtet man innerhalb dieses Packages die Klasse *GithubOAuthHandler* genauer, erkennt man, dass diese Klasse die Hauptaufgabe hat, einen Nutzer über GitHub zu authentifizieren. Allerdings bietet diese Klasse auch Funktionalität, eine erfolgreiche Authentifizierung für weitere Aktionen zu nutzen und eine Authentifizierung zu prüfen, was das SRP verletzt. Durch diese bewusste Verletzung des Prinzips wird in diesem Fall in Kauf genommen, da so eine Fragmentierung ähnlicher und fundamental zusammengehöriger Logik über mehrere sehr kleine Klassen vermieden werden kann.
Betrachtet man das *client*-Package erkennt man eine klare Zusammenstellung von Klassen mit dem Zweck, Daten über die GitHub-API abzurufen und Callback-Methoden mit diesen Daten aufzurufen. Zusammengefasst kümmert sich dieses Package um die Arbeit mit Daten aus der GitHub-API. Die Klasse *GithubOAuthClient* kann dabei als weiteres Beispiel für eine bewusste Verletzung des SRP angesehen werden, da hier die Funktionalität für das abrufen verschiedener Daten gesammelt ist um auch hier eine Fragmentierung über mehrere Klassen zu vermeiden. So kann diese Klasse verwendet werden um Daten zu Repositories, aber auch Daten zu Contributions eines Nutzers abzurufen. Aktuell fällt diese Verletzung nicht weiter negativ auf. Das Potential für negative Effekte, zum Beispiel durch Erweiterung um weitere Funktionalität, ist allerdings gegeben, weshalb in einem zukünftigen Refactoring diese Klasse aufgeteilt werden könnte.
Das *data*-Package ist vergleichsweise trivial, da es nur Klassen sammelt, die als Nutzdaten innerhalb der Anwendung verarbeitet werden. Alle Klassen in diesem Package haben als einzige Aufgabe die Repräsentation von Daten, die von der GitHub-API geladen wurden.
Im *repository*-Package ist Funktionalität zur Verwaltung von Daten realisiert. Dazu gehört auch das Sub-Package *cache*, dessen einziger Zweck das Caching von Daten ist. Diese Funktionalität ist hauptsächlich in der Klasse *ResponseCache* realisiert, die auch sonst keine weiteren Aufgaben erfüllt. Die abstrakte *Repository* Klasse und ihre Ableitungen stellen den Rest des *repository*-Packages dar und haben die einzige Aufgabe, das Caching der Daten für Verwender transparent zu gestalten, sodass ein Verwender Daten aus einem Cache beziehen kann anstatt sie jedes mal über die GitHub-API abzurufen. Weitere Verantwortlichkeiten lassen sich weder innerhalb der einzelnen Klassen, noch insgesamt auf das Package bezogen, erkennen.
Als Letztes Package kann das *ui*-Package genauer betrachret werden. Hier sind alle Klassen gesammelt, die für die Implementierung des User Interfaces benötigt werden. Damit

ist die Aufgabe dieses Packages, UI-Logik zu realisieren. Da das User-Interface in mehrere Teile mit spezifischeren Funktionen aufgeteilt ist, ist auch das gesamte Package in mehrere Sub-Packages zerlegt. Aber auch diese Sub-Packages erfüllen jeweils einen klar abgegrenzten Zweck. Als Beispiel hat das Sub-Packate *dashboard* die einzige Aufgabe, die Dashboard-Unterseite des User-Interfaces darzustellen.

### 4.1.2 Open / Closed Principle (OCP)

Nach dem **Open / Closed Principle** sollten Module immer offen für Erweiterungen und geschlossen für Änderungen sein. Dadurch sollen Erweiterungen erleichtert werden, ohne dass für eine Erweiterung bestehender Code angepasst werden muss. So wird der Code unter anderem auch Modularer.

#### Analyse und Begründung für OCP

Insgesamt wurde beim Design der Klassen immer darauf geachtet, Erweiterungen mit möglichst wenig Aufwand zu ermöglichen. Im Laufe der Entwicklung konnten die Vorteile davon bereits mehrfach ausgenutzt werden um Erweiterungen der bestehenden Funktionalität zu realisieren. Da hier nicht für alle enthaltenen Klassen eine ausführliche Betrachtung erfolgen kann werden stattdessen einige Beispiele angeführt. Als Beispiel für die Umsetzung des OCP, sollen zunächst die Klasse *ContributionCountRanking* und ihre Ableitungen betrachtet werden. Diese Klassen implementieren ein Ranking gegebener Listen nach unterschiedlichen Kriterien. Eine Erweiterung des Systems um weitere Ranking-Kriterien stellt hier kein Problem dar, da eine solche Erweiterung als weitere Ableitung von *ContributionCountRanking* erstellt werden kann. So muss kein bestehender Code geändert werden, eine Erweiterung um neue Funktionalität ist aber trotzdem möglich, was das OCP vollkommen erfüllt.
Ein gutes Beispiel für die Umsetzung des OCP stellt die Klasse *ResponseCache* dar. Diese Klasse wurde dafür konzipiert die Verwendung mit neuen Datentypen möglichst einfach zu gestalten. Anstatt eine Cache-Klasse für einen festen Datentyp zu erstellen wurde diese Klasse durch Verwendung von Generics so ausgelegt, dass alle Datentypen, die ein bestimmtes Interface implementieren, im Cache gespeichert werden können. Damit erfordert eine Erweiterung um Caching anderer Datentypen keinen weiteren Aufwand als die Implementierung der benötigten Interfaces.

### 4.1.3 Liskov Substitution Principle (LSP)

Im **Liskov Substitution Principle** ist beschrieben, dass eine Instanz einer abgeleiteten Klasse jederzeit so verwendbar sein soll wie eine Instanz der Basisklasse. Dabei wird nicht nur gefordert, dass die Verwendung technisch möglich ist, sondern dass diese Art der Verwendung auch logisch sinnvoll und frei von unerwünschten Nebeneffekten ist. Das führt schließlich dazu, dass jede Vererbung genauer betrachtet werden muss um zu prüfen, ob abgeleitete Klassen sinnvoll wie die Basisklasse eingesetzt werden können. Dadurch wird eine bessere, verständlichere Abstraktion gefördert und auch Fehler durch falsche Annahmen bezüglich Polymorphie werden verringert.

