Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Implementierung eines automatischen Refactorings zur Reduzierung von dupliziertem Quellcode

Dennis Maseluk

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Stefan Wagner
Betreuer/in:	M.Sc. Marvin Wyrich
Beginn am:	15. November 2018
Beendet am:	15. Mai 2019

Kurzfassung

Code-Klone sind sehr ähnliche Code-Fragmente, die die Wartung von Software erschweren und den Quellcode unnötig vergrößern. Durch ein Refactoring können Code-Klone entfernt werden, ohne das Verhalten der Software zu verändern. Dieses Refactoring kann manuell oder durch ein Tool (teils-)automatisch durchgeführt werden. In der Abteilung für Software Engineering wird ein Refactoring-Bot entwickelt, der automatisch code smells beheben kann. Die Möglichkeit, duplizierten Quellcode automatisch zu reduzieren, fehlt dem Bot derzeit aber noch. Im Rahmen dieser Masterarbeit wird dieser Bot um das Code-Klon Refactoring erweitert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

1 Einleitung

Refactorings bezeichnen Änderungen am Quellcode, die das beobachtbare Verhalten der Software nicht verändern. Häufig werden Refactorings durchgeführt, um die Qualität des Quellcodes zu erhöhen. Ein Refactoring ist duplizierten Quellcode zu entfernen. Duplizierter Quellcode, sogenannte Code-Klone, entstehen immer wieder in Softwareentwicklungen. Sie erschweren die Wartung von Software, da sie den Quellcode unnötig vergrößern und sollten deshalb vermieden werden [RBS13]. Wenn ein Code-Klon entdeckt wird kann der Softwareentwickler entweder manuell die Klone entfernen oder ein Tool zur (teils-)automatischen Entfernung verwenden.

1.1 Motivation

Häufig werden Refactorings durchgeführt, um die Qualität von Quellcode zu erhöhen. Diese Refactorings kosten aber Zeit und Arbeit, wenn man sie manuell durchführt. Deshalb ist es vorteilhaft, wenn diese Refactorings (teil-)automatisch durchgeführt werden. Der Bot, der in der Abteilung Software Engineering des Instituts für Softwaretechnologie entwickelt wird, ist in der Lage auf Basis der Ergebnisse von SonarQube automatisch Refactorings durchzuführen. Ihm fehlt aber die Funktionalität Code-Klone zu entfernen. Dies sollte geändert werden. Durch diese hinzugefügte Funktionalität kann dann der Bot automatisch die Softwarequalität erhöhen und gleichzeitig die Arbeit eines Programmierers verringern.

1.2 Ziel

Das Ziel der Arbeit ist es, diesen Bot um die automatische Reduzierung von Code-Klonen zu erweitern. Der Bot nutzt die Ergebnisse der Code-Klon-Erkennung von SonarQube und legt diese in Form eines Pull-Requests auf GitHub einem Softwareentwickler zum Review vor. Dieser muss die Änderung dann im Idealfall nur mit einem Klick akzeptieren oder ablehnen.

1.3 Beitrag

Der in dieser Masterarbeit vorgestellte Beitrag ist eine Zusammenfassung von verschiedenen Möglichkeiten zur (teils-)automatischen Entfernung von Code-Klonen. Es wird ebenfalls eine Implementierung der automatischen Code-Klon-Entfernung als Teil eines Bots in Java vorgestellt.

1.4 Aufbau der Masterarbeit

Die Masterarbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen vermittelt dem Leser Hintergrundwissen, das notwendig ist, um die folgenden Kapitel zu verstehen. Es werden Begriffe wie zum Beispiel Code-Klone und Refactorings erklärt. Ebenfalls werden die verschiedenen Arten von Code-Klonen beschrieben und deren möglichen Refactorings.

Kapitel 3 – Verwandte Arbeiten verschafft einen Überblick über ähnliche Forschungsthemen und stellt diese Arbeit in den Kontext von anderen Arbeiten.

Kapitel 4 – Implementierung ist das Hauptkapitel dieser Masterarbeit und handelt von der Implementierung des Bots. Es wird beschrieben auf welche Art der Bot implementiert wurde und welche Algorithmen verwendet wurden.

Kapitel 5 – Evaluation ist ein Kapitel, in dem die Implementierung der Code-Klon-Entfernung auf seine Nützlichkeit getestet wird. Es werden ebenfalls die Ergebnisse der Evaluation aufgelistet. Alle Informationen, die während der Evaluation gesammelt wurden, werden in Tabellen präsentiert.

Kapitel 6 – Diskussion ist ein Kapitel, das die Ergebnisse der Evaluation analysiert und interpretiert. Es wird ebenfalls ein Fazit zur Implementierung gezogen.

Kapitel 7 – Zusammenfassung und Ausblick fasst die wesentlichen Aspekte dieser Arbeit zusammen. Zusätzlich werden zukünftige Forschungsmöglichkeiten in einem Ausblick beschrieben.

2 Grundlagen

Dieses Kapitel befasst sich mit den Grundlagen von Code-Klonen und Refactorings. Es werden Begriffe erklärt, die in dieser Arbeit Verwendung finden und es wird näher darauf eingegangen wie Code-Klon Refactoring abläuft.

2.1 Code-Fragmente

Eine Sequenz von Codezeilen wird auch als Code-Fragment bezeichnet [RCK09]. Dabei kann diese Sequenz auch Kommentare beinhalten. Code-Fragmente können in unterschiedlicher Granularität sein, z.B. als ganze Funktion oder nur eine Sequenz von Befehlszeilen. Man kann Code Fragmente im Quellcode einer Software durch den Dateinamen, die Anfangszeilennummer und die Endzeilennummer identifizieren.

2.2 Code-Klone

Wenn mehrere Code-Fragmente sich sehr ähnlich sind spricht man von Code-Klonen [RZK14]. Die Definition der Ähnlichkeit bestimmt auch den Typ des Klones (siehe Abschnitt 2.3). Ira Baxter, ein Pionier in Klonerkennung, hat folgende Definition für Code Klone aufgestellt: "Clones are segments of code that are similar according to some definition of similarity" [RZK14].

Üblicherweise entstehen Code-Klone durch Copy-and-Paste von Quellcode durch einen Softwareentwickler. Dabei können auch Zeilen hinzugefügt werden. Der kopierte Code wird dann als Klon bezeichnet und die Aktivität als (Code) klonen [RBS13].

2.3 Klon-Typen

Es gibt verschiedene Arten von Code-Klonen [RCK09]. Man kann sie in folgende Typen unterteilen:

```
if (a == b) {
  c = a + b; // K1
} else {
  c = a - b; // K2
}
```

Listing 2.1: Original Code

Typ 1 - Exakte Klone [RBS13] Diese Code-Klone sind fast identisch. Sie können sich aber durch Leerzeichen, Layout und Kommentare unterscheiden.

```
if (a == b) {
  // Kommentar 1
  c = a + b;
} else {
  // Kommentar 2
  c = a - b;
}
```

Listing 2.2: Beispiel für einen Typ 1 Klon

Typ 2 - Umbenannte/parametrisierte Klone [RBS13] Diese Art von Code-Klonen ist syntaktisch identisch, bis auf Variationen in Bezeichnern, Literalen, Typen, Leerzeichen, Layout und Kommentaren.

```
if (g == f) {
  // Kommentar 1
  h = g + f;
} else {
  // Kommentar 2
  h = g - f;
}
```

Listing 2.3: Beispiel für einen Typ 2 Klon

Typ 3 - Near miss Klone [RBS13] Zusätzlich zu den Variationen eines Typ 2 Klons sind bei Type 3 Klonen Befehlszeilen entweder verändert, entfernt oder hinzugefügt worden.

```
if (a == b) {
  // Kommentar 1
  c = a + b;
  // Neuer Befehl
  b = a - c;
} else {
  // Kommentar 2
  c = a - b;
}
```

Listing 2.4: Beispiel für einen Typ 3 Klon

Type 4 - Semantische Klone [RBS13] Typ 4 Code-Klone führen die gleichen Berechnungen aus, aber sie unterscheiden sich syntaktisch.

```
switch (true) {
  // Kommentar 1
  case (a == b):
    c = a + b;
  // Kommentar 2
  case (a != b):
    c = a - b;
}
```

Listing 2.5: Beispiel für einen Typ 4 Klon

2.4 Refactoring

Bei der Aktivität des Refactorings geht es um das Verändern des Quellcodes von Software, sodass sich am Verhalten des Programmes nichts ändert [FBB+99]. Dadurch wird wird die Qualität des Codes erhöht. Der Code lässt sich leichter warten und die Lesbarkeit des Codes wird besser [STV16]. Das Refactoring wird von Entwickler meistens durch Änderungen an den Anforderungen (neue Features und Bug fixes) angestoßen und seltener durch das Finden von Code smells. Man kann als Entwickler das Refaktorieren manuell durchführen oder es von einer Software (teils-) automatisch ausführen lassen. Die Entwicklungsumgebung spielt dabei eine wichtige Rolle [STV16]. So lassen zum Beispiel IntelliJ IDEA Benutzer zu 71% autoatisch refaktorieren und Eclipse Benutzer nur zu 44%. Es gibt also Bedarf nach automatischen Refactoring.