#### Analyse und Begründung für LSP

Bei Betrachtung des Codes fällt auf, dass nur an vergleichsweise wenigen Stellen Vererbung eingesetzt wurde, was die Betrachtung in Bezug auf das LSP erleichtert. Das erste Beispiel für die Umsetzung des LSP ist die Vererbungshierarchie ausgehend von der Klasse *ContributionCountRanking*. Hier lässt sich jede Ableitung der Klasse (z.B. *ContributionCountCommitRanking* und *ContributionCountTotalContributionsRanking*) genau wie die Basisklasse verwenden, da jede der Ableitungen nur eine unterschiedliche Implementierung der gleichen Funktionaliät ist. Alle diese Klassen haben die Aufgabe, eine gegebene Liste zu sortieren, nach welchen Kriterien die Liste sortiert wird hängt aber von der jeweiligen Implementierung ab. Ein Verwender dieser Klassen kann einfach eine Implementierung durch eine andere ersetzen ohne dass Seiteneffekte auftreten.
Ein weiteres Beispiel für die Verwendung des LSP findet sich im *ui*-Package mit der Klasse *DataAccessFragment* und ihren Ableitungen. Alle von dieser Klasse abgeleiteten Klassen können auf die gleich Art verwendet werden um Daten mithilfe eines *DataAccessViewModel*s zu laden. Dabei sind die Ableitungen dafür verantwortlich, ein passendes ViewModel zu laden. Ansonsten kann auf jeder Instanz einer dieser Ableitungen die *loadData*-Methode aufgerufen werden um verschiedene Daten zu laden und im User-Interface anzuzeigen.

### 4.1.4 Interface Segregation Principle (ISP)

Das **Interface Segregation Principle** sagt aus, dass mehrere spezifische Interfaces besser sind als ein einzelnes unspezifisches Interface. Dadurch soll der Code modularer, wartbarer und besser strukturiert werden. Allerdings muss dabei darauf geachtet werden, dass nicht zu viele Interfaces erstellt werden.

#### Analyse und Begründung für ISP

Es wird schnell deutlich, dass innerhalb der Anwendung viele Interfaces eingesetzt werden. Das kann als Zeichen dafür gesehen werden, dass das ISP konsequent umgesetzt wurde und große generische Interfaces weitgehend vermieden wurden. Allerdings kann es auch als Indiz dafür betrachtet werden, dass die Interfaces zu klein gewählt wurden. Da aber nur wenige Klassen mehr als ein Interface implementieren, kann davon ausgegangen werden, dass die Aufteilung nicht zu kleinteilig erfolgt ist. Ein Beispiel für eine Klasse, die mehrere Interfaces implementiert ist die Klasse *GithubOAuthClient*, die die Interfaces *RepositoryDataClient* und *UserContributionsClient* implementiert. Wie bereits in 4.1.1 beschrieben, realisiert diese Klasse die Funktionalität für das Abrufen von verschiedenen Arten von Daten. Ebenfalls bereits beschrieben wurde, dass diese Klasse zu groß ausfällt und eventuell weiter zerlegt werden muss. Deshalb lässt sich diese Klasse auch nicht als echtes Negativbeispiel für eine zu feingranulare Interface-Definition anführen.
Zusammenfassend kann man davon ausgehen, dass das ISP in der gesamten Anwendung umgesetzt wurde.

### 4.1.5 Dependency Inversion Principle (DIP)

Das **Dependency Inversion Principle** besagt, dass High-Level Module nicht von Low-Level Modulen direkt abhängen sollten, sondern beides möglichst von Abstraktionen abhängen sollte. Weiterhin sollen Abstraktionen nicht von Details, sondern Details von Abstraktionen abhängen. Dadurch soll der Code insgesamt wartbarer, modularer und besser wiederverwendbar gestaltet werden.

#### Analyse und Begründung für DIP

Bei der Entwicklung wurde versucht, das DIP wo möglich umzusetzen. Auch hier lässt sich die Klasse *ResponseCache* als Beispiel für die Umsetzung verwenden. Wenn man diese Klasse als Modul höherer Ebene ansieht und die Klassen, deren Objekte im Cache gespeichert werden können, einer niedrigeren Ebene zuordnet, dann erfüllt diese Klasse das DIP. Anstatt von konkreten Klassen abhängig zu sein, bestehen im ResponseCache nur Abhängigkeiten zu Abstraktionen, wie zum Beispiel der abstrakten Klasse *ResponseData*. So wird die Wiederverwendbarkeit dieser Klasse enorm erhöht.
Ein weiteres Beispiel findet sich in der Klasse *GithubOAuthClient*, die anstatt direkt von der einzigen Implementierung des Interfaces *AuthenticationHandler* abhängt, sondern nur von diesem Interface. In diesem Fall wird so hauptsächlich die Testbarkeit der Klasse erhöht, da eine spezifische Test-Implementierung des Interfaces für den Test verwendet werden kann.
Mit den Klassen *UserContributionsRepository* und *RepositoryDataRepository* gibt es noch zwei weitere Beispiele für eine gute Umsetzung dieses Prinzips. Anstatt direkt von der Klasse *GithubOAuthClient* abzuhängen, gibt es nur Abhängigkeiten zu den Interfaces *UserContributionsClient* beziehungsweise *RepositoryDataClient*. Die Erweiterbarkeit wird so deutlich verbessert, da jede beliebige Implementierung des jeweiligen Interfaces verwendet werden kann anstatt ein GithubOAuthClient-Objekt zu verwenden. Dadurch wird auch die bereits angesprochene mögliche Aufteilung der Klasse *GithubOAuthClient* deutlich einfacher möglich.

## 4.2 GRASP

**GRASP** (General Responsibility Assignment Software Pattersn) umfassen mehrere Muster und Prinzipien, die helfen können, zu bestimmen, welche Klasse für eine bestimmte Aufgabe zuständig sein sollte. Im Folgenden soll eine kurze Begründung für die Umsetzung dieser Muster erfolgen, insbesondere die beiden GRASP-Muster *Low Coupling* und *High Cohesion* sollen genauer betrachtet werden.

Das Prinzipt *Information Expert* sagt aus, dass neue Funktionalität in der Klasse implementiert werden soll, in der bereits die meisten benötigten Daten vorhanden sind. Diese Vorgehensweise entspricht im Grunde dem normalen Vorgehen bei einer Objektorientierten Entwicklung. Da dieses Prinzip so grundlegend ist, wird auf die Erläuterung von Beispielen verzichtet.

Das *Creator* Prinzip legt fest, welche Klasse für die Erzeugung einer Instanz dieser Klasse zuständig sein sollte. Demnach gibt es nur vier Fälle, in denen eine Klasse (A) eine Instanz einer anderen Klasse (B) erzeugen darf. Diese Fälle sind:

- A ist Aggregation von B / enthält Objekte von B
- A verarbeitet Objekte von B
- A ist stark von B abhängig
- A ist Information Expert für die Erzeugung von B

Jede beliebige Instanziierung einer Klasse innerhalb des Projekts, kann mit einem der hier angegebenen Gründe begründet werden. Deshalb wird hier ebenfalls darauf verzichtet konkrete Beispiel anzuführen.

Das *Controller* Muster beschreibt einen Controller als Einheit, der bestimmt, welche andere Einheit für bestimmte Ereignisse zuständig ist. Als erste Schnittstelle nach der Benutzeroberfläche delegiert der Controller Aufgaben an weitere Module. Die *-ViewModel* Klassen im *ui* Package können als Beispiel dafür herangezogen werden. Als Schnittstelle zwischen Benutzeroberfläche und eigentlicher Logik delegieren sie unter anderem Ereignisse die in der Benutzeroberfläche auftreten an passende Module.

### 4.2.1 Low Coupling

Low Coupling beschreibt, dass die Kopplung zwischen Modulen oder Elementen möglichst gering sein sollte. Dadurch wird die Testbarkeit, Wiederverwendbarkeit und auch Er-

weiterbarkeit verbessert. Die Reduzierung von Abhängigkeiten und damit der Kopplung zwischen Elementen ist ein grundlegendes Prinzip. In 1 wurden bereits mehrerer Beispiele angeführt, wie Abhängigkeiten aufgelöst wurden um die Kopplung zwischen Elementen zu verringern. Generell wurde aber darauf geachtet, dass Klassen möglichst lose gekoppelt sind, vor allem auch um eine spätere Erweiterung mit geringem Aufwand zu ermöglichen. Auch Prinzipien, wie zum Beispiel das Dependency Inversion Principle, helfen die Abhängigkeiten und damit die Kopplung zwischen Elementen zu verringern. Als Beispiel dafür können hier die bereits in 4.1.5 erklärten Beispiele dienen. Dadurch, dass Klassen so weit wie möglich nicht von konkreten Klassen abhängen, sondern von Abstraktionen, sind diese Klassen untereinander nur lose gekoppelt. Auch das Interface Segregation Principle kann durch bessere Abstraktion dazu führen, dass Klassen untereinander entkoppelt werden.

### 4.2.2 High Cohesion

Die *Cohesion* oder auch *Kohäsion* beschreibt, wie stark Bestandteile einer Klasse zusammenarbeiten. *High Cohesion* bedeutet also, dass die Elemente einer Klasse stark zusammenhängen und es zum Beispiel keine Methoden gibt, die nicht auf andere Bestandteile der Klasse angewiesen sind. Dadurch sollen unter anderem Low-Coupling und auch die Struktur des Codes verbessert werden. Dieses Prinzip weist eine gewisse Ähnlichkeit zum Single Responsibility Principle auf. *High Cohesion* sagt zum Beispiel aus, dass eine Klasse keine Methoden enthalten soll, die nichts mit den anderen enthaltenen Methoden zu tun haben oder keine Instanzvariablen der Klasse verwenden. Da solche Methoden mit hoher Wahrscheinlichkeit auch eine andere Verantwortlichkeit abbilden, sind diese auch aufgrund des Single Responsibility Principles nicht erwünscht. In anderen Punkten unterscheiden sich die Prinzipien jedoch, sodass eine Methode, die zwar Methoden oder Instanzvariablen einer Klasse benötigt, aber einem komplett anderen Zweck dient als der Rest der Klasse, trotzdem einer hohen Kohäsion nicht widerspricht.
Trotzdem kann aufgrund der beschriebenen Ähnlichkeit davon ausgegangen werden, dass eine Hohe Kohäsion vorliegt, wenn das Single Responsibility Principle konsequent umgesetzt wurde. Wie bereits in 4.1.1 beschrieben, entsprechen die meisten Klassen dem SRP, weshalb an dieser Stelle davon ausgegangen wird, dass diese Klassen auch eine angemessen hohe Kohäsion aufweisen.

Das nächste Prinzip, das in **GRASP** enthalten ist, ist *Polymorphism*. Damit soll erreicht werden, dass unterschiedliches Verhalten eines Typs durch Ausnutzung der Polymorphie realisiert werden soll anstatt andere Lösungen, wie zum Beispiel switch-statements zu verwenden. Als eine der Säulen Objektorientierter Softwareentwicklung ist es generell

sinnvoll Polymorphie einzusetzen, wo dies möglich ist. Auch dieses Prinzip hat einige Überschneidungen mit dem Liskov Substitution Principle, für das bereits Beispiele beschrieben wurden, die auch für diesen Abschnitt geeignet sind.

Eine *Pure Fabrication*, ist eine Klasse, die nicht in der Problemdomäne vorkommt und eine Art Hilfsklasse darstellt. So werden unter anderem technische Implementierungsdetails von der Problemdomäne getrennt. Ein sehr gutes Beispiel für eine *Pure Fabrication* ist die Klasse *ResponseCache*. Ein Cache wie dieser ist in den allermeisten Fällen eine reine Hilfsklasse und nicht in der Domäne enthalten.

*Indirection / Delegation* beschreibt, dass zwei Einheiten nicht direkt Kommunizieren, sondern über eine dritte Einheit, die Aufrufe delegiert oder weiterleitet. Damit wird direkt die Kopplung zwischen Klassen verringert, weshalb dieses Prinzip an vielen Stellen angewendet wird. Ein Beispiel dafür sind alle Aufrufe der eigentlichen Logik aus der Benutzerschnittstelle. Die Benutzerschnittstelle ruft nie direkt die eigentliche Logik auf. Stattdessen delegieren Event-Handler die Aufgaben an weitere Module der Anwendung. So wird sichergestellt, dass die Benutzeroberfläche vom Rest des Systems möglichst entkoppelt ist.

*Protected Variations* dienen dazu, das System vor den Auswirkungen eines Wechsels einer konkreten Implementierung zu schützen. Dies wird sichergestellt, indem Interfaces eingesetzt werden, um das System von konkreten Implementierungen unabhängig zu gestalten. Die Verwendung von Interfaces zur Entkopplung der Komponenten wurde bereits an mehreren Stellen beschrieben (siehe 4.1.5, 4.2.1, ...), weshalb hier nicht weiter darauf eingegangen werden soll.

## 4.3 DRY

*DRY* (Don't repeat yourself) sagt aus, dass man Redundanz im Code, zum Beispiel durch mehrfache Implementierung der gleichen Funktionalität, möglichst vermeiden sollte. Stattdessen soll der Code so gestaltet werden, dass einzelne Elemente wiederverwendbar sind, sodass es gar keinen Grund gibt, eine Funktion mehrfach zu implementieren. Im Abschnitt 3.1.1 wurde bereits der gesamte Code durchsucht und dabei festgestellt, dass es nur sehr wenige Wiederholungen im Programmcode gibt. Deshalb soll hier das gleiche Ergebnis nicht noch weiter ausgeführt werden.