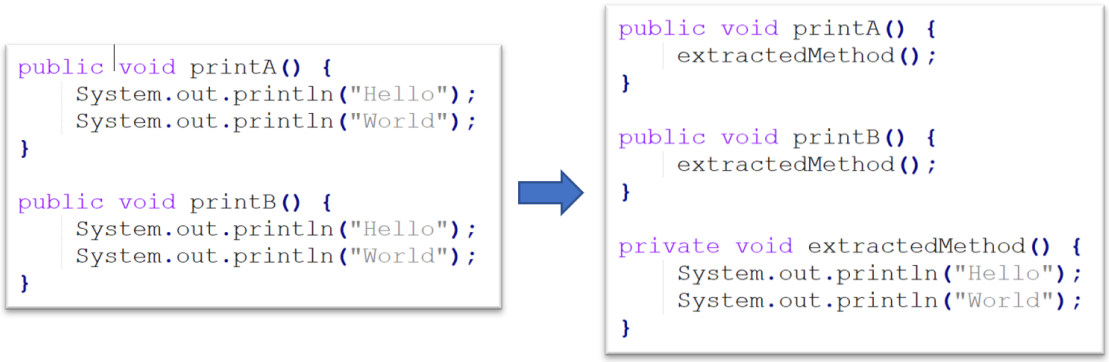

Abbildung 2.1: Beispiel für ein Code-Klon Refactoring

2.4.1 Code-Klon Refactoring

Ein Code-Klon-Szenario wird durch die Beziehungen zwischen den Klonen festgelegt. Es ist nicht möglich eine einzige Refactoring-Technik zu verwenden, um alle Code-Klon Szenarien aufzulösen. Deshalb braucht man für jedes Szenario eine spezifische Lösung. In der Arbeit von Georges Golomingi Koni-N'Sapu [Kon01] werden 8 Szenarien beschrieben in der Code-Klone erscheinen können und welche Refactoring-Techniken verwendet werden sollten, um sie zu entfernen.

Szenario 1

Das erste und das einfachste Szenario ist, wenn sich zwei Klone in derselben Methode befinden. Um diese Klone zu entfernen wird eine Extraktionsmethode in derselben Klasse erstellt. Die neue Methode enthält den Code der Klone und auf die Stelle der Klone wird ein Methodenaufruf zu der neuen extrahierten Methode erstellt. Falls sich lokale Variablen in der Methode mit den Klonen befinden, müssen diese eventuell parametrisiert werden, um keine Abhängigkeiten zu zerstören.

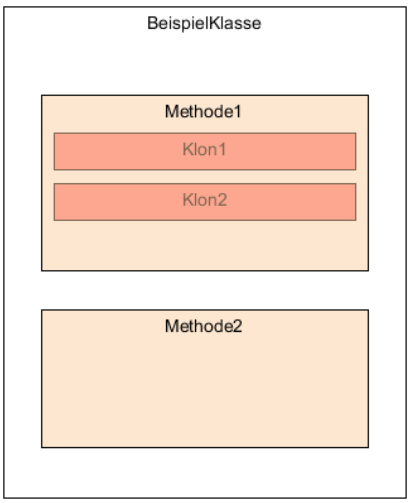

Abbildung 2.2: Code-Klone befinden sich in derselben Methode

Szenario 2

Wenn sich zwei Klone in zwei unterschiedlichen Methoden in derselben Klasse befinden. Dieses Szenario lässt sich ebenfalls durch neue extrahiert Methode lösen. Wieder muss überprüft werden ob Variablen parametrisiert werden müssen. Eine Alternative, um diese Klone zu entfernen, wäre die Refactoring-Technik, eine Methode vollständig in die andere Methode rein zu kopieren oder nur einen Methodenaufruf der anderen Methode hinzuzufügen ("Insert Method Call").

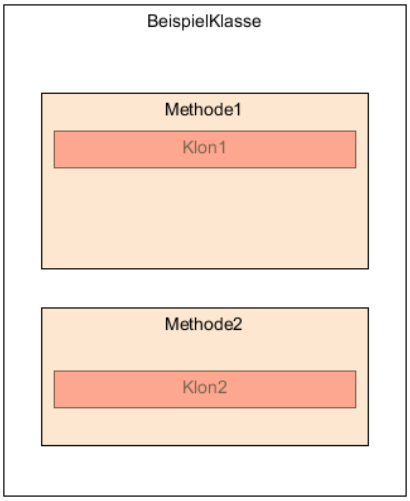

Abbildung 2.3: Code-Klone befinden sich in der selben Klasse

Szenario 3

Die Klone befinden sich in Geschwisterklassen. Geschwisterklassen sind Klassen mit derselben Elternklasse und sie befinden sich auf derselben hierarchischen Ebene. Die beste Technik die Klone zu entfernen, wäre sie in die gemeinsame Elternklasse zu ziehen.

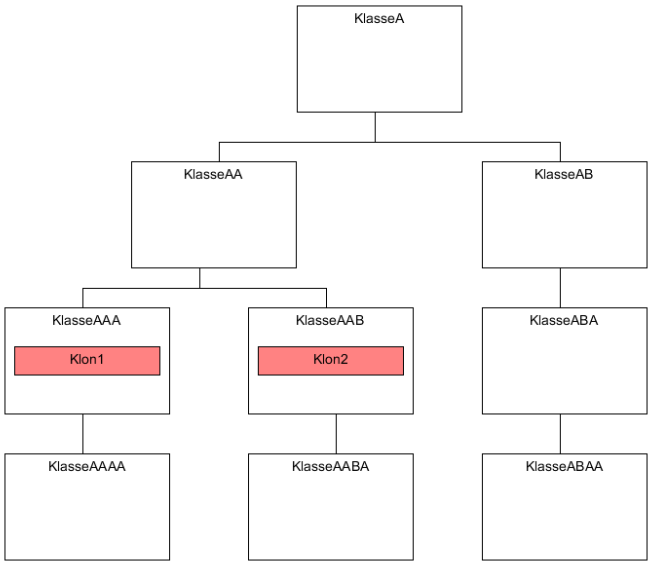

Abbildung 2.4: Code-Klone befinden sich in Geschwisterklassen

Szenario 4

In diesem Szenario befinden sich die Klone in einer Klasse und in dessen Elternklasse. In diesem Fall kann man die Klone in die Elternklasse extrahieren oder das Template Method Pattern verwenden, falls die Methodennamen der Klone identisch sind. Dafür muss eine abstrakte Vorlagenmethode in der Elternklasse definiert werden und die abweichenden Befehlszeilen in den Unterklassen.

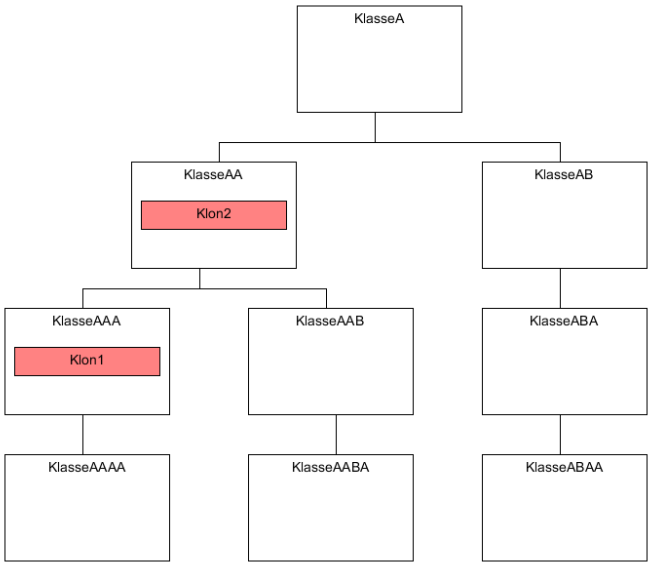

Abbildung 2.5: Code-Klone befinden sich in einer Klasse und dessen Elterklasse

Szenario 5

Im Gegensatz zu Szenario 4 erbt eine Klasse nicht direkt von der Elternklasse mit dem Klon. Es befinden sich Klassen dazwischen. Das bedeutet, dass wenn eine extrahierte Methode in der Elternklasse erstellt wird, hat das Einfluss auf die Klassen, die dazwischen liegen. Es muss daher abgewogen werden, ob es sich lohnt in die Elternklasse zu extrahieren.