# 5. Unit Testing

## 5.1 Analyse und Begründung für Umfang der Tests

Um die Funktionalität der einzelnen Komponenten gewährleisten zu können werden Unit-Tests eingesetzt. Dabei werden für die einzelnen Tests nur die, für diesen Test, relevanten Teile des Systems verwendet. Da Abhängigkeiten zu anderen Komponenten die Tests nicht beeinflussen sollen werden alle anderen Komponenten durch Fake-/Mock-Objekte ersetzt. Das Zusammenspiel mit den anderen Komponenten kann in Integrationstests getestet werden. Außerdem tragen Unit-Tests auch zur Dokumentation bei, indem das gewünschte Verhalten der Komponente für Regel- und Ausnahmefälle in den Testfällen dokumentiert ist.
Für dieses Projekt wird *JUnit* als Framework für die Erstellung und Ausführung von Java-Unit-Tests verwendet. Bei der Implementierung der Tests wurde darauf geachtet die ATRIP-Regeln (*Automatic, Thorough, Repeatable, Independent, Professional*) möglichst zu beachten.
Teile des Codes in diesem Projekt werden für Android-Spezifische UI-Aufgaben benötigt und können deshalb nur schlecht getestet werden. Das führt auch dazu, dass die Code-Coverage über das gesamte Projekt vergleichsweise klein sein kann und in diesem Fall keine sinnvolle Aussage über die Genauigkeit der Tests ermöglicht. Stattdessen sollte hier zur Beurteilung der Testabdeckung nur die Code-Coverage der nicht-UI-Klassen betrachtet werden.
Allgemein wird darauf verzichtet triviale Funktionen, wie zum Beispiel Getter zu testen. Tests dieser Funktionen würden bei großem Aufwand nur einen minimalen Mehrwert bringen, da keine ëchte"Funktionalität getestet wird. Stattdessen sollen sich die Tests auf relevante Funktionalität des Systems fokussieren. Das bedeutet, dass vor allem die Klassen getestet werden sollen, die häufig verwendet werden und auch die Klassen, die für die Funktion der Anwendung unerlässlich sind. UI-Klassen, die aufgrund starker Abhängigkeiten zu Android-Klassen schwer testbar sind sollen weniger ausführlich getestet werden.
Für die Tests innerhalb dieses Systems wurden drei Kernfunktionalitäten identifiziert, die von einer hohen Testabdeckung am stärksten profitieren. Diese sind das Abrufen von Da-

ten von der API, die Konvertierung von API-Datentypen zu Datentypen der Anwendung und in diesem Zusammenhang auch das Caching von abgerufenen Daten um die Netzwerklast zu reduzieren.
Diese Funktionalitäten verteilen sich auf die Packages *analysis*, *auth*, *client*, *data* und *repository*. Abbildung igt einen Code-Coverage-Report dieser Packages (Stand: Commit 28770de). Sten Pittet beschreibt in [3], dass 80% Code-Coverage als gutes Ziel anerkannt sind, weshalb auch hier das Ziel gesetzt wird, mindestens 80% der Methoden zu testen. Der Code-Coverage-Report, zeigt, dass alle relevanten Klassen getestet wurden. Allerdings

| | | | |
|---|---|---|---|
| de.lukaspanni.opensourcestats.analysis | 100% (5/5) | 86% (13/15) | 86% (64/74) |
| de.lukaspanni.opensourcestats.auth | 33% (1/3) | 28% (6/21) | 23% (21/89) |
| de.lukaspanni.opensourcestats.client | 100% (5/5) | 64% (11/17) | 45% (28/61) |
| de.lukaspanni.opensourcestats.data | 100% (12/12) | 82% (61/74) | 89% (165/185) |
| de.lukaspanni.opensourcestats.repository | 100% (3/3) | 37% (6/16) | 57% (27/47) |
| de.lukaspanni.opensourcestats.repository.cache | 100% (2/2) | 100% (12/12) | 100% (32/32) |
| de.lukaspanni.opensourcestats.type | 100% (3/3) | 50% (4/8) | 55% (5/9) |
| de.lukaspanni.opensourcestats.ui | 0% (0/1) | 0% (0/1) | 0% (0/2) |

Abbildung 5.1: Code Coverage Report

fällt dabei zunächst auf, dass die Coverage des *ui*-Packages sehr schlecht ist, da der Fokus auf Tests der Funktionalität und nicht der Darstellung gelegt wurde. Außerdem schwankt die Methoden-Coverage stark, was darauf zurückzuführen ist, dass versucht wurde, triviale Funktionalität nicht ausführlich zu testen. Im Folgenden werden Funktionalität und Tests der einzelnen Packages noch weiter erläutert.

Das analysis-Package enthält alle Klassen, die für die Erstellung von Analysen verwendet werden. Da diese Funktionalität, vermutlich noch am stärksten weiterentwickelt wird, wird hier eine gute Code-Coverage als sehr wichtig betrachtet. Sowohl auf Methoden als auch auf Zeilen bezogen wird hier eine gute Code-Coverage von 86% erreicht.

Im auth-Package, das alle Klassen die zur Authentifizierung benötigt werden enthält, wurde nur eine vergleichsweise kleine Code-Coverage von gerade mal 28% aller Methoden erreicht, was als schlecht zu bewerten ist. Da für die Authentifizierung eine Third-Party-Library verwendet wird, ist hier allerdings nicht mit angemessenem Aufwand eine höhere Coverage erreichbar. Deshalb wurde die Entscheidung getroffen, die schlechte Coverage zu akzeptieren.

Das data-Package enthält alle Klassen, die als Nutzdaten von der Anwendung verarbeitet werden. Antworten, die von der API empfangen werden, werden in einem ersten Schritt in

einen solchen Datentyp konvertiert. Diese Konvertierungen werden in den Konstruktoren der Klassen durchgeführt. Ansonsten werden diese Klassen hauptsächlich zur Speicherung von Anwendungsdaten verwendet und enthalten nur vergleichsweise wenig weitere Funktionalität. Da die Konvertierung von API-Datentyp in Anwendungsdatentyp sehr häufig verwendet wird und die Anwendungsdatentypen generell an vielen verschiedenen verwendet werden, sind hier ausführliche Tests notwendig. Der Fokus der Tests liegt dabei auf den Konstruktoren, die wenigen weiteren Funktionalitäten der Klassen sind weniger wichtig und meist trivial. Die Code-Coverage von 82% aller Methoden wird deshalb als gut angesehen, vor allem wenn die Line-Coverage von 89% in die Betrachtung mit einbezogen wird.