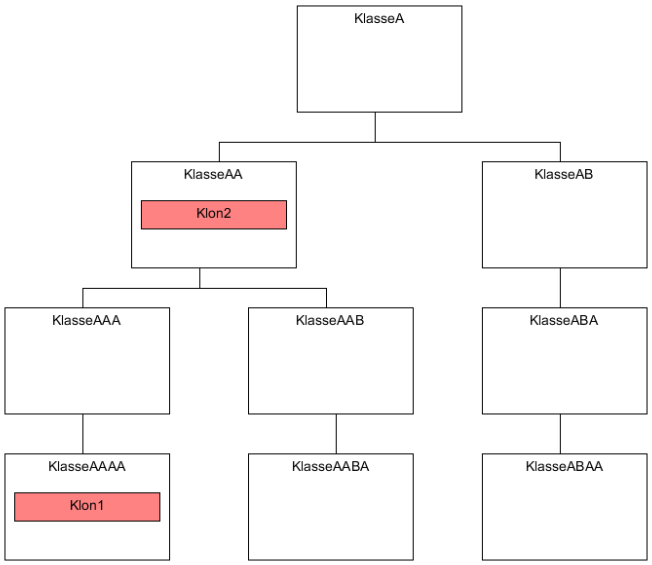

Abbildung 2.6: Code-Klone befinden sich in einer Klasse und deren Elternklasse von der sie nicht direkt erbt

Szenario 6

Die Klassen mit den Klonen befinden sich auf derselben hierarchischen Ebene und deren Elternklassen sind Geschwisterklassen. In diesem Fall muss die extrahierte Methode in die Klasse geschrieben werden, die zwei Ebene höher in der Hierarchie liegt. Wieder muss darauf geachtet werden, dass die dazwischen in der Hierarchie liegenden Klassen, von den extrahierten Code beeinflusst werden können.

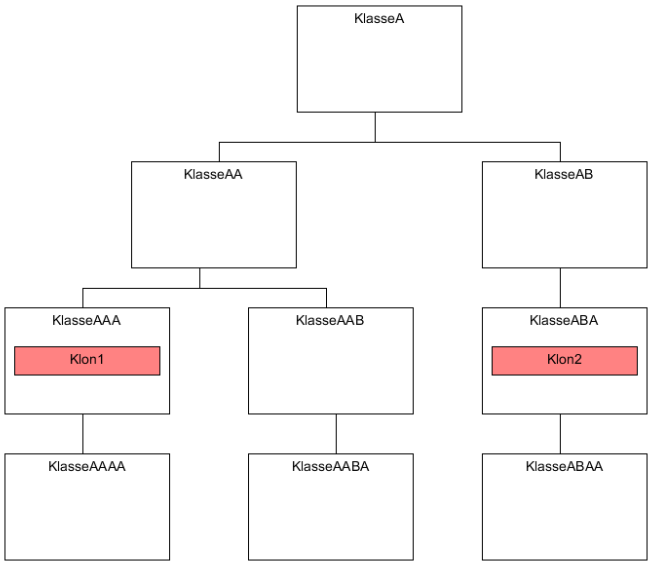

Abbildung 2.7: Code-Klone befinden sich in einer Klasse und dessen Cousinenklasse

Szenario 7

In diesem Szenario sind die Mengen der Vorfahren der Klassen mit den Klonen nicht disjunkt. Sie befinden sich in derselben Hierarchie, sind aber nicht auf der selben Ebene der Hierarchie. So wie in Szenario 6 gibt es keine ideale Lösung und es muss genau überprüft werden, wohin der Code extrahiert wird.

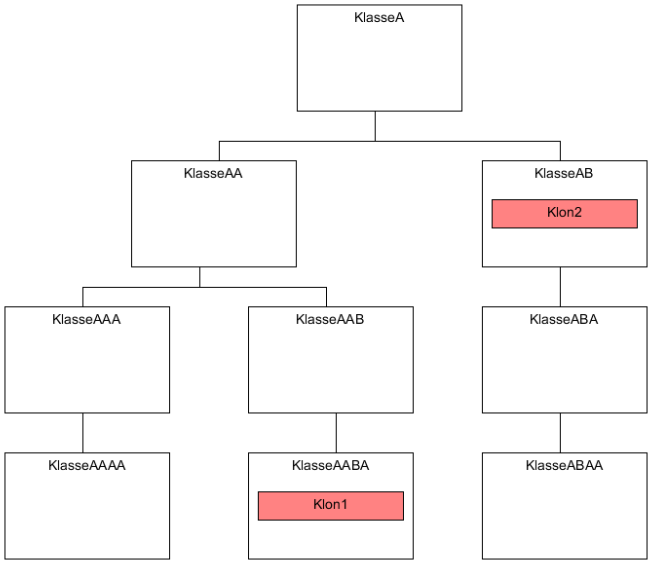

Abbildung 2.8: Code-Klone befinden sich in der selben Hierarchie

Szenario 8

Die Klassen haben keine gemeinsamen Vorfahren. In diesem Fall bleibt nur die Lösung den Klon in eine der beiden Klassen zu extrahieren oder eine neue Klasse zu erstellen, die den Klon enthält. Dann muss noch nur noch die neue Klasse in den anderen Klassen benutzt werden.

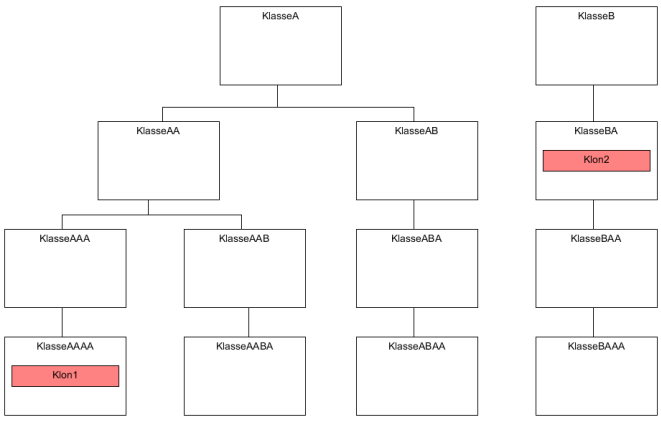

Abbildung 2.9: Code-Klone befinden sich in nicht verwandten Klassen

Bei der Entfernung der Code-Klon Befehlszeilen muss darauf geachtet werden, dass das Programm nicht zerstört wird und es keine Kompilierungsfehler auftreten. Das beobachtbare Verhalten des Programmes darf nicht verändert werden. Ebenso muss darauf geachtet werden, dass die neue hinzugefügte Methode, die richtigen Parameter und den richtigen Rückgabewert besitzt. Das bedeutet, die Code-Klon Befehlszeilen müssen, bevor sie entfernt werden, analysiert werden. In der Arbeit von Nikolaos Tsantalis et al. [TMK15] werden mehrere Vorbedingungen aufgezählt, die erfüllt werden müssen, bevor die Befehlszeilen entfernt werden können. Diese Bedingungen verhindern, dass das Refactoring die Funktionalität des Programmes verändert. Wird eine der Vorbedingungen verletzt, kann der Code-Klon nicht entfernt werden.

Alle vorhandenen unterschiedlichen Befehlszeilen, die zwischen den Code Klonen gemappt werden können, wie zum Beispiel unterschiedliche Werte für gleiche Literale, müssen parametrisiert werden. Im Wesentlichen bedeutet dies, dass die unterschiedlichen Werte als Argumente an den extrahierten Methodenaufruf übergeben werden. Da die Reihenfolge der Befehle in einem Programm wichtig ist, sollte diese bewahrt werden. Ebenfalls muss auf verschiedene Abhängigkeiten geachtet werden, im Bezug auf Daten. Folgende Abhängigkeiten können verletzt werden, wenn die falsche Reihenfolge bei der Parametrisierung übernommen wird. Eine Gegenabhängigkeit (anti-dependency) kann verletzt werden. Eine Gegenabhängigkeit findet dort statt, wo ein Wert gelesen wird und in einer Iteration später überschrieben wird [MHL95]. Eine weitere Abhängigkeit ist die Ausgabeabhängigkeit (output dependency). Hier handelt sich, um Befehlszeilen die denselben Wert überschreiben. Bei der Datenabhängigkeit (data dependency) handelt sich um eine Befehlszeile, die abhängig von dem Ergebnis einer anderen Befehlszeile ist. Ebenfalls muss darauf geachtet werden, dass die Kontrollabhängigkeit (controll dependency) nicht verletzt wird. Bei der Kontrollabhängigkeit handelt sich um Befehlszeilen, die von dem Ergebnis einer anderen Befehlszeile abhängen, ob sie ausgeführt werden oder nicht.

Bedingung 1 Die Parametrisierung der Unterschiede von Code-Klonen sollte keine existierenden Kontroll-, Daten-, Gegen- und Ausgabeabhängigkeiten verletzen [TMK15].

Anweisungen, die nicht zwischen Code-Klonen gemappt werden können, können auch nicht mit den gemappten Code-Fragmenten extrahiert werden. Deshalb müssen sie verschoben werden, entweder vor oder nach der extrahierten Methode. Die Verschiebung der Anweisungen sollte keine Abhängigkeiten brechen.

Bedingung 2 Die nicht gemappten Anweisungen sollten vor oder nach den gemappten Anweisungen verschoben werden können, ohne bestehende Daten-, Anti- und Ausgabeabhängigkeiten zu brechen. [KT14].

In Java kann eine Methode maximal den Wert einer Variablen zurückgeben. Infolgedessen sollten die duplizierten Codefragmente höchstens eine Variable an die ursprüngliche Methode zurückgeben, aus der sie extrahiert worden sind. Der Fall, dass mehrere Variablen zurückgegeben werden, wird nicht näher erläutert, da der Refactoring Bot nur auf Java Code arbeitet.

Bedingung 3 Die duplizierten Code-Fragmente sollten mindestens eine Variable des selben Types zurückgeben [KT14].

Eine break Anweisung wird benutzt, um die innerste for-, while- oder do-Schleife zu beenden. Die continue Anweisung wird verwendet, um die laufende Iteration zu überspringen. Werden diese Anweisungen gemappt, sollten auch die entsprechenden Schleifen gemappt werden, sonst werden Kompilierungsfehler ausgegeben.

Bedingung 4 Gemappte Verzweigungsanweisungen (break, continue) sollten von entsprechenden gemappten Schleifenanweisungen begleitet werden [KT14].

In manchen Fällen enthalten die Code-Klon Fragmente Variablen mit unterschiedlichen Unterklassentypen von einer gemeinsamen Superklasse. Diese Variablen sollten generalisiert werden, wenn sie in die extrahierte Methode verschoben werden. Deshalb muss drauf geachtet werden, dass diese Variablen keine Methoden in den Unterklassen aufrufen.

Bedingung 5 Gemappte Variablen mit unterschiedlichen Unterklassentypen sollten nur Methoden aufrufen, die in der gemeinsamen Oberklasse deklariert sind oder in den jeweiligen Unterklassen überschrieben werden. [TMK15].

Probleme können auch Instanzvariablen verursachen, die innerhalb der Klon-Fragmente modifiziert werden. Die Instanzvariablen werden parametrisiert, der extrahierten Methode übergeben. Jedoch modifiziert die extrahierte Methode nur den Wert des lokalen Parameters und nicht die Instanzvariable, weil in Java alle Parameter by value übergeben werden.

Bedingung 6 Die Parametrisierung von Feldern, die zu den Unterschieden von gemappten Anweisungen gehören, sind nur dann möglich wenn sie nicht modifiziert werden [TMK15].

Die zugehörigen Methodendeklarationen in den unterschiedlichen gemappten Anweisungen sollten keinen Typ void zurückgeben. Da es nicht möglich ist einen Parameter eines Typ voids in der extrahierten Methode zu erstellen.

Bedingung 7 Die Parametrisierung von Methodenaufrufen die zu den Unterschieden von gemappten Anweisungen gehören, sind nur dann möglich wenn sie keinen void Typen zurückgeben [TMK15].

Eine konditionelle return Anweisung wird benutzt, um einen Kontrollflussblock zu beenden und die Methode direkt zu verlassen. Würde diese return Anweisung in der extrahierten Methode aufgerufen, würde es die Original Methode nicht auf dieselbe Art verlassen, wie sie es vorher tat. Es gäbe Möglichkeiten dies mit extra hinzugefügten Anweisungen zu beheben, aber das würde nur die Komplexität des Codes erhöhen.

Bedingung 8 Die gemappten Anweisungen in einem Klon-Fragment sollten keine konditionellen return Anweisungen beinhalten [TMK15].

Werden alle diese Vorbedingungen eingehalten werden, kann der Code-Klon sicher entfernt werden.

3 Verwandte Arbeiten

Dieses Kapitel befasst sich mit verwandten Arbeiten, die sich um das Thema Refactoring von Code-Klonen beschäftigen. Diese Arbeiten lassen sich in zwei Kategorien einteilen. Arbeiten, die ihren Fokus nur auf das Code-Klon Refactoring legen und Arbeiten, die sich mit Tools beschäftigen, die Code-Klone refaktorieren können.

3.1 Code-Klon Refactoring

Die Arbeit von Nikolaos Tsantalis et al. [TMK15] beschreibt in Detail wie man ein Paar von Code-Klonen sicher entfernen kann, ohne das Verhalten des Programms zu ändern. Es wird insbesondere ein Ansatz vorgestellt, der die Unterschiede zwischen Klonen daraufhin beurteilt, ob sie sicher parametrisiert werden können oder nicht. Die Evaluationsergebnisse dieser Arbeit zeigen auf, dass der vorgestellte Ansatz die Klone, die als refaktorierbar beurteilt wurden, tatsächlich ohne Kompilierungsfehler sicher entfernt werden konnten.

Eine weitere Arbeit, die sich mit dem Refactoring von Code-Klonen beschäftigt ist die Arbeit von Giri Panamoottil Krishnan et al.[KT14]. Es wird ein Ansatz vorgestellt, der nicht triviale Unterschiede zwischen Code-Klonen entdeckt und sie parametrisiert. Die vorgestellte Technik kann Typ 1, Typ 2 und Typ 3 Klone refaktorisieren. Die Klone müssen also keine identische Kontrollabhängigkeitsstruktur haben, um von der Technik gefunden zu werden. Diese Technik untersucht ebenfalls eine Reihe von Vorbedingungen, um festzustellen, ob eine Klongruppe sicher refaktoriert werden kann. In der Arbeit wurde die vorgeschlagene Technik mit einem konkurrenzfähigen Klon-Refactoring-Tool verglichen und das Ergebnis war, dass der Ansatz in der Lage ist, eine größere Anzahl von refaktorierbaren Klonen zu finden.

In dem Buch "Refactoring: Improving the Design of Existing Code" von Martin Fowler et al. [FBB+99] wird ein Überblick rund um das Thema Refactorings gegeben. Es werden viele code smells aufgezählt und erklärt, unter anderem Code-Klone und welche Art man sie am besten refaktoriert. Befinden sich zum Beispiel zwei Klone in zwei Methoden derselben Klasse, dann muss nur eine Extraktionsmethode erstellt werden und die Methode von beiden Klonstellen aus aufgerufen werden. Ein weiteres häufiges Code-Klon Szenario ist, wenn Klone sich in zwei Geschwisterunterklassen befinden. Sind die Klone identische Methoden, können sie in die Elternklasse hochgezogen werden. Wenn der Code ähnlich, aber nicht gleich ist, muss eine Extraktionsmethode in der Elternklasse erstellt werden. Wenn sich Code-Klone in zwei unabhängigen Klassen befinden, sollte eine Klasse eine Extraktionsmethode erhalten und die andere Klasse diese dann verwenden. Eine weitere Möglichkeit ist, dass die Extraktionsmethode in eine dritte Klasse geschrieben wird. Der Autor verweist darauf, dass man selbst entscheiden muss, wo die Methode am meisten Sinn macht.