Im repository-Package, sind einerseits Repository-Klassen enthalten, die den Zugriff auf Anwendungsdaten abstrahieren und dabei das Caching der Daten transparent machen. Andererseits finden sich hier auch die Klassen, die für die Umsetzung des Cachings verwendet werden. Die Repository-Klassen übernehmen die Funktion, die Daten aus dem Cache abzurufen, wenn diese dort vorhanden sind und ansonsten die Daten über Funktionalität, die im client-Package implementiert ist von der API-abzurufen. Als zentrale Komponente, um auf Daten zuzugreifen sind die Repository-Klassen deshalb entscheidend für die Funktionalität der gesamten Anwendung. Aus diesem Grund ist es wichtig hier eine möglichst hohe Testabdeckung zu erreichen. Die niedrige Method-Coverage von nur 37% ist hier trotzdem angemessen, da sie auf nicht getestete triviale Funktionen zurückzuführen ist. Diese trivialen Funktionen sind verschieden Parametrisierte Konstruktoren und Hilfsfunktionen, die selbst wiederum nur Standard-Parameter für getestete Funktionen bereitstellen (siehe: UserCotributionsRepository.load...)
Die Komponente zum Caching ist ebenfalls wichtig, da ohne funktionierendes Caching sehr viele Anfragen über das Netzwerk gesendet werden müssen, was nicht nur die Menge der übertragenen Daten unnötig erhöht, sondern auch die Reaktionszeit der Anwendung verschlechtert. Deshalb ist auch im cache-Package eine hohe Testabdeckung wünschenswert. In diesem Package wird sowohl für Method- als auch Line-Coverage ein Wert von 100% erreicht, was als sehr gut zu bewerten ist. Durch diese hohe Coverage soll sichergestellt werden, dass möglichst wenige Fehler beim Caching auftreten können und alle anderen Komponenten den Cache mit hoher Sicherheit verwenden können.

Das client-Package enthält alle Klassen, die benötigt werden, um Daten von der API abrufen zu können und steht damit, wie auch das cache-Package, im Zusammenhang mit dem Datenzugriff über die Repository-Klassen. Es ist offensichtlich, dass auch diese Funktionalität von großer Bedeutung für die Funktionalität der Anwendung ist, und deshalb auch in diesem Package eine möglichst hohe Testabdeckung angestrebt wird. Method- und

Line-Coverage sind hier mit 64% beziehungsweise 45% nicht als gut zu bewerten. Hier sollte die Coverage durch weitere Testes erhöht werden, was sich allerdings als schwierig herausstellte, da eine starke Kopplung an eine Thirt-Party-Library besteht, die nur schwer aufgelöst werden kann. Deshalb wird die Code-Coverage aktuelle akzeptiert, sollte in Zukunft aber noch deutlich ausgebaut werden.

Im Screenshot ist zusätzlich das Package *type* aufgeführt. Dieses enthält ausschließlich automatisch generierten Java-Code, der für die Verwendung bestimmter Typen in Verbindung mit der GitHub-GraphQL-API benötigt wird. Die Code-Coverage dieses Packages ist deshalb nicht relevant.

Im Gegensatz zu den genannten Bereichen ist eine hohe Testabdeckung im *ui*-Package weniger wichtig und bringt nur wenig Nutzen. Zusätzlich ist hier aufgrund der genannten Einschränkungen eine hohe Testabdeckung nur mit sehr hohem Aufwand möglich. Im Anbetracht des geringen Nutzens ist der Aufwand hier eine hohe Testabdeckung zu erreichen nicht gerechtfertigt, weshalb hier weitestgehend auf Tests verzichtet wird.

## 5.2 Analyse und Begründung für Einsatz von Fake-/Mock-Objekten

Fake- und Mock-Objekte werden benötigt, um Abhängigkeiten einer Komponente zu anderen Komponenten in Unit-Tests zu reduzieren. Sie implementieren dafür zum Beispiel das benötigte Interface, aber davon nur die aktuell benötigte Funktionalität. Fake- und auch Mock-Objekte wurden eingesetzt, um Abhängigkeiten zu Third-Party-Komponenten, z.B. Android-Spezifische Klassen, und zu eigenen Klassen zu ersetzen. Alle Fake- und Mock-Objekte sind im Package *mock* gesammelt.
Im Folgenden sollen beispielhaft einige der Fake-/Mock-Klassen genauer erläutert werden.

### FakeUserContributionsClient

Diese Klasse ist ein Fake-Objekt, das das Interface *UserContributionsClient* implementiert. Diese Interface wir außerhalb der Tests für Clients verwendet, die bestimmte Daten (spezifisch: Objekte der Klasse *UserContributionsResponse*) von einer API abrufen.
Offensichtlich sollen Tests von Klassen, die das Interface verwenden, auch ohne echte API-Anfragen möglich sein, da so auch weitere unerwünschte Abhängigkeiten im Test reduziert werden können. Die Klasse *FakeUserContributionsClient* implementiert dazu die Funktionen des Interfaces und zeichnet die Parameter der Methodenaufrufe auf, sodass Auswertungen im Test möglich sind.
Listing 5.1 zeigt beispielhaft, wie die Klasse *FakeUserContributionsClient* im Rahmen eines Tests eingesetzt werden kann. Das Code-Beispiel stammt aus UserContributionsRepositoryUnitTest.

```
@Test
public void
   test_repository_summary_force_reload_client_not_cache() {
    TimeSpan timeSpan = TimeSpanFactory.getCurrentWeek();
    FakeUserContributionsClient client = new
       FakeUserContributionsClient();
    ResponseCache<TimeSpan, UserContributionsResponse> cache =
       new ResponseCache<>();
    UserContributionsRepository testObject = new
       UserContributionsRepository(cache, client);

    testObject.loadUserContributionsInTimeSpan(timeSpan,
       response -> {
    }, true);

    assertThat(client.isCalled(), is(true));
    //if hits or misses > 0 repository has tried to retrieve
       from cache
    assertThat(cache.getMisses(), is(equalTo(0)));
    assertThat(cache.getHits(), is(equalTo(0)));
}
```

Listing 5.1: Beispielhafte Verwendung von FakeUserContributionsClient

In diesem Beispiel soll getestet werden, dass eine Instanz der Klasse *UserContributionsRepository* bei setzen des Parameters forceReload die Daten nicht aus dem Cache

bezieht sondern einen UserContributionsClient verwendet. Um zu testen, ob eine Methode des UserContributionClient aufgerufen wird, wird in diesem Fall ein Objekt der Klasse *FakeUserContributionsClient* als Implementierung von UserContributionClient an die Repository-Instanz übergeben. Nach dem Aufruf der Methode *loadUserContributionsInTimeSpan* der Repository-Klasse wird geprüft, ob das Fake-Objekt einen Aufruf der Methode aufgezeichnet hat und ob Methoden des Caches aufgerufen wurden. Wenn das Fake-Objekt aufgerufen wurde und der Cache nicht, dann ist das Verhalten in diesem Fall korrekt.

### FakeRespnseData

Die Klasse *FakeResponseData* ist abgeleitet von der abstrakten Klasse *ResponseData*, die die Basisklasse für alle Daten darstellt, die von der API als Antwort empfangen werden. Eingebettet in ein Objekt der Klasse *CacheEntry* werden *ResponseData*-Objekte im internen Cache (Objekte der Klasse *ResponseCache*) gespeichert.
Für Tests der Klasse *ResponseCache* werden also *CacheEntry*-Objekte benötigt, die selbst wiederum ein Objekt der Klasse *ResponseData* benötigen. Um diese Objekte für Tests erstellen zu können muss eine konkrete Klasse von *ResponseData* abgeleitet werden. *FakeResponseData* übernimmt diese Funktion und implementiert die Speicherung eines einfachen Integer-Werts als Test-Datum und die *equals*-Methode um die Gleichheit zu anderen Objekten feststellen zu können. Diese Funktionalität ist für Tests ausreichend.
Listing 5.2 zeigt, wie Objekte der Klasse *FakeResponseData* in *ResponseCache*-Tests verwendet werden. Das Codebeispiel stammt aus ResponseCacheUnitTest.

```
@Test
public void cache_get_returnsCorrectEntry() {
    ResponseCache<TimeSpan, FakeResponseData> testCache = new
       ResponseCache<>();
    FakeResponseData testResponseDataCorrect = new
       FakeResponseData(42);
    FakeResponseData testResponseDataWrong = new
       FakeResponseData(51);
    TimeSpan correctTimeSpan = new TimeSpan(new Date(2020, 11,
       9), new Date(2020, 11, 10));
    TimeSpan wrongTimeSpan = new TimeSpan(new Date(2020, 10,
       1), new Date(2020, 10, 2));
    testCache.put(correctTimeSpan, testResponseDataCorrect);
    testCache.put(wrongTimeSpan, testResponseDataWrong);

    ResponseData cachedData = testCache.get(correctTimeSpan);
    assertThat(cachedData, is(testResponseDataCorrect));
}
```

Listing 5.2: Beispielhafte Verwendung von FakeResponseData

Der hier gezeigte Test soll testen, ob der *ResponseCache* bei mehreren enthaltenen Elementen, die zu einem Key (in Form eines *TimeSpan*-Objekts) passenden Daten zurückgibt. Dazu werden gleich zwei *FakeResponseData*-Objekte erstellt und mit eigenen Keys im Cache gespeichert. Abschließend wird geprüft, ob ein aus dem Cache angefordertes *Fake-*

*ResponseData*-Objekt, korrekt ist.

# 6. Clean Architecture

# 7. Domain Driven Design

## 7.1 Analyse der Ubiquitous Language

Kern des *Domain Driven Design* ist das gemeinsame Verständnis der Problemdomäne. Dies ist notwendig, um ein präzises Modell der Domäne erstellen zu können. Um die Kommunikation zwischen Entwicklern und Domänenexperten zu ermöglichen wird die sogenannte **Ubiquitous Language** verwendet. Da weder Domänenexperten die Sprache der Entwickler, noch umgekehrt die Entwickler die Sprache der Domänenexperten verstehen ist die Ubiquitous Language als gemeinsame Sprache unerlässlich. Diese wird von Entwicklern und Domänenexperten gleichermaßen verwendet, um Begriffe, Prozesse und Konzepte eindeutig zu bezeichnen. Wenn die Ubiquitous Language korrekt eingesetzt wird werden in der Domäne und im Quellcode der Anwendung die gleichen Begriffe für die gleichen Konzepte verwendet. So können Missverständnisse und Mehrdeutigkeiten sicher vermieden werden, was gerade bei komplexen Domänen den Softwareerstellungsprozess erleichtert. Beim definieren der Ubiquitous Language sollte der Fokus auf die Begriffe und Konzepte der Kerndomäne gelegt werden. Andere Bereiche der Domäne können auch weniger genau modelliert werden, damit der Aufwand angemessen bleibt.

Die Kerndomäne dieses Projekts ist die Übersicht über Beteiligungen (Contributions) an Öffentlichen Repositories auf der Plattform *GitHub*. Die Ubiquitous Language ist dabei vergleichsweise einfach zu definieren, da sich die Anwendung an Entwickler richtet, die in diesem Fall gleichzeitig auch Domänenexperten sind. Trotzdem muss darauf geachtet werden, dass einheitliche Begriffe verwendet werden. Da die Plattform GitHub eindeutige Begriffe für die Bezeichnung von verschiedenen, wichtigen Konzepten verwendet und kein Einfluss auf diese Bezeichnungen besteht, wurde entschieden diese Bezeichnungen möglichst zu verwenden. Dabei werden diese Bezeichnungen nicht nur im Code sondern auch im User-Interface eingesetzt.

## Repository / Repositories

Der Begriff Repository beschreibt in diesem Kontext ein Verzeichnis, das zur Ablage von z.B. Programmcode verwendet werden kann. Dabei bietet ein Repository bei GitHub aber auch noch weitere Funktionen, die hier aufgrund der Fehlenden Relevanz für das System nicht genauer erläutert werden. Das Verständnis eines Repositoriy als Verzeichnis zur Ablage von Daten ist ausreichend. Der Begriff wird sowohl im Code als auch im User-Interface verwendet. Beispiele für die Verwendung des Begriffs im Code finden sich in der Klasse *RepositoryDataResponse*, die Daten zu einem Repository speichert, die von der API abgerufen wurden und in der Klasse *ContributionRepositories*, die beschreibt, in welchen Repositories Contributions durchgeführt wurden.
Beim Begriff Repository existiert allerdings eine Doppeldeutigkeit, da der Begriff Repository innerhalb des Systems auch als Bezeichnung für Klassen, die das *Repository-Pattern* implementieren verwendet wird. Das führt zum Beispiel zu der Mehrdeutigen Bezeichnung der Klasse RepositoryDataRepository. Diese Mehrdeutigkeit kann nur schwer beseitigt werden, da einerseits der Repository-Begriff aus der Domänensprache verwendet werden soll, andererseits aber auch Repository-Pattern-Klassen eindeutig gekennzeichnet werden sollen. Deshalb wird diese Mehrdeutigkeit akzeptiert und festgelegt, dass Klassen die das Repository-Pattern implementieren, den Begriff Repository am Ende des Klassennamens enthalten. Kommt der Begriff an anderen Stellen vor, kann mit hoher Wahrscheinlichkeit davon ausgegangen werden, dass ein Repository im Sinne der Domäne gemeint ist.

## Contribution / Contributions

Beteiligungen an Repositories werden auf GitHub als Contributions bezeichnet. Die API, die verwendet wird um Daten abzurufen, verwendet ebenfalls diese Bezeichnung. Folglich soll auch im Code diese Bezeichnung für das gleiche Konzept verwendet werden.
Der Begriff Contributions ist allerdings nur ein Sammelbegriff für vier weitere Konzepte, die von großer Bedeutung für die Ubiquitous Language der Problemdomäne sind. Diese Konzepte sind *Commits*, *Issues*, *Pull Requests* und *Pull Request Reviews*, und werden im Folgenden beschrieben. Die jeweilige Anzahl in einem bestimmten Zeitraum lässt sich über die GitHub-API abrufen und ist innerhalb dieser Anwendung von hoher Relevanz, da aus diesen Daten verschiedene Statistiken erstellt und dem Nutzer angezeigt werden.

### Commit / Commits

Ein Commit kann in diesem Kontext als Prozess vorgenommene Änderungen am Quellcode zu bestätigen verstanden werden. Die Anzahl der Commits in einem bestimmten

Zeitraum erfasst so, wie viele unterschiedliche Änderungen ein Nutzer in diesem Zeitraum auf GitHub öffentlich gemacht hat.

#### Issue / Issues

Issues werden bei GitHub verwendet um Ideen, neue Features, Aufgaben und Bugs zentral zu verwalten und stellen die zweite Art der Contribution dar. Die Anzahl der Issues beschreibt, wie viele neue Issues ein Nutzer in einem bestimmten Zeitraum erstellt hat. Diese Beschreibung reicht für das Verständnis im Rahmen dieses Projekts aus.

#### Pull Request / Pull Requests

GitHub verwendet sogenannte Pull Requests um Änderungen Code aus einem Branch in einen anderen, z.B. den main-Branch, zu übernehmen. Dabei bietet ein Pull Request die Möglichkeit weitere Tests oder auch ein Code-Review durchzuführen, bevor der Pull Request angenommen wird. Damit stellen Pull Requests ein wichtiges Konzept dar, das die Kollaboration erleichtert. Die Anzahl an Pull Requests beschreibt, wie viele Pull Requests eine Nutzer in einem bestimmen Zeitraum erstellt hat.

#### Pull Request Review / Pull Request Reviews

Ein Pull Request Review beschreibt bei GitHub ein Code-Review im Rahmen eines Pull Requests. Das Kommentieren von Änderungen eines anderen Nutzers, die dieser in Form eines Pull Requests einreicht, wird dabei als Pull Request Review gewertet. Die Anzahl von Pull Request Reviews beschreibt, wie viele Code-Reviews von Pull Requests anderer Nutzer durchgeführt wurden.

Die Klasse *ContributionCount* ist ein Beispiel dafür, wie alle Bezeichnungen, die im Zusammenhang mit Contributions stehen, auch im Code verwendet werden. Dies wird in Listing 7.1 nochmals verdeutlicht. Die Klasse ContributionCount hat je eine Instanzvariable für die Anzahl von Commits, Issues, Pull Requests und Pull Request Reviews und verwendet somit alle vorgestellten Bezeichnungen, die in der Domäne im Zusammenhang mit Contributions verwendet werden.

```
public final class ContributionCount {

    private final int commitCount;
    private final int issueCount;
    private final int pullRequestCount;
    private final int pullRequestReviewCount;
```

```
    public ContributionCount(int commits, int issues, int
       pullRequests, int pullRequestReviews) {
        this.commitCount = Math.max(commits, 0);
        this.issueCount = Math.max(issues, 0);
        this.pullRequestCount = Math.max(pullRequests, 0);
        this.pullRequestReviewCount =
           Math.max(pullRequestReviews, 0);
    }
[...]
}
```

Listing 7.1: Auszug aus der Klasse ContributionCount

Auch im User-Interface werden diese Bezeichnungen gemeinsam verwendet, wie zum Beispiel die UI-Klasse *OverviewCard* zeigt, die ebenfalls all diese Bezeichnungen verwendet. Generell sind diese Begriffe von großer Bedeutung für das System und werden deshalb noch an vielen weiteren Stellen verwendet. Weitere Beispiele sind unter anderem die Klasse *ContributionCountChange*, die eine Veränderung der Contribution-Anzahl im Vergleich zu einem anderen Zeitraum beschreibt und die zugehörige UI-Klasse *ProgressCard*.

## 7.2 Analyse und Begründung von Value Objects

**ValueObjects** sind vergleichsweise einfache Objekte, die nur durch ihre Werte beziehungsweise Eigenschaften beschreiben werden. Das hat unter anderem zur Folge, dass ValueObjects als gleich angesehen werden, wenn ihre Werte gleich sind. Allgemein beschreiben ValueObjects oft eine bestimmte Sache durch die Kombination mehrerer Attribute dieser Sache. Typische ValueObjects sind unter anderem Adressen und Geldbeträge. Dabei sind ValueObjects außerdem unveränderbar, so dass alle Werte bereits im Konstruktor gesetzt werden müssen. Diese scheinbare Einschränkung der Unveränderlichkeit kann vorteilhaft sein, da zum Beispiel die Gültigkeit von Werten nur an einer Stelle (im Konstruktor) geprüft werden muss und ein so erstelltes ValueObject zu jeder Zeit gültig ist. Damit sind ValueObjects auch leicht testbar, da nur die Gültigkeitsprüfungen im Konstruktor getestet werden müssen. Die Test-Klassen ContributionCountUnitTest und ContributionCountChangeUnitTest sind gute Beispiele für Tests von ValueObjects, die hier verwendet wurden.
Umgesetzt werden ValueObjects, indem jegliche Änderungen an Werten des Objekts eingeschränkt werden. Dazu bieten verschiedene Programmiersprachen verschiedene Sprachfeatures.
Konkret bedeutet das für die hier verwendete Sprache Java:

- Klasse muss als final markiert sein, sodass auch keine Veränderungen durch Vererbung möglich sind
- Alle Membervariablen müssen ebenfalls als final gekennzeichnet werden, sodass die Variablen spätestens im Konstruktor gesetzt werden und sonst nicht mehr geändert werden können
- Dabei ist darauf zu achten, dass über den Konstruktor auch nur gültige Objekte erstellt werden können
- Die *equals* und auch die *hashCode* Methoden müssen geeignet überschrieben werden, sodass die Gleichheit durch gleiche Werte der Membervariablen festgelegt wird

### 7.2.1 Verwendete ValueObjects

Im *data*-Package existieren mehrere Klassen, die **ValueObjects** darstellen.
Diese Klassen sind:

- **ContributionCount**
  Stellt die Anzahl von GitHub-Contributions dar, ausgedrückt als Sammlung der Anzahl von Commits, Issues, Pull-Requests und Pull-Request-Reviews.

- **ContributionCountChange**
  Repräsentiert die Differenz zweier ContributionCount-Objekte und bietet die Möglichkeit die Unterschiede zwischen aktueller und vorangegangener Periode in Bezug auf Commits, Issues Pull-Requests und Pull-Request-Reviews zu berechnen.

- **ContributionRepositories**
  Sammelt die Repositories, für die Commits, Issues, Pull-Requests und Pull-Request-Reviews durchgeführt wurden.

- **TimeSpan**
  Repräsentiert eine Zeitspanne, bestehend aus einem Start- und einem Enddatum.

- **RepositoryName**
  Repräsentiert den vollen Namen eines GitHub-Repositories, der sich aus Besitzer des Repositories und Bezeichnung des Repositories zusammensetzt.

Verwendet werden diese ValueObjects aufgrund ihrer Vorteile, wie zum Beispiel der leichten Testbarkeit. So kann leicht sichergestellt werden, dass das Verhalten der Objekte den Erwartungen entspricht.
Objekte der Klasse *TimeSpan* zum Beispiel werden für das Caching, für das Abrufen von Daten und auch im User-Interface für die Anzeige von Zeiträumen eingesetzt. Die gute Testbarkeit von ValueObjects ist dadurch an mehreren Stellen von Vorteil und erleichtert die Entwicklung, da davon ausgegangen werden kann, dass sich TimeSpan-Objekte korrekt verhalten. Für die anderen genannten ValueObjects sind die Vorteile ähnlich.

## 7.3 Analyse und Begründung Entities

**Entities** haben im Gegensatz zu ValueObjects innerhalb der Domäne eine eindeutige ID, über diese sie identifiziert werden können, und die die Gleichheit von Entities bestimmt. Zusätzlich können sich Entities auch während ihrer Lebenszeit verändern und sind nicht immutable. Auch bei Entities ist es wichtig, dass sie nicht mit unerlaubten Werten initialisiert werden können und darüber hinaus auch nicht durch eine Aktion ungültige Zustände erreichen dürfen.
Aktuell konnten keine Objekte innerhalb des Projekts identifiziert werden, die die Eigenschaften einer Entity erfüllen. Der Grund dafür ist vermutlich, die vergleichsweise geringe Komplexität des Gesamtsystems.

## 7.4 Analyse und Begründung von Aggregates

**Aggregates** sind ein Konzept zur Zusammenfassung von mehreren Entities und ValueObjects zu einer Einheit. Als Einheit werden Aggregates auch immer nur vollständig geladen und gespeichert, sodass ein Aggregate immer nur vollständig vorhanden sein kann. Eingesetzt werden Aggregates um die Beziehungen zwischen einzelnen Entities und ValueObjects zu vereinfachen, indem mehrere Entities und ValueObjects in einem Aggregate zusammengefasst werden und Beziehungen zu den enthaltenen Entities und ValueObjects nur über das Aggregate realisiert werden. Außerdem bildet auch ein einzelne Entity bereits ein Aggregate. Zur Verwaltung des Zustands eines Aggregates gibt es immer eine Aggregate Root Entity, über die alle Zugriffe auf das Aggregat durchgeführt werden. So können an einer Stelle alle Zugriffe auf Validität geprüft werden, was die Prüfung von Domänenregeln einfacher macht.
Aggregates sind vergleichsweise komplex und kommen deshalb auch häufig nur in komplexen Systemen eingesetzt. Es gibt im Projekt keine klassischen Aggregates, die mehrere Entities und ValueObjects zusammenfassen, da es bereits keine echten Entities gibt. Die Klasse *UserContributionsResponse* ist einem Aggregate ähnlich, da hier mehrere ValueObjects zusammengefasst werden. Da diese Klasse allerdings kein Verhalten auf den enthaltenen ValueObjects durchführt, sondern diese nur erstellt, speichert und bei Bedarf zurückgibt, kann man diese Klasse nicht als klassisches Aggregate ansehen.

## 7.5 Analyse und Begründung Repositories

**Repositories** stellen die Schnittstelle zwischen Domäne und Datenmodell dar und stellen demnach Methoden bereit um Daten zu lesen, speichern oder zu löschen. Dabei werden Repositories häufig eingesetzt, um einen einfachen Datenzugriff für die Domäne zu ermöglichen und die technischen Details, wie zum Beispiel Datenbankzugriffe, zu kapseln. Normalerweise operieren Repositories auf der Basis von Aggregates. Um die Domäne frei von technischen Details zu halten werden Interfaces für Repositories innerhalb der Domäne verwendet und die eigentliche Implementierung findet außerhalb der Domäne statt. Zusätzlich bieten Repositories die Möglichkeit nicht nur allgemein Daten abzurufen, sondern Daten anhand von bestimmten Kriterien zu finden, was den Zugriff auf Daten für die Domänenobjekte erleichtert.
Da, wie bereits beschrieben, keine Aggregates verwendet werden, finden sich auch keine typischen Repositories. Stattdessen gibt es im *repository*-Package mehrere Klassen, die die Bezeichnung Repository tragen. Diese erfüllen einen ähnlichen Zweck und dienen ebenfalls der Kapselung technischer Details des Datenzugriffs. Sie rufen Daten aus einem internen Cache ab oder verwenden die GitHub-API um die Daten zu laden und im Cache zu speichern. Allerdings bieten sie nicht die Möglichkeit Daten zurückzugeben. Stattdessen werden den Datenzugriffsfunktionen Callback-Funktionen übergeben, die mit den abgerufenen Daten aufgerufen werden. Dies ist eine bewusste Entwurfsentscheidung, die getroffen wurde, um nicht nur die Anbindung an die verwendete GitHub-API und die gleichzeitige Verwendung eines eigenen Caching-Mechanismus zu erleichtern. Zusätzlich wird so eine Asynchrone Datenverarbeitung gefördert, die gerade bei hohen Netzwerklatenzen bei Mobilgeräten, und vielen einzelnen API-Anfragen die Wartezeit für den Nutzer merklich verringern kann.

## 7.6 Fazit Domain Driven Design

Domain Driven Design umfasst verschiedene Konzepte, die genutzt werden können, um den Fokus auf die eigentliche Domäne zu steigern und die zusätzliche Komplexität zu verringern. Viele dieser Konzepte sind allerdings sehr aufwändig und der Nutzen wird häufig erst in größeren Anwendungen wirklich sichtbar. Im Rahmen dieser Arbeit wurden verschiedene Konzepte des Domain Driven Design genutzt. In *Analyse der Ubiquitous Language* wurde gezeigt, wie das Konzept einer einheitlichen Sprache sinnvoll genutzt wurde um Mehrdeutigkeiten möglichst zu vermeiden. Auch ValueObjects wurden genutzt, um bestimmte Dinge, wie zum Beispiel GitHub-Contributions, abbilden zu können und die Vorteile von unveränderlichen Objekten zu nutzen. Andere Konzepte, wie zum Bei-

spiel Entities, Aggregates und Repositories wurden bewusst nicht, beziehungsweise nicht im eigentlichen Sinne, verwendet, da der Aufwand für die Verwendung dieser Konzepte in Relation zum Nutzen für ein vergleichsweise kleines System nicht gerechtfertigt ist. Zusätzlich beruht diese Entscheidung auf auf der Tatsache, dass die Verwendung eines Konzepts nur des Konzepts wegen nicht sinnvoll ist.

# 8. Literaturverzeichnis

[1] M. Geirhos, *Entwurfsmuster: das umfassende Handbuch*. Rheinwerk-Verlag, 2015.

[2] M. Lippert and S. Roock, *Refactoring in large software projects: performing complex restructurings successfully*. John Wiley & Sons, 2006.

[3] S. Pittet, "Introduction to code coverage." `https://www.atlassian.com/continuous-delivery/software-testing/code-coverage`. Accessed: 25.02.2021.