3.2 Code-Klon Refactoring Tools

In der Arbeit von Davood Mazinanian et al.[MTSV16] wird das Tool JDeodorant vorgestellt. Es wurde als Eclipse[1] Plug-in entwickelt und bietet Funktionen für die Analyse und das Refactoring von nicht trivialen Code-Klonen in Java-Projekten. Es ist in der Lage vom Benutzer ausgewählte Klonpaare zu untersuchen und in einer Visualisierung die Unterschiede hervorzuheben, die parametrisiert werden können oder nicht. Werden keine Vorbedingungen verletzt kann JDeodorant die Klone automatisch entfernen. Das Tool beschränkt sich aber nicht nur auf Code-Klone, sondern kann auch weitere code smells entfernen, wie zum Beispiel Gottklassen oder zu lange Methoden.

Es gibt Studien, die sich mit der Verwendung von Refactoring Tools befassen, wie die von Mohsen Vakilian et al.[VCN+12]. Die Studie zeigt auf, dass Programmierer einige automatische Refactorings aus verschiedenen Gründen nicht verwenden. Entweder weil der Aufwand des Lernens oder des Konfigurierens zu groß ist oder weil sie sich derer einfach nicht bewusst sind. Problematisch sind ebenfalls Refactoring Tools dessen Ergebnis nicht vorhersehbar ist. Die Studie zeigt aber auch auf, dass Programmierer nicht völlig abgeneigt von Refactoring Tools sind, solange sie gut zu bedienen und deren Ergebnisse vorhersehbar sind.

[1] https://www.eclipse.org/

4 Implementierung

Dieses Kapitel beschäftigt sich mit der Implementierung des Code-Klon-Refactorings. Die Implementierung handelt sich um eine Erweiterung des Refactoring Bots, der in der Abteilung SE entwickelt wird. Es wird im Detail beschrieben wie der Algorithmus des Code-Klon-Refactorings abläuft und erklärt wieso bestimmte Entscheidungen bei der Implementierung getroffen wurden.

4.1 Refactoring-Bot

Der Refactoring-Bot aus der Abteilung Software Engineering[1] führt automatische Refactorings durch. Es handelt sich um eine Spring Boot[2] Applikation und man benutzt eine Swagger Benutzeroberfläche, um ihr zu interagieren. Der Bot ist so implementiert, das er um beliebige Refactorings erweitert werden kann. Er nutzt die Ergebnisse einer statischen Codeanalyse und basierend darauf führt er die Refactorings durch. Das Ergebnis des Refactorings wird dann als Pull-Request dem Entwickler auf GitHub vorgelegt. Der Entwickler kann daraufhin den Pull-Request akzeptieren oder ablehnen. Es ist ebenfalls möglich durch Kommentare in natürlicher Sprache in den vorgeschlagenen Pull-Requests zu interagieren.

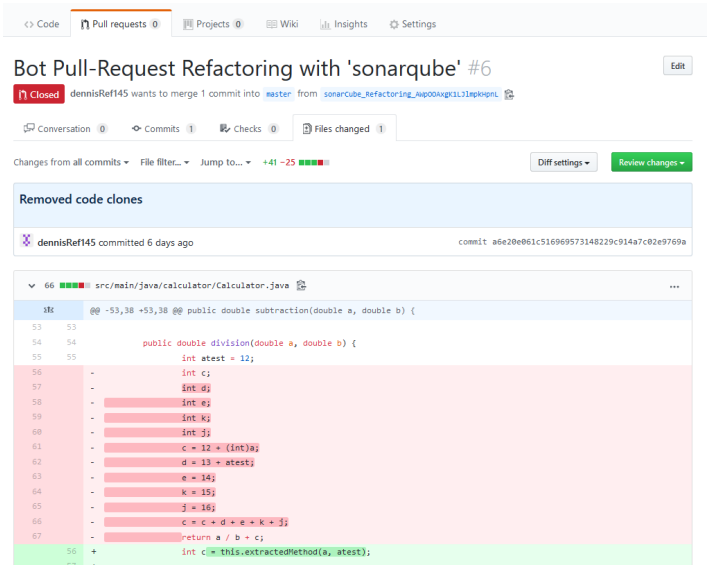

Abbildung 4.1: Beispiel eines Bot Pull-Requests

[1] https://github.com/Refactoring-Bot/Refactoring-Bot
[2] https://spring.io/projects/spring-boot

Sobald man den Bot startet, wird die Swagger Benutzeroberfläche in einem Browser aufgerufen. Man kann dann eine Bot-Konfiguration erstellen. In dieser Git Konfiguration werden Informationen festgelegt, wie zum Beispiel den verwendeten Analysis Service, Repository Service und den Namen des Projekts. In dieser Masterarbeit wurde SonarCloud[3] als Analysis Service verwendet und GitHub als Repository Service.

4.2 Algorithmen und Formeln

Das Code-Klon-Refactoring wurde in einem Algorithmus implementiert, das aus verschiedenen Schritten besteht. In der Abbildung 4.2 ist ein Überblick über die einzelne Schritte des Algorithmus zu sehen. Wenn man den Bot ausführt und die Ergebnisse der statischen Code Analyse erhält und sich darin ein Issue mit der Regel "common-java:DuplicatedBlocks" befindet, wird der Code-Klon-Refactoring Code ausgeführt. SonarQube kann nur Typ 1 und Typ 2 Klone finden, die mindestens 10 Codezeilen umfassen. Bei Typ 2 Klonen findet es nur Klone, die unterschiedliche Literale besitzen. Das heißt es würde nicht alle Typ 2 Klone finden, wie zum Beispiel ein Klon mit unterschiedlichen Variablennamen.

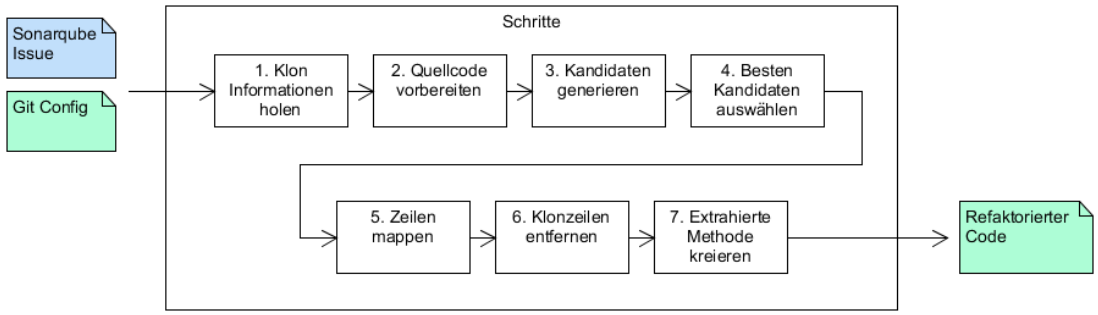

Abbildung 4.2: Übersicht über den implementierten Code-Klon-Refactoring Algorithmus

Im ersten Schritt des Algorithmus werden die Informationen über die Code-Klone über die SonarQube Web API geholt. Dies ist notwendig, da das Issue für duplizierten Code die Information über die Klone nicht enthält. Mit den separat erhaltenen Code-Klon Informationen lässt sich nun überprüfen, ob sich der Klon überhaupt vom Algorithmus entfernen lässt. Der Algorithmus kann nur Klone entfernen, die sich innerhalb einer Methode befinden. Das bedeutet, werden Klassenvariablen oder mehrere Methoden gemeinsam als Klon markiert, bricht der Algorithmus ab und es wird daraufhin kein Refactoring durchgeführt. Der Algorithmus ist in der Lage Klongruppen mit mehr als zwei Klonen zu refaktorisieren.

Ist die Überprüfung erfolgreich, ob der Klon refaktorierbar ist, wird in den nächsten Schritt des Algorithmus übergegangen. In diesem Schritt werden Vorbereitungen getroffen, die notwendig sind, um den dritten Schritt erfolgreich auszuführen. Der Quellcode der Datei mit dem Code-Klon muss von allen Dependencies bereinigt werden. Dependencies wie zum Beispiel eine Klasseninstanz einer

[3] https://sonarcloud.io/about

anderen Klasse als die der eigenen Datei oder bestimmte import Befehlszeilen. Diese Dependencies können zu Fehlern bei der Ausführung des dritten Schrittes führen. Es wird also im zweiten Schritt eine temporäre Datei erstellt mit dem vorbereiteten Quellcode. Es werden fast alle import Befehlszeilen entfernt. Ebenfalls wird der Code für die Elternklasse und das Interface entfernt. Um den Code Fluss nicht zu zerstören, werden alle Verweise auf andere Klassen erfasst und als eine neue innere Klasse der Datei hinzugefügt. Die hinzugefügten inneren Klassen besitzen dann die passenden gefundenen Methodenaufrufe, Klassenvariablen und Konstruktoren. Um alle Klassen zu finden, wurde ein "PrepareCode" Algorithmus implementiert. Man übergibt dem Algorithmus eine CompilationUnit, die iterativ durch alle Nodes durchläuft. Während alle Nodes der CompilationUnit durchlaufen werden, werden sie gleichzeitig analysiert. Es wird überprüft welches MetaModel sie haben und darauf basierend wie mit ihnen umgegangen wird. Besitzt zum Beispiel ein Node das MetaModel "variableDeclarationExprMetaModel", dann kann dieser Node Informationen für inneren Klassen beinhalten. Die neue Datei, die durch diesen Algorithmus erstellt wird, kann trotzdem noch Fehler beinhalten. Ein Fehler ist zum Beispiel "inconvertible types", der dann auftritt, wenn eine innere Klasse in eine andere gecastet wird. Befindet sich so ein Fehler in der neuen Datei, dann wird beim parsen dieser Datei eine Exception geworfen und das Refactoring wird abgebrochen.

Nachdem die neue vorbereitete Datei erstellt wurde, wird zum dritten Schritt übergegangen. Es werden nun Kandidaten generiert, die sicher aus den Methoden entfernt werden können, ohne das beobachtbare Verhalten des Programms zu ändern. Hierfür muss zuerst in der neuen vorbereiteten Datei nach den Klonen gesucht werden, da sich die Zeilen möglicherweise verändert haben. Um die möglichen extrahierbaren Kandidaten zu generieren, werden Teile von dem Code von Johannes Hubert verwendet. Seine Arbeit befasst sich mit dem Refaktorieren von zu langen Methoden und es enthält einen Algorithmus, um herauszufinden welche Teile vom Quellcode sicher entfernt werden können. Er verwendet unter anderem das Checker Framework[4]. Dieses Framework kann nicht fehlerfrei ausgeführt werden, wenn es eine Datei analysiert, die Dependencies enthält. Aus diesem wird die vorbereitete Datei und nicht die Originaldatei dem Checker Framework übergegeben. Sobald die Kandidaten generiert wurden, wird in den nächsten Schritt übergegangen, dem Selektieren des besten Kandidaten.

Alle Kandidaten werden analysiert und mit einer Zahl bewertet. Die Beurteilung ergibt sich aus zwei Zahlen. Die erste Zahl ("lengthScore"), die ermittelt wird, ist die Länge des Kandidaten. Je mehr Zeilen der Kandidat enthält, umso größer wird die Zahl. Die zweite Zahl ("cloneExceedScore") kann entweder 0 oder 1 sein. Sie wird mit 0 bewertet, wenn der Kandidat eine Zeile enthält, die nicht Teil eines Klons ist und mit 1, wenn der Kandidat nur Zeilen enthält, die Teil eines Klons sind.

```
candidate.score = lengthScore * cloneExceedScore;
```

Listing 4.1: Kandidaten Bewertung

Beide Zahlen werden multipliziert, um eine endgültige Bewertung für den Kandidaten zu ermitteln. Dadurch ist der höchstbewertete Kandidat, der der die meisten Zeilen eines Klons abdeckt und keine Zeile außerhalb der Klons beinhaltet.

[4] https://checkerframework.org/

Der Kandidat beinhaltet die Information welche Befehlszeilen extrahiert werden können, den Outputvariablen und den Inputvariablen. Bei den Inputvariablen scheint es aber noch Fehler zu geben, da diese nie berechnet werden. Das bedeutet, ich musste separat noch einmal den Klon analysieren, um alle Inputvariablen zu erhalten. Diese werden nämlich als Parameter, der neuen extrahierten Methode übergeben. Mit der Information, welche Zeilen des Klons extrahiert werden können, geht es in den nächsten Schritt über. Die Zeilen von der vorbereiteten Datei müssen auf die Zeilen der Orginaldatei gemappt werden. Da SonarQube auch Typ 2 Klone mit unterschiedlichen Literalen finden kann, können die Befehlszeilen nicht einfach miteinander verglichen werden. Zuerst müssen alle unterschiedliche Literale in den Klonen gefunden werden. Die Information über die Position der Literale wird dann beim mapping der Kandidatzeilen und der Zeilen der Originaldatei berücksichtigt. Zusätzlich werden die Informationen über die unterschiedlichen Literale gesammelt, um sie später als Parameter der extrahierten Methode zu übergeben.

Im sechsten Schritt werden die Klonzeilen entfernt. Dafür wird einer Schleife alle Dateien, die einen Klon enthalten eingelesen. Als nächstes wird nach den Nodes mit den Klonzeilen gesucht und durch eine Visitor Klasse entfernt. Zusätzlich wird ein Methodenaufruf in die erste entfernte Zeile geschrieben. Die Information für den Rückgabewert des Methodenaufrufes wird von dem ausgewählten Kandidaten entnommen und die Parameter aus den Informationen, die in den vorigen Schritten gesammelt wurden.

Im letzten Schritt wird die eigentliche extrahierte Methode ans Ende der Klasse hinzugefügt. Die Rückgabewert wird wieder aus dem Kandidaten entnommen und die Parameter aus den Informationen, die in den vorigen Schritten gesammelt wurden. Die Befehlszeilen, die in die extrahierte Methode eingefügt werden, werden immer vom ersten Klon einer Klongruppe übernommen. Der Name der extrahierten Methode ist auf "extractedMethod" festgelegt. Nachdem hinzufügen der Methode ist der Algorithmus beendet und die neuen Dateien mit dem refaktorierten Code werden als Pull-Request auf GitHub hochgeladen.

5 Evaluation

In diesem Kapitel wird die Implementierung evaluaiert. Die Evaluation beschäftigt sich vorallem mit der Frage der Nützlichkeit der Implementierung. Wie viele Projekte mit Code-Klonen lassen sich refaktorieren? Wie viele Refactorings sind erfolgreich und korrekt?

5.0.1 Methodik

Die Nützlichkeit wird folgendermaßen getestet. Es werden 75 zufällige Projekte aus der Sonarcloud[1] ausgewählt. Auf Sonarcloud werden Ergebnisse für statische Code Analysen gehostet und können frei untersucht werden. Es gibt auf der Plattform die Möglichkeit nach Issues zu suchen. Die 75 zufälligen Projekte, die für die Evaluation verwendet werden, stammen aus dem Suchergebnis mit dem Issue "Source files should not have any duplicated blocks". Die Projekte werden nacheinander unterucht, so wie sie in dem Suchergebnis vorgeschlagen werden. Es werden jeweils maximal 4 Issues eines Projekts untersucht. Hat ein Projekt weniger als 4 Issues werden vorhandenen Issues untersucht und dann wird zum nächsten Projekt gewechselt. Falls in einem Issue ein Klon auf eine schon vorher untersuchte Datei verweist, dann wird das Issue übersprungen. Es wird immer nur die erste Klongruppe in einem Issue untersucht, da der Refactoring-Bot ebenfalls nur die erste Klongruppe refaktoriert. Bei de Untersuchung der Issues wird der Bot einmal ausgeführt, falls überhaupt noch das GitHub Projekt existiert und der Klon refaktorierbar ist. Wurde der Bot ausgeführt, wird überprüft ob das Refactoring korrekt ist und der Programmcode nicht zerstört wurde. Ist der Programmcode nach dem Refatoring noch ausführbar? Ist der Code Klon vollständig extrahiert worden? Sind beim Refactoring Fehler aufgetreten? Diese Fragen werden während der Evaluation beantwortet.

5.0.2 Ergebnisse

Die Ergebnisse werden in Tabellen präsentiert. Die Tabellen bestehen aus folgenden Spalten:

Nr. Die Nummerierung für die Einträge in der Tabelle.

Projekt Enthält den Namen des Projekts mit der Code Klon Datei.

Datei Enthält den Namen der Code Klon Datei.

Möglich Enthält die Information, ob der Klon überhaupt vom Refactoring Bot entfernt werden kann. Klone, die sich nicht innerhal einer Methode befinden, können nicht refaktoriert werden.

[1] https://sonarcloud.io

Art Enthält die Information, ob die Klassen mit den Klonen irgendwie verwandt sind. Die Art der Verwandtschaft wird durch eine Zahl symbolisiert.

- **0** Nicht verwandt
- **1** Klone in der gleichen Klasse
- **2** Geschwisterklassen
- **3** Elternklasse/Kindklasse
- **4** Gleiche Hierarchie

Erfolgreich Beinhaltet die Information ob das Refactoring erfolgreich war oder nicht. Das heißt das keine Fehler beim Refactoring geworfen wurden und das Refactoring korrekt ist.

Größe Enthält die Information, wie viele Zeilen der größte Klon einer Klongruppe umfasst.

Nr.	Projekt	Datei	Möglich	Art	Erfolgreich	Größe
1	testng	Assert.java	ja	1	nein	26
2	testng	TestRunner.java	nein	0	-	17
3	testng	AfterClass.java	nein	0	-	47
4	testng	AfterGroups.java	nein	0	-	53
5	PMD	ASTUserClass.java	nein	2	-	14
6	PMD	ApexParserVisitor Adapter.java	nein	0	-	448
7	PMD	DumpFacade.java	ja	0	nein	15
8	PMD	AbstractStatistical ApexRule.java	nein	4	-	16
9	AssertJ fluent assertions	AbstractBooleanArray Assert.java	nein	2	-	162
10	AssertJ fluent assertions	Assertions.java	nein	0	-	141
11	AssertJ fluent assertions	AssertionsForClass Types.java	nein	0	-	101
12	AssertJ fluent assertions	AssertionsForInterface Types.java	nein	0	-	157
13	checkstyle	OuterTypeFilename Check.java	nein	2	-	16

Tabelle 5.1: Ergebnisse der Evaluation von 1 bis 13

Nr.	Projekt	Datei	Möglich	Art	Erfolgreich	Größe
14	checkstyle	TodoComment Check.java	nein	2	-	16
15	checkstyle	TrailingComment Check.java	nein	2	-	17
16	checkstyle	UpperEllCheck.java	nein	2	-	14
17	inmarsat PR	InmarsatPlugin.java	ja	0	nein	13
18	inmarsat PR	InmarsatMessage Retriever.java	ja	0	nein	13
19	AvatarMod crows/bison-fix	ModelIceShield.java	nein	2	-	36
20	AvatarMod crows/bison-fix	ModelSandPrison.java	nein	2	-	15
21	AvatarMod crows/bison-fix	RenderAirBubble.java	nein	2	-	60
22	AvatarMod crows/bison-fix	RenderCloudburst.java	ja	2	nein	16
23	Food Truck Manager develop	User.java	nein	0	-	15
24	Food Truck Manager develop	Leaderboard.java	nein	0	-	37
25	oxygen-dita-prolog-updater	AuthorPageDocument Util.java	ja	0	nein	21
26	sem-project revert-204-coverage-improvement	Leaderboard.java	nein	0	-	37
27	sem-project v0.4.0	Leaderboard.java	nein	0	-	37
28	sem-project coverage-improvement-3	Leaderboard.java	nein	0	-	37
29	sem-project coverage-improvement	Leaderboard.java	nein	0	-	37

Tabelle 5.2: Ergebnisse der Evaluation von 14 bis 29

Nr.	Projekt	Datei	Möglich	Art	Erfolgreich	Größe
30	sem-project state-coverage	Leaderboard.java	nein	0	-	37
31	sem-project master	Leaderboard.java	nein	0	-	37
32	sem-project merge-into-devel	Leaderboard.java	nein	0	-	37
33	mdm-agent	DataStorage.java	nein	0	-	29
34	mdm-agent	EditUserActivity.java	nein	2	-	44
35	mdm-agent	ConnectionHTTP.java	ja	1	nein	39
36	mdm-agent	EnvInfoAbout.java	nein	0	-	12
37	release	Config.java	ja	1	nein	17
38	release	JdiTestResultError.java	nein	2	-	33
39	release	VMReference.java	ja	1	nein	13
40	release	ConstructorDialog.java	ja	2	nein	14
41	Laurentius-cef-plugin	CEFInInterceptor.java	nein	0	-	23
42	Laurentius-cef-plugin	CEFOutMailEvent Listener.java	nein	0	-	36
43	Laurentius-cef-plugin	MEPS InIntercep-tor.java	ja	0	nein	26
44	Laurentius-meps-plugin	MEPSStatus Sub-mitter.java	ja	0	nein	11
45	noti	RequestContext.java	nein	0	-	15
46	noti	MetadataGet Inter-ceptor.java	nein	2	-	17
47	noti	Error Representa-tion.java	nein	2	-	39
48	noti	Audience.java	nein	2	-	64
49	ProofScript Parser	DebuggerMain.java	ja	1	nein	37
50	ProofScript Parser	InspectionView.java	nein	2	-	18
51	ProofScript Parser	SequentView.java	ja	2	nein	26
52	ProofScript	WelcomePane.java	nein	2	-	56
53	PresentAPI	EventModel.java	nein	2	-	15
54	contest	Evento.java	nein	0	-	18

Tabelle 5.3: Ergebnisse der Evaluation von 30 bis 54

Nr.	Projekt	Datei	Möglich	Art	Erfolgreich	Größe
55	contest	Pessoa.java	nein	0	-	15
56	contest	Revisao.java	nein	0	-	23
57	filescanner-engine	ByteArraySpec.java	nein	4	-	20
58	filescanner-engine	ConditionalCompositeSpec.java	nein	0	-	18
59	filescanner-engine	DWordArraySpec.java	nein	2	-	20
60	filescanner-engine	EncodedInputSpec.java	nein	2	-	15
61	OrderNao	OrderController.java	ja	1	nein	16
62	OrderNao	OrderDao.java	ja	0	nein	15
63	OrderNao	TrackOrder RowMapper.java	ja	0	nein	14
64	Microservice Framework update-test-utils-master	RequesterProducer.java	nein	0	-	24
65	Microservice Framework update-test-utils-master	AbstractJdbc Repository.java	nein	0	-	11
66	Microservice Framework update-test-utils-master	JdbcDataSource Provider.java	nein	0	-	15
67	Microservice Framework update-test-utils-master	JsonObjectBuilder Wrapper.java	nein	0	-	127
68	EvE-Online Utilities	AddBluePrintCommand.java	nein	0	-	16
69	EvE-Online Utilities	AddItemCommand	nein	0	-	30
70	ICVServices	ICVAccessToken.java	nein	0	-	16
71	JHub	Owner.java	nein	0	-	50
72	gradle-jrebel-plugin	RebelDslClasspath-Resource.java	nein	0	-	23
73	gradle-jrebel-plugin	RebelClasspath-Resource.java	nein	0	-	46

Tabelle 5.4: Ergebnisse der Evaluation von 55 bis 73

Nr.	Projekt	Datei	Möglich	Art	Erfolgreich	Größe
74	Gumga Framework Backend develop	GumgaGeneric-Repository.java	ja	1	nein	14
75	Gumga Framework Backend develop	GumgaService.java	nein	2	-	63

Tabelle 5.5: Ergebnisse der Evaluation von 74 bis 75

6 Diskussion

In diesem Kapitel werden die Schlüsse aus der Evaluation gezogen und die Frage der Nützlichkeit beantwortet.

Insgesamt konnten von den untersuchten Issues nur maximal 18 refaktoriert werden. Die restlichen Issues hat Klone, die sich nicht nur innerhalb einer Methode befanden, sondern auch andere Methoden oder Klassenvariablen einschlossen. Das bedeutet das im besten Fall nur ungefähr ein Viertel refaktoriert werden kann. Zusätzlich zu den 18 möglichen refaktorierbaren Issues konnte keine erfolgreich refaktoriert werden. Bei allen Versuchen wurde die vorbereitete Datei nicht korrekt erstellt und es wurden beim parsen der Datei Exceptions geworfen. Damit lässt sich nur eine Schlussfolgerung zur Nützlichkeit der Implementierung ziehen. Sie ist bei echten Dateien nicht einsetzbar.

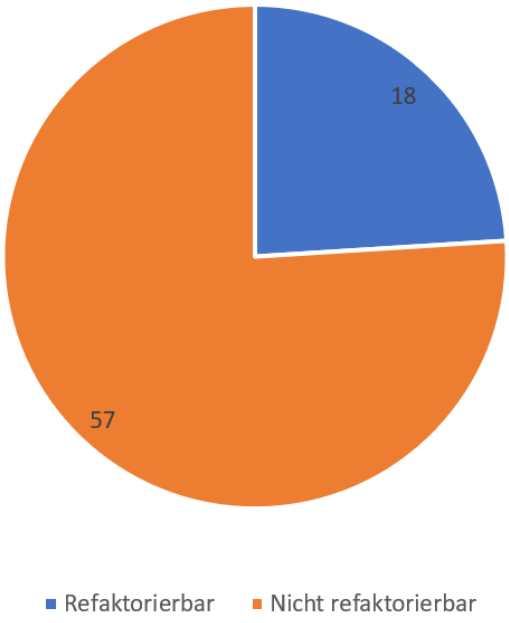

Abbildung 6.1: Kuchendiagramm mit Daten der Refaktorierbarkeit der untersuchten Issues

Interessant ist auch die Verteilung der Verwandtschaften der Klon-Klassen. Über die Hälfte (ca. 57%) der untersuchten Klone befinden sich in nicht verwandten Klassen. Auf dem zweiten Platz liegen die Geschwisterklassen mit ca. 30%. Klone in der gleichen Klasse kommen auf ca. 9% und Klone in der gleichen Hierachie ca. 3%. Klone in einer Eltern/Kind Verwandtschaft wurden nicht gefunden. Das der Anteil an nicht verwandten Klassen so hoch ist, könnte sich dadurch erklären

lassen, das diese sich am schwersten refaktorieren lassen. Ein Entwickler könnte es für überflüssig halten eine neue Klasse für eine extrahierte Methode zu erstellen und deshalb beläst er es bei dem Klon. Die andere Alternative ist es, eine extrahierte Methode in einer der beiden nicht verwandten Klassen zu schreiben. Das könnte ebenfalls dem Entwickler nicht gefallen, eine Abhängigkeit zwischen zwei nicht verwandten Klassen entstehen zu lassen. Das wäre eine Erklärung wieso diese Art von Klonen am häufigsten in der Evaluation aufkam. Einen Zusammenhang zwischen der Größe der Klone und der Art der Verwandschaft lässt sich nicht erkennen.

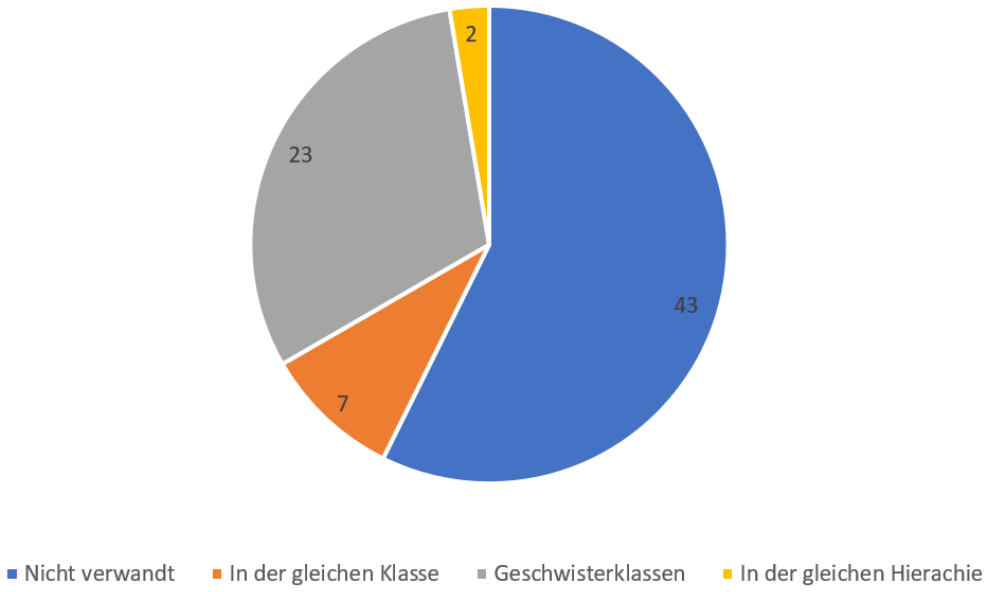

Abbildung 6.2: Kuchendiagramm mit Daten der Verwandtschaft der untersuchten Klone

7 Zusammenfassung und Ausblick

Im Rahmen dieser Masterarbeit wurde eine Erweiterung für den Refactoring Bot, der in der Abteilung Software Engineering des Instituts für Softwaretechnologie entwickelt wird, implementiert. Es wurde eine zusätzliche Code Klon Refactoring Funktion hinzufügt. Der Bot sollte damit in der Lage sein durch SonarQube gefundene Code Klone zu refaktorieren. Es lassen sich aber nur Klone innerhalb einer Methode refaktorieren. Ebenfalls beschränkt sich das Refactoring nur auf Typ 1 und Typ 2 Klone, da die statische Analyse Plattform SonarQube nur diese Typen finden kann. Der Algorithmus für das Code Klon Refactoring besteht aus mehreren Schritten. Ein Schritt benutzt den Checkerframework, um Kandidaten zu generieren, die es ermöglichen sicher Code zu entfernen. Dieses Framework hat aber während der Entwicklung viele Probleme verursacht, da es nicht mit Dateien arbeiten kann, die bestimmte Abhängigkeiten haben. Es musste eine Lösung gefunden werden, um das Checkerframework fehlerfrei verwenden zu können. Da der Code für die Kandidatengenerierung von einem anderen Studenten implementiert und verwendet wurde, wurde von mir beschlossen eine Umgehungslösung zu implementieren. Jede Datei, die dem Checkerframework übergeben wird, wird vorher von allen Abhängigkeiten befreit. Zum Beispiel wird der import einer Klasse entfernt und stattdessen eine innere Klasse hinzugefügt. Diese Lösung funktionierte in Tests mit kleineren Dateien. Während der Evaluation stellte sich heraus, dass dies mit komplexeren Dateien zu Problemen führt, sodass kein einziges erfolgreiches Refactoring durchgeführt werden konnte. Eine weitere Erkenntnis lässt sich durch die untersuchten Klone gewinnen. Die häufigste Art von Klonen sind die von nicht verwandten Klassen. Das lässt sich möglicherweise dadurch erklären, dass sie am schwersten zu refaktorisieren sind.

Ausblick

Um das Code-Klon-Refactoring des Bots fehlerfrei durchführen zu können, muss entweder an der implementierten Umgehungslösung weitergearbeitet werden oder das Checkerframework komplett ersetzt werden. Abseits vom Checkerframework kann auch die Technik der Code-Klon Entfernung erweitert werden. Bis jetzt können nur Klone innerhalb von Methoden refaktoriert werden. Man könnte das Refactoring von Klonen, die aus Klassenvariablen und/oder mehreren Methoden bestehen, ergänzen. Bis jetzt wird auch nur die extrahierte Methode in eine der Klonklassen geschrieben. Eine zusätzliche Erweiterung wäre, wenn die extrahierte Methode in eine verwandte Klasse geschrieben werden könnte.

Literaturverzeichnis

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999 (cit. on pp. 19, 29).

[Kon01] G. G. Koni-N'Sapu. "A scenario based approach for refactoring duplicated code in object oriented systems." In: *Master's thesis, University of Bern* (2001) (cit. on p. 20).

[KT14] G. P. Krishnan, N. Tsantalis. "Unification and refactoring of clones." In: *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE. 2014, pp. 104–113 (cit. on pp. 26, 29).

[MHL95] D. E. Maydan, J. L. Hennessy, M. S. Lam. "Effectiveness of data dependence analysis." In: *International Journal of Parallel Programming* 23.1 (1995), pp. 63–81 (cit. on p. 25).

[MTSV16] D. Mazinanian, N. Tsantalis, R. Stein, Z. Valenta. "JDeodorant: clone refactoring." In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2016, pp. 613–616 (cit. on p. 30).

[RBS13] D. Rattan, R. Bhatia, M. Singh. "Software clone detection: A systematic review." In: *Information and Software Technology* 55.7 (2013), pp. 1165–1199 (cit. on pp. 15, 17–19).

[RCK09] C. K. Roy, J. R. Cordy, R. Koschke. "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach." In: *Science of computer programming* 74.7 (2009), pp. 470–495 (cit. on p. 17).

[RZK14] C. K. Roy, M. F. Zibran, R. Koschke. "The vision of software clone management: Past, present, and future (keynote paper)." In: *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. IEEE. 2014, pp. 18–33 (cit. on p. 17).

[STV16] D. Silva, N. Tsantalis, M. T. Valente. "Why we refactor? confessions of github contributors." In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2016, pp. 858–870 (cit. on p. 19).

[TMK15] N. Tsantalis, D. Mazinanian, G. P. Krishnan. "Assessing the refactorability of software clones." In: *IEEE Transactions on Software Engineering* 41.11 (2015), pp. 1055–1090 (cit. on pp. 25–27, 29).

[VCN+12] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, R. E. Johnson. "Use, disuse, and misuse of automated refactorings." In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press. 2012, pp. 233–243 (cit. on p. 30).

Alle URLs wurden zuletzt am 10. 05. 2019 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift