Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Gruppierung und Priorisierung von Code Smells für das automatische Refactoring

Marvin Christian Knodel

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Stefan Wagner
Betreuer:	Marvin Wyrich, M.Sc.
Beginn am:	1. Oktober 2019
Beendet am:	1. April 2020

Kurzfassung

Automatisch durchgeführte Refactorings sind eine effiziente Methode, um Code Smells zu beheben und dabei keine neuen Fehler im Code unterzubringen. Der Refactoring-Bot des ISTE SE ist so eine Methode für das automatische Refactoring, er refactored Code Smells und erstellt dann Pull Requests für die Refactorings in dem Repository. Der Bot arbeitet mit den Befunden von SonarQube und erstellt pro Befund, den er gerefactored hat einen Pull Request im Repository. Da jedoch viele einzelne Pull Requests schwer zu überblicken sind, soll in dieser Arbeit durch priorisieren und gruppieren der Code Smells die Arbeit mit den vom Bot erzeugten Pull Requests verbessert werden. Ob die implementierte Gruppierung und Priorisierung dann tatsächlich eine Verbesserung für Entwickler darstellt, soll in einer Studie herausgefunden werden. In der Studie haben viele Probanden angegeben, dass sie die Refactorings lieber nach Klasse und der Art des Refactorings gruppiert haben wollen. Die implemetierte Priorisierung wurde von den meisten Probanden gut befunden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1. Einleitung

1.1. Motivation

Software enthält durch ihre Größe und Komplexität oft sogenannte Code Smells, Hinweise auf unsauber geschriebene und schlecht designte Code-Stellen, die sich negativ auf die Qualität des Quellcodes auswirken [FFZR15; Fow18; SYA+12]. Durch diese schlechten Designentscheidungen wird die Software fehleranfälliger und schlechter wartbar. Da man Code Smells oft nicht verhindern kann und sie oft erst beim erneuten Bearbeiten der Code-Stelle auffallen, oder sogar erst beim Bearbeiten des Codes entstehen, gibt es Möglichkeiten diese Code Smells im Nachhinein zu beheben, dies passiert oft durch sogenannte Refactorings. Nach Fowler ist Refactoring "[...] the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [Fow18](p. 9).

Das automatisierte Durchführen von Refactorings ist ein Weg, die Effizienz von Refactorings zu erhöhen und gleichzeitig die Fehleranfälligkeit gegenüber manuell durchgeführten Refactorings zu senken [WB19]. Der am Institut für Software Technologie in Fachbereich Software Engineering der Universität Stuttgart entwickelte Refactoring-Bot[1] ist so eine Methode zum automatischen Durchführen von Refactorings für Java. Er arbeitet auf den Ergebnissen von SonarQube[2], indem er pro Befund in dem betroffenen GitHub Repository einen Pull Request anlegt. In diesem Pull Request hat er den Befund durch das passende Refactoring verbessert. Der Pull Request kann dann von Entwicklern entweder akzeptiert oder abgelehnt werden [WB19]. Da aber in großen Software-Projekten oft hunderte, oft ähnliche Code Smells entdeckt werden und nicht alle geändert werden sollen, da sie von den Entwicklern als irrelevant angesehen werden, ist es mühsam die ganze Liste an vom Bot vorgeschlagenen Refactorings durch zu gehen und zu entscheiden welches Refactoring man annehmen möchte.

1.2. Ziele

In dieser Arbeit soll der Refactoring-Bot erweitert werden, sodass er nicht mehr für jeden Befund einen Pull Request anlegt, sondern verschiedene Befunde sinnvoll gruppiert und

[1] https://github.com/Refactoring-Bot/Refactoring-Bot
[2] https://www.sonarqube.org/

priorisiert. Ziel dieser Arbeit ist es durch das Gruppieren und Priorisieren die Akzeptanz der vom Refactoring-Bot angelegten Pull Request zu erhöhen und das Refactoring für Entwickler leichter zu machen, wodurch die Codequalität von Software-Projekten verbessert wird.

Es wird auch eine Studie durchgeführt, mit der herausgefunden werden soll, ob die User die Implementierung für sinnvoll halten.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2: Beschreibt die Grundlagen, die in dieser Arbeit benötigt werden.

Kapitel 3: Fasst die vorherigen Arbeiten zusammen, auf denen diese Arbeit aufbaut.

Kapitel 4: Die Methodik beschreibt das Vorgehen bei der Implementierung sowie das Studiendesign.

Kapitel 5: Die Ergebnisse der durchgeführten Studie werden beschrieben.

Kapitel 6: Diskutiert und Interpretiert die Ergebnisse der Studie.

Kapitel 7 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2. Grundlagen

In diesem Kapitel stehen die Grundlagen und Definitionen, welche für das Verständnis der Arbeit vorausgesetzt werden.

In Kapitel 2.1 werden einige Grundlagen von Bots vorgestellt und welche Arten von Bots es gibt. In Kapitel 2.2 wird kurz erklärt was Refactoring ist und in Kapitel 2.3 wird erläutert was Code Smells sind.

2.1. Bot

Da in dieser Arbeit der Bot von Wyrich und Bogner erweitert wird, übernehme ich auch ihre Definition von einem Software-Bot: "[...] a bot is intelligent software that acts (to some extent) autonomously to achieve a defined goal and offers functionality for interaction." [WB19](p. 3.).

Es gibt jedoch weitere Merkmale eines Bots die im Folgenden auch noch vorgestellt werden. Lebeuf et al. [LSZ17] stellen einige Arten von Bots vor. Dabei unterscheiden sie die Arten von Bots durch ihr Interaktionsmodell [LSZ17]. Als verschiedene Modelle mit dem Bot zu interagieren, nennen sie die Arten mit speziellen Kommandos in der Kommandozeile und mit natürlicher Sprache durch Sprache oder Schrift [LSZ17]. Als weiteres Modell zur Unterscheidung von Bots nennen Lebeuf et al. die "pull-based" und die "push-based" Ansätze [LSZ17]. Bei den "pull-based" Ansätzen startet der User die Interaktion mit dem Bot [LSZ17]. Wenn der Bot die Interaktion mit dem User startet, nennt man das einen "push-based" Ansatz [LSZ17]. Weitere Arten wie Lebeuf et al. [LSZ17] Bots charakterisieren, sind über ihre Intelligenz:

- Anpassung, wenn der Bot Kontextabhängig mit dem User interagiert.
- Argumentation, einfache logische Regeln vs. fortgeschrittene KI.
- Autonomie, komplett autonom, menschlicher Trigger, oder eine Mischung aus beidem.

Die letzte Art wie Lebeuf et al. [LSZ17] Bots charakterisieren, ist über ihren Zweck:

- General/Allgemeine-Bots: sie unterstützen einige einfache Aufgaben.
- Transaktion-Bots: führen automatisch Transaktionen mit externen Systemen aus, dies geschieht alles im Namen des Users.

- Information-Bots, bieten den Usern Informationen an.
- Produktivität-Bots, verbessern die Produktivität der User.
- Zusammenarbeit-Bots, helfen den Usern bei der Kommunikation, Koordination und der Zusammenarbeit.

Weiter beschreiben Lebeuf et al. Methoden, um Bots zu erstellen und zu hosten [LSZ17]. Dabei gehen sie auch auf die drei Gesetze für Robotik von Isaac Asimov [Asi42] ein: "Isaac Asimov's three laws of robotics state that robots must not harm humans, must obey orders, and must protect themselves." [LSZ17](p. 22) und sagen, dass diese Regeln auch für Software-Bots angewandt werden können.

Der Refactoring-Bot wäre, nach den Kategorien von Lebeuf et al., ein Bot der auf dem "pull-based" Ansatz basiert, da man ihm erst mitteilen muss welches Repository er analysieren soll, hier startet also der User die Interaktion mit dem Bot. Außerdem soll er die Produktivität von Entwicklerteams steigern, einfache Aufgaben (das Refactoring) übernehmen und den Entwicklern Informationen anbieten (die Durchgeführten Refactorings, die mit den Pull Requests dargestellt werden).

2.2. Refactoring

Refactoring ist nach Martin Fowler "[...] the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."[Fow18](p. 9). Refactoring verbessert also Code-Stellen, um die Chancen auf Bugs zu minimieren [Fow18]. Martin Fowler [Fow18] stellt das Refactoring vor und beschreibt, welche Möglichkeiten es gibt um Code Smells zu beheben. Er stellt einige Arten des Refactorings vor, die der Refactoring-Bot verwendet. Für diese Arbeit ist es wichtig zu verstehen wie die jeweiligen Refactorings funktionieren, damit man entscheiden kann wie man sie Gruppiert.

2.3. Code Smell

Beck und Fowler [BFB99] beschreiben einen Code Smell als eine Code-Stelle, welche nach Refactoringmaßnahmen verlangen. In ihrem Paper Bad Smells in Code [BFB99] geben Beck und Fowler dann einige Merkmale wie duplizierter Code, lange Methoden etc. an, bei denen man Code Smells identifizieren kann.

3. Verwandte Arbeiten

Wyrich und Bogner [WB19] präsentieren in ihrem Paper den Refactoring-Bot, der in dieser Arbeit erweitert wird. Sie beschreiben, dass ein automatisches Refactoring im Gegensatz zu manuellem Refactoring effektiver und weniger fehleranfälliger ist [WB19]. Ziel des Refactoring-Bots ist es die Codequalität mit minimalen Bemühungen der Entwickler zu verbessern [WB19]. Dabei handelt der Refactoring-Bot autonom und verhält sich wie andere Entwickler, um sich in das Entwicklerteam zu integrieren, dies gelingt ihm, indem er für die Änderungen die er an dem Code vornimmt Pull Requests anlegt um den anderen Entwicklern mitzuteilen dass er Änderungen durchgeführt hat [WB19]. Entwickler können dann über Kommentare der Pull Requests mit dem Refactoring-Bot kommunizieren [WB19]. Ein weiterer Vorteil der Refactoring-Bots ist es, dass er nur auf der Syntax des Programms arbeitet [WB19]. Das bedeutet, dass er nicht durch komplexe Codestücke verwirrt werden kann, während Menschen oft die Semantik des Codes verstehen müssen, um Änderungen vornehmen zu können [WB19]. Um den Refactoring-Bot anwenden zu können, benötigt man ein GitHub Repository mit Java Code, welches der Refactoring-Bot automatisch forkt, um damit zu arbeiten, einen Projekt-Schlüssel und einen Host für eine SonarQube Instanz des Projektes [WB19]. Der Refactoring-Bot arbeitet dann mit den Ergebnissen von SonarQube [WB19]. Zum Schluss werden, die erst noch lokal durchgeführten Refactorings dann durch Pull Requests in das originale Repository übertragen, wobei die Entwickler in der erstellten Konfiguration angeben können, wieviele Pull Request der Refactoring-Bot maximal erstellen soll [WB19]. Der Refactoring-Bot von Wyrich und Bogner wird in dieser Arbeit erweitert.

Vidal et al. [VMD16] gehen auf drei verschiedene Methoden ein, um Code Smells zu priorisieren. Sie priorisieren die Code Smells nach den Kriterien: Stabilität der Komponente, also wie oft sie verändert wurden, der von Usern festgelegten Relevanz für Arten von Code Smells und nach Auswirkungen der Code Smells auf wichtige Modifikationsszenarien [VMD16]. Sie präsentieren den semi-automatischen SpIRIT (Smart Identificatio of Refactoring opportunITies) Ansatz zum priorisieren der Code Smells [VMD16]. SpIRIT analysiert Komponenten des Softwaresystems auf Code Smells und priorisiert diese auf ihrer Wichtigkeit basierend auf den oben genannten Kriterien [VMD16]. Nach der Implementierung von SpIRIT haben Vidal et al. zwei Fallstudien durchgeführt, um zu testen, ob ihre Priorisierungsansätze sinnvoll sind. Die Fallstudien haben die Vermutung von Vidal et al. bestätigt, dass der SpIRIT-Ansatz für die Priorisierung von Code Smells von Vorteil für Entwickler ist [VMD16]. In dieser Arbeit soll für den Refactoring-Bot eine Priorisierung der Code Smells implementiert werden, daher sind einige der Kriterien von Vidal et al. für diese Arbeit von Interesse.

Fontana et al. [FFZR15] stellen den *Intensity Index* vor, der beschreibt wie kritisch die gefundenen Code Smells sind. Dabei beschreiben sie zwei Schritte zum Priorisieren von Code Smells 1) die echten Code Smells ausfiltern, da die Software zum Entdecken von Code Smells auch falsch positive Befunde entdecken kann und 2) die kritischsten Code Smells identifizieren, wobei sie sich auf den zweiten Teil fokussieren [FFZR15]. Um zu entscheiden welche Code Smells kritisch sind, erstellen sie einen Index, sie nennen ihn *Intensity Index*, welcher angibt wie kritisch ein Code Smell Befund verglichen zu den anderen Code Smell Befunden im selben Code ist [FFZR15]. Der Index von Fontana et al. geht von 1-10, wobei 10 der kritischste Wert ist [FFZR15]. Ihren Index evaluieren Fontana et al. dann durch ein Experiment, indem sie ihn auf sechs Code Smells (*GodClass*, *DataClass*, *BrainMethod*, *ShotgunSurgery*, *DispersedCoupling* und *MessageChain*) in 74 Systemen anwenden [FFZR15]. Sie haben dabei herausgefunden, dass für die Code Smells kaum ein kritischer Wert *Very Low* annimmt, während die meisten Code Smells einen kritischen Wert von *High* bekamen [FFZR15]. Diesen Index verwenden sie dann, um die Code Smells zu priorisieren, damit der Entwickler schneller die Code Smells beheben kann, die einen kritischen Wert von *Very High* oder *High* haben, da sie dafür nur wenig Zeit haben [FFZR15]. Für diese Arbeite ist die Art, wie Fontana et al. die Code Smells filtern für die Priorisierung der Code Smells von Bedeutung.

Die Suche nach Literatur, die sich mit dem Gruppieren von Code Smells bzw. mit der Gruppierung von Refactorings beschäftigt, gestaltete sich sehr schwer. Da ich keine Veröffentlichungen gefunden habe, die sich vollständig mit dem Thema befassen, muss ich Daten aus anderer Literatur, die sich mit Code Smells und Refactoring befassen, verwenden.

Sjøberg et al. [SYA+12] befassen sich mit dem Aufwand den Code Smells für die Wartung von Software haben können. Dabei führen sie eine Studie mit sechs Softwareentwicklern durch. Jeder der Entwickler muss drei Wartungsaufgaben in zwei von vier äquivalenten aber unabhängigen, in Java implementierten, Systemen durchführen [SYA+12]. Bei ihrer Studie sind sie zu dem Schluss gekommen, dass der Aufwand Code mit Code Smells zu ändern nicht signifikant höher ist, als der Aufwand Code ohne Code Smells zu ändern [SYA+12]. Sjøberg et al. [SYA+12] und Fowler [Fow18] nennen viele Code Smells wie *DataClass*, *GodClass* etc. und einige Refactorings, die innerhalb einer Klasse stattfinden wie z. B. *ExtractMethod*. Daraus schließe ich, dass es in manchen Fällen sinnvoll ist Refactorings zu gruppieren, die in der gleichen Klasse angewandt werden, da sie nur Auswirkungen auf die innere Struktur der Klasse haben. Ein Beispiel für Refactorings, die der Refactoring-Bot unterstützt und nur die innere Struktur einer Klasse betreffen, ohne Auswirkungen außerhalb der Klasse zu haben, wäre *ReorderModifier* einer privaten Methode. Die Aufteilung der Refactorings in verschiedene Klassengruppen wird ein Teil des hier implementierten Gruppierungsalgorithmus sein.

Murphy-Hill und Black stellen den Code Smell-Detektor Stench Blossom vor, mit dem Entwickler einen schnellen Überblick über die Code Smells in ihrem Code bekommen. [MB10]. In Stench Blossom kommt die Ausgabe als gemeinsame Ausgabe verschiedener Code Smell Analysatoren zustande, wobei jeder Analysator eine skalare Metrik in einem bekannten Bereich berechnet [MB10]. Außerdem haben sie, durch ihre Arbeit an Stench Blossom, Guidelines entwickelt von denen sie denken, dass sie für die Entwicklung eines Code Smell-Detektors

nützlich sind [MB10]. Murphy-Hill und Black führen dann noch ein Experiment durch, indem sie Programmierer verschiedene Code Smells ohne und mit dem Stench Blossom analysieren lassen und sie Refactoring Urteile fällen lassen [MB10]. In diesem Experiment haben Murphy-Hill und Black herausgefunden, dass die Programmierer mit dem Tool mehr Code Smells identifizieren konnten als ohne Stench Blossom [MB10]. Außerdem haben sie mit dem Experiment bestätigt, dass Code Smells subjektiv sind [MB10], als Beispiel geben sie hier an, dass verschiedene Probanden eine unterschiedliche Auffassung davon haben, wann eine Methode oder Klasse zu groß ist [MB10]. Außerdem bestätigen sie durch ihr Experiment noch ihre weiteren zwei Hypothesen, dass Programmierer mit dem Tool sicherere und begründetere Refactoringentscheidungen treffen als ohne das Tool, und dass ihre Guidelines wünschenswerte Designüberlegungen für den Entwurf von Code Smell-Detektoren sind [MB10]. Murphy-Hill und Black [MB10] beschreiben in ihrem Paper unter anderem auch Schwierigkeiten beim Refactoring des Code Smell *DataClumps*, da beim Ändern des Codes eines Objekts alle Stellen, an dem das Objekt verwendet wird, auf Änderungen untersucht werden müssen [MB10]. Daher kann man hieraus entnehmen, dass es sinnvoll ist Refactorings zu Gruppieren, die sich mit dem selben Objekt befassen.

4. Methodik

In diesem Kapitel wird darauf eingegangen, wie die Implementierung der Gruppierung und der Priorisierung vorgenommen wurde und wie die Studie, mit der einige Forschungsfragen zu der Implementierung beantwortet werden soll (siehe Kapitel 4.2), gestaltet wurde.

Dabei beschreibt Kapitel 4.1 wie ich meine Lösung realisiert habe. Kapitel 4.1 ist in weitere Teile gekapselt, die sich jeweils mit einer Aufgabe der Implementierung befassen. Hierbei befasst sich Kapitel 4.1.2 mit der Gruppierung und Kapitel 4.1.3 mit der Priorisierung der Pull Requests.

In Kapitel 4.2 wird dann beschrieben was die Forschungsfragen sind und in den Kapiteln 4.2.1 und Kapitel 4.2.2 wird beschrieben wie das Experiment aufgebaut ist und wie die Daten erhoben werden, welche Daten erhoben werden und wie damit die Forschungsfragen beantwortet werden sollen.

4.1. Implementierung

Bei der Erweiterung des Refactoring-Bots gibt es zwei Aufgaben die separat implementiert werden, zum einen die Gruppierung der Code Smells (Kapitel 4.1.2) und zum anderen die Priorisierung der Code Smells (Kapitel 4.1.3). Der Aspekt, dass sich die Priorisierung auch auf die Gruppierung auswirken kann, ist hier in so fern richtig, dass die Code Smells bevor sie gruppiert werden bereits priorisiert wurden. Die Priorisierung wirkt sich also so auf die Gruppierung aus, dass die ersten Gruppen die erstellt werden die Code Smells enthalten, die für diese Art von Gruppe die höchste Priorität haben. Ich habe mich dazu entschieden die Code Smells erst zu priorisieren, dann zu gruppieren und zuletzt die Gruppen zu priorisieren. Dies hat den Vorteil, dass die resultierenden Gruppen von Refactorings jene Refactorings enthalten, die für ihre Gruppenart, siehe Kapitel 4.1.2, die höchste Priorität besitzen. Die Priorisierung der Gruppen ist dann noch nötig, damit z. B. eine Gruppe, die am Anfang angelegt hat und fünf der wichtigsten Refactorings enthält, nicht von einer Gruppe die zuletzt angelegt wurde und 20 der am niedrigsten priorisierten Refactorings enthält, überholt wird.

4.1.1. Gesamter Algorithmus

Der komplette Algorithmus, zu sehen in Abbildung 4.1, der in dieser Abschlussarbeit implementiert wird, besteht aus drei Phasen. Bevor der Algorithmus mit der Priorisierung der Code Smells beginnt, wird die Liste an Code Smells bereinigt. Das heißt, dass die Code Smells, die der Refactoring-Bot schon einmal gerefactored hat und von den Entwicklern jedoch nicht oder noch nicht angenommen wurden aus der Liste entfernt werden, um unnötige Duplikate zu verhindern. Diese Funktion ist schon in dem Refactoring-Bot implementiert und wurde hier zum besseren Verständnis erwähnt.

In der ersten Phase priorisiert der Refactoring-Bot die Code Smell-Befunde von *SonarQube* wie in Kapitel 4.1.3 beschrieben. Anschließend werden die Code Smells nach den in Kapitel 4.1.2 genannten Kriterien in Gruppen gegliedert. Bevor der Refactoring-Bot dann für jede Gruppe die nötigen Refactorings durchführt und einen Pull Request erstellt, werden die Gruppen noch priorisiert, wie in Kapitel 4.1.3 erläutert. Nach der Gruppierung werden für jede Gruppe die passenden Refactorings durchgeführt und einzeln commited. Der fertige Pull Request besteht dann aus den durchgeführten Refactorings für die betrachtete Code Smell Gruppe. Das pushen der Pull Requests beginnt dann mit der am höchsten priorisierten Gruppe und erstellt im Anschluss jeweils für die nächst niedriger priorisierten Gruppen die nächsten Pull Requests. Die Pull Requests werden so lange erzeugt, bis der Refactoring-Bot sein Limit für die Pull Requests erreicht hat. Das Verhalten, dass der Bot nur bis zu einem bestimmten Limit Pull Requests erzeugt, wurde bereits vor dieser Arbeit implementiert und ist in Abbildung 4.1 nicht dargestellt. Somit stehen, nach der Anwendung des Refactoring-Bots, die am höchsten priorisierte Gruppen in den Pull Requests auf GitHub. Da, die am höchsten priorisierte Gruppe zuerst gepusht wurde, steht sie jetzt in GitHub ganz hinten in der Reihe von Pull Requests, die von dem Refactoring-Bot erzeugt wurden, dies kann man beheben, indem man in GitHub die Pull Requests mithilfe der Sortierfunktion so sortiert, dass der Pull Request, der als erstes erzeugt wurde oben steht.

4.1.2. Gruppierung

Bevor der Refactoring-Bot Refactorings, für die gefundenen Code Smells anwendet, sollen diese erst Gruppiert werden, damit man später ganze Gruppen von Refactorings in einem Pull Request hat, da zu viele kleine Pull Requests schwer zu Überblicken sind wie Wyrich und Bogener [WB19] erwähnen. Wyrich und Bogener erwähnen jedoch auch, dass zu große Pull Requests ebenfalls schwer zu reviewen sind [WB19], deshalb befasst sich dieses Kapitel auch mit der passenden Gruppengröße. Die Gruppierung der Code Smells erfolgt durch die folgenden Eigenschaften:

- Code Smells gleicher Art.
- Code Smells in der selben Klasse.

Wyrich und Bogener vermuten in ihrem Paper dass es sinnvoll sein kann Refactorings vom selben Typen zu Gruppieren [WB19]. Daher wird die erste Art der Gruppenbildung durch die Art der Refactorings bestimmt. Hierbei handelt es sich um Refactorings, die nur eine oder wenige aufeinanderfolgende Code-Zeilen betreffen. Für den Refactoring-Bot wären das zur Zeit die *AddOverrideAnnotation* und die *RemoveCommentedOutCode* Refactorings. Es ist aber nicht bei jeder Art von Refactoring sinnvoll sie so zu Gruppieren, zum Beispiel kann es sinnvoll sein alle *RemoveCommentedOutCode* Refactorings zu gruppieren, da sie die Struktur des Codes nur an der Stelle betrifft, an der der Kommentar steht und ein entfernter Kommentar keine Auswirkungen auf andere Stellen in der Datei oder sogar andere Dateien hat. Jedoch macht es keinen Sinn alle *RemoveMethodParameter* Refactorings zu gruppieren, da sich hier der Code unter Umständen an vielen Stellen innerhalb oder außerhalb der Klasse ändert [Fow18]. Dies wird schnell unübersichtlich.

Die zweite Gruppenart von Refactorings bilden die Refactorings, die sich innerhalb einer Klasse befinden. Diese Entscheidung kommt daher, da es viele Refactorings gibt, die sich nur auf die innere Struktur einer Klasse auswirken. Sjøberg et al. [SYA+12] und Fowler [Fow18] nennen hier solche Refactorings wie *ExtractMethod*.

Es würde auch Sinn machen Refactorings, die sich über die gleichen Klassen ausdehnen, wie die Änderung eines Methodenkopfes, der in anderen Klassen aufgerufen wird, gemeinsam zu Gruppieren, wie Murphy-Hill und Black [MB10] erwähnen, können sich Änderungen, wie Refactorings, an Objekten über mehrere Dateien ausdehnen. Jedoch erfährt man erst welche Dateien durch das Refactoring geändert wurden, wenn der Code Smell bereits gerefactored wurde und für diesen Schritt werden die Gruppen an Code Smells schon benötigt. Daher wird diese Art von Code Smell in der Klassengruppe gespeichert, in der der Methodenkopf geändert wird.

Der Refactoring-Bot unterstützt derzeit, Stand: 22.02.2020, 5 Refactorings:

- *AddOverrideAnnotation*,
- *RemoveCommentedOutCode*,
- *RemoveMethodParameter*,
- *RenameMethod* und
- *ReorderModifier*.

Diese werde ich wie folgt Gruppieren. Die *AddOverrideAnnotation* und die *RemoveCommentedOutCode* werden jeweils in der ersten Art der Gruppierung gruppiert, da sie keine großen strukturelle Auswirkungen auf Klassen oder Objekte haben, außer dass Zeilen gelöscht oder hinzugefügt werden. Damit gibt es jeweils Gruppen die nur *AddOverrideAnnotation* Refactorings beinhalten und Gruppen die nur *RemoveCommentedOutCode* Refactorings beinhalten.

Bei den Refactorings *RemoveMethodParameter*, *RenameMethod* und *ReorderModifier* muss zuerst überprüft werden, ob sie zur selben Klasse gehören. Wenn sie zur selben Klasse gehören, dann werden sie in einer Gruppe gespeichert, welche für diese Klasse zuständig ist, und für diese

Gruppe wird ein Pull Request angelegt. Jede Klasse hat also ihre eigene Gruppe, oder mehrere Gruppen falls eine Gruppe an Refactorings zu groß wird.

Bei der Gruppierung muss man auch darauf achten, dass die Gruppen nicht zu groß werden, denn große Pull Requests sind genau so schwer zu Handhaben wie viele kleine Pull Requests wie Wyrich und Bogener [WB19] erwähnen. Daher müssen große Gruppen von Refactorings sinnvoll aufgeteilt werden.

Gousios et al. [GPD14] geben in ihrer Studie an, dass ein Pull Request im Allgemeinen weniger als 10 commits beinhaltet. Außerdem geben sie an, dass ein Pull Request meistens weniger als 20 geänderte Dateien enthält und die Anzahl der geänderten (Code-)Zeilen im Durchschnitt weniger als 500 pro Pull Request beträgt, mit einem Median von 20 Zeilen pro Pull Request [GPD14].

Da das Refactoring *AddOverrideAnnotation* in einer eigenen Gruppe gespeichert wird und jeweils nur eine Zeile im Code ändert, werden die Gruppen mit den *AddOverrideAnnotation* Refactorings auf 20 Code Smells pro Gruppe beschränkt, da nach Gousios et al. [GPD14] der Median der geänderten (Code-)Zeilen pro Pull Request bei 20 liegt. Bei den *RemoveCommentedOutCode* Refactorings wird die Anzahl an Befunden für eine Gruppe auf 10 gesetzt, da der auskommentierte Code nicht immer auf eine Zeile beschränkt ist, sondern auch mehrere Zeilen beinhalten kann. Da es durch kleinere Bugs, die das *RemoveCommentedOutCode* Refactoring des Refactoring-Bots betreffen, sehr schwer ist die Anzahl an gelöschten Kommentarzeilen zu zählen wird hier anstatt der Begrenzung auf 500 geänderte Code-Zeilen pro Pull Requests ein Limit von 10 Commits, also 10 *RemoveCommentedOutCode* Refactorings, pro Pull Request verwendet, da nach Gousios et al. Pull Requests generell weniger als 10 Commits beinhalten [GPD14].

Für die Refactoring-Gruppen welche die Klassengruppen betreffen, habe ich eine Grenze von 20 Code Smells pro Gruppe aufgestellt, da bei den, bis jetzt unterstützten Refactorings nur wenige Code-Zeilen pro Refactoring geändert werden. Diese Grenze kommt zustande, da nach Gousios et al. [GPD14] der Median für geänderte Code-Zeilen bei 20 liegt.

Beispiel: Der Algorithmus für die Gruppierung bekommt als Input eine nach Kapitel 4.1.3 priorisierte Liste an Code Smells, wobei der am höchsten priorisierte Befund am Anfang der Liste, hier im Beispiel ganz oben in der Tabelle steht, siehe Tabelle 4.1.

Der Gruppierungsalgorithmus geht die Liste jetzt der Reihe nach durch und überprüft zuerst, ob er den Listeneintrag in eine Gruppe von Refactorings gleicher Art oder in eine Klassengruppe einfügen soll. Im nächsten Schritt überprüft er dann, ob so eine Gruppe, für diese Klasse oder diese Art von Refactoring bereits existiert und ob noch genügend Platz verfügbar ist. Wenn so eine Gruppe bereits existiert und noch nicht voll ist, dann fügt er den Befund in die vorhandene Gruppe ein. Sollte keine entsprechende Gruppe existieren oder die existierenden Gruppen schon voll sein, dann legt er eine neue passende Gruppe an.

Nach dem Durchlauf des Algorithmus gibt es dann eine Liste von Gruppen, die in der Tabelle 4.2 dargestellt wird, wobei die Gruppen durch doppelte Linien voneinander getrennt sind.

Code Smell	Klasse	Anzahl Änderungen
RemoveMethodParameter	A	16
RemoveMethodParameter	B	15
AddOverrideAnnotation	B	12
ReorderModifier	A	8
RenameMethod	B	8
AddOverrideAnnotation	A	1
AddOverrideAnnotation	C	1
RemoveCommentedOutCode	B	3
RemoveCommentedOutCode	A	5
RemoveCommentedOutCode	C	7

Tabelle 4.1.: Liste der priorisierten *BotIssues*.

Code Smell	Klasse	Anzahl Änderungen
RemoveMethodParameter	A	16
ReorderModifier	A	8
RemoveMethodParameter	B	15
RenameMethod	B	8
AddOverrideAnnotation	B	12
AddOverrideAnnotation	A	1
AddOverrideAnnotation	C	1
RemoveCommentedOutCode	B	3
RemoveCommentedOutCode	A	5
RemoveCommentedOutCode	C	7

Tabelle 4.2.: Liste von den *BotIssueGroups*, wie sie nach der Gruppierung aussehen.

Die einzelnen Schritte des Gruppierungsalgorithmus können in Abbildung 4.2 verfolgt werden.

4.1.3. Priorisierung

Da es in großen Software-Projekten trotz der Gruppierung von Code Smells immer noch zu sehr vielen unübersichtlichen Pull Requests für die vom Refactoring-Bot durchgeführte Refactorings geben kann und die Entwickler weder Lust noch Zeit haben alle Pull Requests auf ihre Relevanz hin zu untersuchen, benötigt es eine sinnvolle Methode um wichtige Refactorings hervorzuheben. Daher ist eine Priorisierung der Code Smells und später der Refactoring-Gruppen sinnvoll, damit man wichtige Refactorings die unbedingt vorgenommen werden

sollen weit oben in den Pull Requests stehen hat und nicht so wichtige Refactorings weiter unten in den Pull Requests. Für die Priorisierung gibt es zwei Algorithmen, zum Einen, um die Code Smells zu priorisieren und zum Anderen, um die Gruppen an Refactorings zu priorisieren. Im ersten Teil dieses Kapitels gehe ich darauf ein wie die Code Smells priorisiert werden und im zweiten Teil werde ich beschreiben wie die Gruppen kurz vor dem Anlegen des Pull Requests priorisiert werden.

Priorisierung der Code Smells

Viele der Refactorings sind nicht so wichtig wie andere, wie Vidal et al. zum Beispiel beschreiben ist ein Code Smell in einer Klasse, die seit ihrer Erzeugung nicht mehr geändert wurde nicht so wichtig wie ein Code Smell der in einer Klasse liegt, die ständig geändert wird [VMD16]. Daher muss man eine sinnvolles Ranking für die Code Smells finden und implementieren. Nicht nur Vidal et al., sondern auch Wyrich und Bogener unterstützen es Code Smells höher zu priorisieren, wenn sie in einer Code Stelle stehen, die oft bearbeitet wurde [VMD16; WB19], da hier die Wahrscheinlichkeit höher ist, dass diese Code-Stellen in der Zukunft noch einmal bearbeitet werden [VMD16]. Vidal et al. führen noch zwei weitere Faktoren ein um Code Smells zu priorisieren, erstens den Einfluss eines Code Smells auf Modifizierbarkeitsszenarien und zweitens die Relevanz des Typs eines Code Smells [VMD16].

In dieser Arbeit werden die Code Smells so priorisiert, dass diejenigen Code Smells, die in Code Stellen stehen, die oft geändert wurden eine höhere Priorität bekommen, nach der Begründung von Vidal et al. [VMD16]. Sollten zwei Code Stellen die gleiche Anzahl an Änderungen haben, dann entscheidet das letzte Änderungsdatum darüber, welcher Code Smell die höhere Priorität bekommt. Wie in Abbildung 4.1 zu sehen ist, gibt es zwei Priorisierungsphasen für die Code Smells. Die erste Priorisierung findet statt, wenn die Code Smells noch durch *SonarIssue* repräsentiert werden. In diesem Schritt werden die Code Smells nach ihrem Datum priorisiert, so dass der Code Smell mit dem neuesten Datum an erster Stelle in der Liste steht, der Code für diese Funktion wird in Listing 4.2 gezeigt. Dies geschieht vor dem Übersetzen in *BotIssues*, da die *SonarIssue* noch das letzte Änderungsdatum speichern, die *BotIssues* aber nicht mehr. Die zweite Priorisierung der Code Smells findet nach dem Übersetzen der Code Smells von *SonarIssue* in *BotIssues* statt. Hier werden die *BotIssues* dann mit dem stabilen Sortierverfahren *BubbleSort* nach der Anzahl ihrer Änderungen priorisiert, sodass der Code Smell mit den meisten Änderungen die höchste Priorität hat. Die Anzahl an Änderungen einer Code-Stelle ist hierbei die Anzahl an Commits, die man mit GitHub herausfinden kann, zu sehen in Listing 4.3. Dabei wird die Anzahl an Änderungen eines Code Smells in dem vom Refactoring-Bot erzeugten Objekt vom *BotIssue*, welches zu dem jeweiligen Code Smell gehört, gespeichert. *BubbleSort* wurde deshalb ausgewählt, weil es ein stabiles Sortierverfahren ist, dass bedeutet, dass die Reihenfolge der *BotIssues* in der vorsortierten Liste bei gleicher Anzahl an Änderungen bestehen bleibt und somit die *BotIssues* mit einem neueren Änderungsdatum aber gleicher Anzahl an Änderungen die höhere Priorität bekommen. So hat zum Beispiel ein Code Smell, dessen Code-Stelle 5-mal bearbeitet wurde und zuletzt am 20.12.2019 um 17:19 Uhr bearbeitet

Listing 4.1 Hilfsmethode, um die *SonarIssues* nach ihrem Datum zu sortieren. Diese Funktion wurde aus Stackoverflow übernommen. Link: https://stackoverflow.com/questions/109383/sort-a-mapkey-value-by-values

```
private static <K, V extends Comparable<? super V>> Map<K, V> sortByValue(Map<K, V> map) {
                        List<Map.Entry<K, V>> list = new ArrayList<>(map.entrySet());
                        list.sort(Map.Entry.comparingByValue());

                        Map<K, V> result = new LinkedHashMap<>();
                        for (Map.Entry<K, V> entry : list) {
                                    result.put(entry.getKey(), entry.getValue());
                        }

                        return result;
            }
```

wurde, eine höhere Priorität als ein Code Smell, dessen Code-Stelle ebenfalls 5-mal bearbeitet wurde und zuletzt am 20.12.2019 um 17:09 Uhr bearbeitet wurde. Der Algorithmus für die Code Smell Priorisierung ist in Abbildung 4.3 dargestellt.

Für die Priorisierung der *SonarIssues* nach dem Datum wurde eine kleine Hilfsfunktion[1] aus Stackoverflow übernommen, zu sehen in Listing 4.1.

Da es mit den *RemoveCommentedOutCode* Refactorings Probleme gab, wenn mehrere Zeilen Kommentare entfernt werden stimmt der Code nicht mehr mit der SonarQube Analyse überein, werden die *RemoveCommentedOutCode* Refactorings vor dem Priorisieren nach der Anzahl der Änderungen aus der Liste entfernt, separat priorisiert und zum Schluss wieder an die ursprüngliche, aber jetzt priorisierte, Liste von Code Smells angefügt. Diese leichte Verfälschung der Priorisierung von den Code Smells muss in Kauf genommen werden, damit der Algorithmus funktionieren kann. Die *RemoveCommentedOutCode* Code Smells werden jetzt nach ihrer Zeilennummer priorisiert, wobei eine größere Code-Zeilennummer eine höhere Priorität bedeutet. Durch dieses Priorisierungsverfahren ist bei den *RemoveCommentedOutCode* Refactorings jedoch nicht mehr sichergestellt, dass *RemoveCommentedOutCode* Code Smells die sich in derselben Klasse befinden auch später in derselben Gruppe landen.

Beispiel: Als Input bekommt dieser Algorithmus eine Liste an *SonarIssues* von SonarQube, zu sehen in Tabelle 4.3.

Diese Liste wird dann, bevor die SonarQubeIssues in die BotIssues übersetzt werden nach dem Datum sortiert, so dass der Eintrag mit dem neuesten Datum die höchste Priorität hat. Das Ergebnis dieses Schrittes ist in Tabelle 4.4 zu sehen.

Nach diesem Schritt werden die *SonarIssues* dann in *BotIssues* übersetzt. Bevor die *BotIssues* dann nach ihrer Anzahl an Änderungen sortiert werden, werden die *RemoveCommentedOutCode* Code Smells in einer separaten Liste gespeichert, Tabelle 4.5.

[1] https://stackoverflow.com/questions/109383/sort-a-mapkey-value-by-values

Listing 4.2 Methode, um die *SonarIssues* nach ihrem Datum zu sortieren.

```
private List<SonarIssue> dateSort(SonarQubeIssues sqIssues){
                        Date creationDate;
                        DateFormat format = new
                            SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss'+'SSSS");
                        //in this List the current findings are saved
                        List<SonarIssue> sonarIssues;
                        //In this map the ID of the finding and its date are saved
                        Map<String, Date> sonarIssueMap = new HashMap<>();
                        //In this list the sorted SonarIssues are saved
                        List<SonarIssue> sortedIssues;

                        //the current findings are fetched
                        sonarIssues = sqIssues.getIssues();

                        //this loop runs throug all findings of the project
                        for (SonarIssue sonarIssue : sonarIssues){
                                    try {
                                                //for each finding a date object is
                                                    created with the date from the issue
                                                creationDate =
                                                    format.parse(sonarIssue.getCreationDate());
                                                sonarIssueMap.put(sonarIssue.getKey(),
                                                    creationDate);

                                    } catch (ParseException e) {
                                                e.printStackTrace();
                                    }
                        }
                        sonarIssueMap = sortByValue(sonarIssueMap);
                        //new list for each project
                        sortedIssues = new ArrayList<>();

                        //loop to bring the SonarIssues list in the correct order
                        for (Map.Entry<String, Date> entry : sonarIssueMap.entrySet() ){
                                    for (SonarIssue issue : sonarIssues){
                                                if
                                                    (entry.getKey().equals(issue.getKey())){
                                                            sortedIssues.add(issue);
                                                }
                                    }
                        }
                        Collections.reverse(sortedIssues);

                  return sortedIssues;
            }
```

Listing 4.3 Methode, um die Commits einer Klasse zu zählen, damit man die Anzahl an Änderungen ermitteln kann.

```
public int countCommitsFromHistory(BotIssue issue, GitConfiguration gitConfig, String
    branch){
                int count = 0;
                Git git;
                Iterable<RevCommit> commits;
                String path = issue.getFilePath();
                path = path.replaceAll("\\\\", "/");
                Repository repository;

                try {
                            repository = new
                                FileRepository(botConfig.getBotRefactoringDirectory() +
                                gitConfig.getConfigurationId() + "/.git");
                            ObjectId objID = repository.resolve(Constants.HEAD);
                            //switchBranch(gitConfig, branch);
                      git = Git.open(new File(botConfig.getBotRefactoringDirectory() +
                          gitConfig.getConfigurationId() /*+ path*/));
                      commits = git.log().add(objID).addPath(path).call();

                      for (RevCommit commit : commits){
                                  count++;
                      }
                }catch (Exception e){
                            logger.error(e.getMessage(), e);
                } finally {
                            return count;
                }
     }
```

Im Anschluss wird die Liste mit den übrigen Code Smells mit dem stabilen Sortierverfahren *BubbleSort* nach der Anzahl an Änderungen sortiert, so dass der Eintrag mit den meisten Änderungen die höchste Priorität bekommt und Einträge, die dieselbe Anzahl an Änderungen haben nach ihrem Änderungsdatum priorisiert bleiben. Die Liste mit den *RemoveCommentedOutCode* Code Smells wird nach der Zeilennummer sortiert, sodass die größte Zeilennummer die höchste Priorität bekommt. Tabelle 4.6 gibt den Zustand der zwei Listen nach diesen zwei Schritten wieder.

Am Ende der Priorisierung wird die Liste der *RemoveCommentedOutCode* Code Smells hinten an die Liste der restlichen Code Smells angefügt. In Tabelle 4.7 steht die Liste, wie sie nach dem Code Smell Priorisierungsalgorithmus aussieht.

Diese Liste, Tabelle 4.7 bekommt dann, der in Kapitel 4.1.2 beschriebene Gruppierungsalgorithmus als Input übergeben.

Code Smell	Klasse	Anzahl Änderungen	Letzte Änderung
AddOverrideAnnotation	C	1	21.07.2015
RemoveMethodParameter	B	15	26.07.2018
RenameMethod	B	8	15.03.2017
AddOverrideAnnotation	B	12	28.01.2020
RemoveMethodParameter	A	16	17.01.2019
AddOverrideAnnotation	A	1	21.07.2019
RemoveCommentedOutCode	A	5	01.01.2016
RemoveCommentedOutCode	B	3	03.05.2019
RemoveCommentedOutCode	C	7	20.07.2019
ReorderModifier	A	8	26.07.2018

Tabelle 4.3.: Liste von *SonarIssues*.

Code Smell	Klasse	Anzahl Änderungen	Letzte Änderung
AddOverrideAnnotation	B	12	28.01.2020
AddOverrideAnnotation	A	1	21.07.2019
RemoveCommentedOutCode	C	7	20.07.2019
RemoveCommentedOutCode	B	3	03.05.2019
RemoveMethodParameter	A	16	17.01.2019
RemoveMethodParameter	B	15	26.07.2018
ReorderModifier	A	8	26.07.2018
RenameMethod	B	8	15.03.2017
RemoveCommentedOutCode	A	5	01.01.2016
AddOverrideAnnotation	C	1	21.07.2015

Tabelle 4.4.: Liste, der nach dem Datum sortierten *SonarIssues*.

Priorisierung der Gruppen

Die Priorisierung der Gruppen wird durch die Summe der Prioritätseigenschaften der in ihr enthaltenen Refactorings bestimmt, also durch die Summe der Anzahl an Änderungen, der in ihnen enthaltenen Code Smells. Abbildung 4.4 gibt einen kurzen Überblick über die Priorisierung der Gruppen.

Beispiel: Der Algorithmus summiert für jede Gruppe die Anzahl an Änderungen, der in ihr enthaltenen *BotIssues* auf, Tabelle 4.8.

Anschließend werden die Gruppen mit diesem Wert priorisiert, so dass die Gruppe, welche die Meisten Anzahl an Änderungen hat, die höchste Priorität bekommt und somit an erster Stelle in der Liste steht, Tabelle 4.9.

Code Smell	Klasse	Anzahl Änderungen	Letzte Änderung	Zeilen Nummer
AddOverrideAnnotation	B	12	28.01.2020	
AddOverrideAnnotation	A	1	21.07.2019	
RemoveMethodParameter	A	16	17.01.2019	
RemoveMethodParameter	B	15	26.07.2018	
ReorderModifier	A	8	26.07.2018	
RenameMethod	B	8	15.03.2017	
AddOverrideAnnotation	C	1	21.07.2015	
RemoveCommentedOutCode	C	7	20.07.2019	17
RemoveCommentedOutCode	B	3	03.05.2019	325
RemoveCommentedOutCode	A	5	01.01.2016	18

Tabelle 4.5.: Liste der *RemoveCommentedOutCode BotIssues* wird von den restlichen *BotIssues* getrennt.

Code Smell	Klasse	Anzahl Änderungen	Letzte Änderung	Zeilen Nummer
RemoveMethodParameter	A	16	17.01.2019	
RemoveMethodParameter	B	15	26.07.2018	
AddOverrideAnnotation	B	12	28.01.2020	
ReorderModifier	A	8	26.07.2018	
RenameMethod	B	8	15.03.2017	
AddOverrideAnnotation	A	1	21.07.2019	
AddOverrideAnnotation	C	1	21.07.2015	
RemoveCommentedOutCode	B	3	03.05.2019	325
RemoveCommentedOutCode	A	5	01.01.2016	18
RemoveCommentedOutCode	C	7	20.07.2019	17

Tabelle 4.6.: Die priorisierten Listen.

4.2. Evaluation

Durch die Studie sollen folgende Forschungsfragen beantwortet werden:

RQ1: Haben die Probanden die Gruppierung der zweiten Version des Refactoring-Bots (meine Version) besser empfunden, als die einzelnen Pull Requests aus Version 1 (die ursprüngliche Version des Bots)?

RQ2: Haben die Probanden die Reihenfolge der Pull Requests der zweiten Version als sinnvoll empfunden?

RQ3: Finden die Probanden die Art und Weise wie die Code Smells in Version zwei gruppiert wurden sinnvoll?

Code Smell	Klasse	Anzahl Änderungen	Letzte Änderung	*
RemoveMethodParameter	A	16	17.01.2019	
RemoveMethodParameter	B	15	26.07.2018	
AddOverrideAnnotation	B	12	28.01.2020	
ReorderModifier	A	8	26.07.2018	
RenameMethod	B	8	15.03.2017	
AddOverrideAnnotation	A	1	21.07.2019	
AddOverrideAnnotation	C	1	21.07.2015	
RemoveCommentedOutCode	B	3	03.05.2019	325
RemoveCommentedOutCode	A	5	01.01.2016	18
RemoveCommentedOutCode	C	7	20.07.2019	17

Tabelle 4.7.: Die Liste der *BotIssues* nach der Priorisierung. *Zeilen Nummer

Code Smell	Klasse	Anzahl Änderungen	Summe
RemoveMethodParameter	A	16	
			24
ReorderModifier	A	8	
RemoveMethodParameter	B	15	
			23
RenameMethod	B	8	
AddOverrideAnnotation	B	12	
AddOverrideAnnotation	A	1	14
AddOverrideAnnotation	C	1	
RemoveCommentedOutCode	B	3	
RemoveCommentedOutCode	A	5	15
RemoveCommentedOutCode	C	7	

Tabelle 4.8.: Die Liste der Gruppen vor der Priorisierung.

RQ4: Welche Verbesserungsvorschläge haben die Probanden zur Gruppierung oder Priorisierung der Code Smells?

Bei den Probanden für das Experiment handelte es sich um Studierende des Fachbereich Informatik an der Universität Stuttgart. Diese wurden per E-Mail über verschiedene E-Mail-Verteiler der Universität Stuttgart zur Studie eingeladen oder sie wurden in der Universität Stuttgart, Campus Vaihingen, von mir angesprochen, ob sie an meiner Studie teilnehmen wollen.

Code Smell	Klasse	Anzahl Änderungen	Summe
RemoveMethodParameter	A	16	
			24
ReorderModifier	A	8	
RemoveMethodParameter	B	15	
			23
RenameMethod	B	8	
RemoveCommentedOutCode	B	3	
RemoveCommentedOutCode	A	5	15
RemoveCommentedOutCode	C	7	
AddOverrideAnnotation	B	12	
AddOverrideAnnotation	A	1	14
AddOverrideAnnotation	C	1	

Tabelle 4.9.: Die Liste der Gruppen nach der Priorisierung.

4.2.1. Studienaufbau

Es gibt eine Gruppe von 8 - 10 Probanden. Die Probanden kommen jeweils zu Einzelterminen zu dem Experiment, wobei jeder Proband unabhängig von den anderen Probanden das Experiment durchführt. Jeder Proband bekommt auf einem Laptop zwei GitHub-Repositories auf denen sowohl die Analyse mit SonarQube, als auch der Refactoring-Bot ohne Priorisierung und Gruppierung (im Folgenden nur noch die erste Version des Refactoring-Bots genannt), sowie meine Version des Refactoring-Bots, die im Lauf dieser Arbeit implementiert wurde, angewandt wurden. Die GitHub-Repositories bestehen zum größten Teil aus Java-Code und wurden zuvor aus open source Projekten auf GitHub ausgewählt und mit SonarQube analysiert. Als Repositories wurden *resilience-simulator*[2] und *jackson-databind*[3] ausgewählt. In diesem Java-Code befinden sich viele Code Smells, welche mit dem Refactoring-Bot gerefactored wurden.

Das *resilience-simulator* Repository enthält mit der ersten Bot-Version insgesamt 15 Pull Requests, davon ein *RemoveMethodParameter* Refactoring und 14 *RemoveCommentedOutCode* Refactorings. Mit der zweiten Version des Bots enthält das Repository noch vier Pull Requests, wobei das *RemoveMethodParameter* Refactoring eine eigene Gruppe bildet und die *RemoveCommentedOutCode* Refactorings drei Gruppen bilden. Der Grund für die vom beschriebenen Gruppierungsalgorithmus abweichende Gruppengröße wird in Kapitel 6.5 beschrieben.

Für das *jackson-databind* Repository erstellte die erste Bot-Version sechs Pull Requests, drei *RemoveCommentedOutCode* Refactorings und drei *ReorderModifiers* Refactorings. Diese wurden

[2] https://github.com/LitschiW/resilience-simulator, Commit Id: 87779a9, Date: Aug 15, 2018
[3] https://github.com/FasterXML/jackson-databind, Commit Id: 25263a5, Date: Feb 10, 2020

in der zweiten Version in zwei Gruppen gruppiert, die *RemoveCommentedOutCode* Refactorings bilden hierbei die eine Gruppe, und da die *ReorderModifiers* Refactorings alle in derselben Klasse liegen, bilden sie die zweite Gruppe.

Am Anfang der Studie bekommen die Probanden eine kurze Einweisung was ihre Aufgabe ist, wie sie die GitHub-Repsitories bedienen, welche Daten während der Studie erhoben werden und was ihre Rechte sind. Die Einverständniserklärung, die die Probanden vor der Studie unterschrieben, steht in Anhang A.1. Jeder Proband bearbeitet die Reposiories nacheinander an einem Laptop. Das bedeutet, dass jeder Proband erst ein Repository mit beiden Bot Versionen und dann das zweite Repository mit beiden Bot Versionen bearbeitet. Die Reihenfolge in welcher die Repositories und Bot Versionen von den Probanden bearbeitet werden, variiert dabei.

Die Aufgabe der Probanden besteht darin, die Pull Requests für das aktuell betrachtete Repository zu analysieren, dabei können sie sich völlig frei entscheiden welche der Pull Requests sie annehmen und welche sie ablehnen. Nach dieser ersten Runde werden die Probanden gefragt, wie sie die Arbeit mit diesen Pull Requests fanden, ob ihnen auffällt was sich in der zweiten Version geändert hat, ob sie diese Änderung gut finden und ob sie irgend etwas an der Erzeugung der Pull Requests anders gemacht hätten.

Nachdem das erste Repository mit beiden Versionen des Refactoring-Bots analysiert wurden, und die Zwischenbefragung der Probanden statt gefunden hat, bekommen die Probanden das zweite Repository gezeigt und sie müssen auch hier beide Versionen der Pull Requests betrachten. Die Probanden entscheiden auch hier wieder ob sie die Pull Requests annehmen und geben Feedback. Die Probanden bearbeiten die Repositories so lange, bis sie alle Pull Requests durchgearbeitet haben.

Im Anschluss an das Experiment kommt eine kurze Diskussionsrunde in der die Probanden nochmals Feedback dazu geben können, was sie sinnvoll und was sie weniger sinnvoll, an der Art wie die Pull Requests dargestellt wurden, fanden und ob sie andere Vorschläge haben, wie man die Pull Requests sinnvoll zusammenfassen und priorisieren kann. Nach der Diskussionsrunde folgt das Ausfüllen eines Fragebogens, Anhang A.2 und sie dürfen sich zwei kleine Tafeln Rittersport als Aufwandsentschädigung nehmen. Nachdem alle Probanden an der Studie teilgenommen haben, werden die Probanden dann per E-Mail über den Zweck der Studie aufgeklärt und bekommen gesagt, welche der Versionen von mir implementiert wurde.

4.2.2. Datenerhebung

Für die Erhebung der Daten verwende ich zwei Methoden:

- persönliches Feedback des Users während und nach dem Experiment und
- einen Fragebogen.

In Kapitel 4.2 wurde bereits beschrieben, was für Daten für die Studie erhoben werden. Mit dem Feedback während und nach dem Experiment soll festgestellt werden, welche Version der Pull Requests die Probanden bevorzugen. Mit dem Fragebogen soll dann herausgefunden werden, wie viel Erfahrung die Probanden bereits mit GitHub / GitLab und Pull Requests haben um so Rückschlüsse ziehen zu können, ob die Priorisierung und Gruppierung bei einer speziellen Teilmenge der Probanden besonders gut oder besonders schlecht ankommt.

Feedback

Das Feedback, dass der Proband während und nach dem Experiment gibt, wird mit einer Diktiergerät App[4] auf meinem Handy aufgenommen und anschließend, zur besseren Übersicht, als Protokoll abgetippt. Mit dem Feedback, dass während des Experimentes aufgenommen wird, soll herausgefunden werden, ob der Proband Pull Requests annimmt, die auch einzelne Refactorings enthalten, die er nicht haben möchte (obwohl man dieses Problem mit den Kommentaren lösen kann (Ausblick)), oder ob er gerade wegen diesen Refactorings, die er nicht haben möchte den ganzen Pull Request ablehnt. Außerdem wird er gefragt für wie wichtig er den aktuell betrachteten Pull Request findet. Und es wird aufgenommen, welchen Pull Request er annimmt, bzw. ablehnt.

Mit dem Feedback nach dem Experiment kann der Proband nochmals genauer erörtern, warum er manche Pull Requests abgelehnt hat und warum er manche angenommen hat. Außerdem kann der Proband hier Verbesserungsvorschläge machen, die er sich für den Refactoring-Bot vorstellen kann.

Fragebogen

Der Fragebogen, zu sehen in Anhang A.2, wird dazu verwendet, um herauszufinden wie viel Programmiererfahrung mit Java und Erfahrung mit GitHub / GitLab die Probanden haben.

[4] https://play.google.com/store/apps/details?id=com.media.bestrecorder.audiorecorder

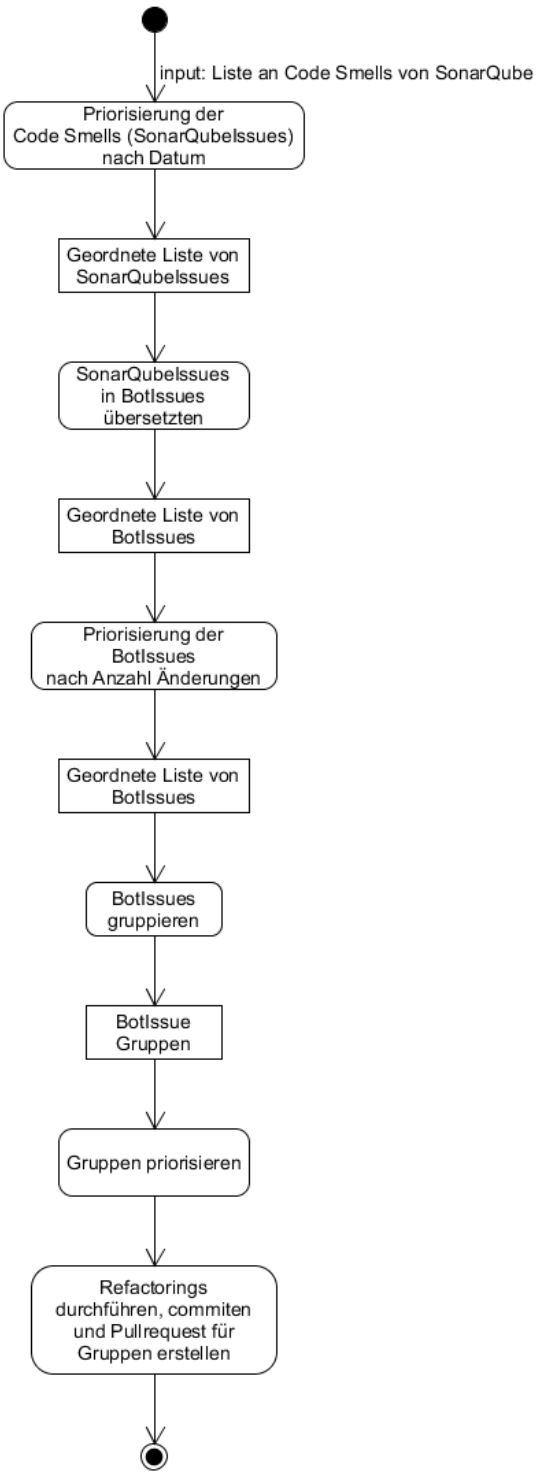

Abbildung 4.1.: Der komplette Algorithmus, der in dieser Arbeit implementiert und vorgestellt wird.

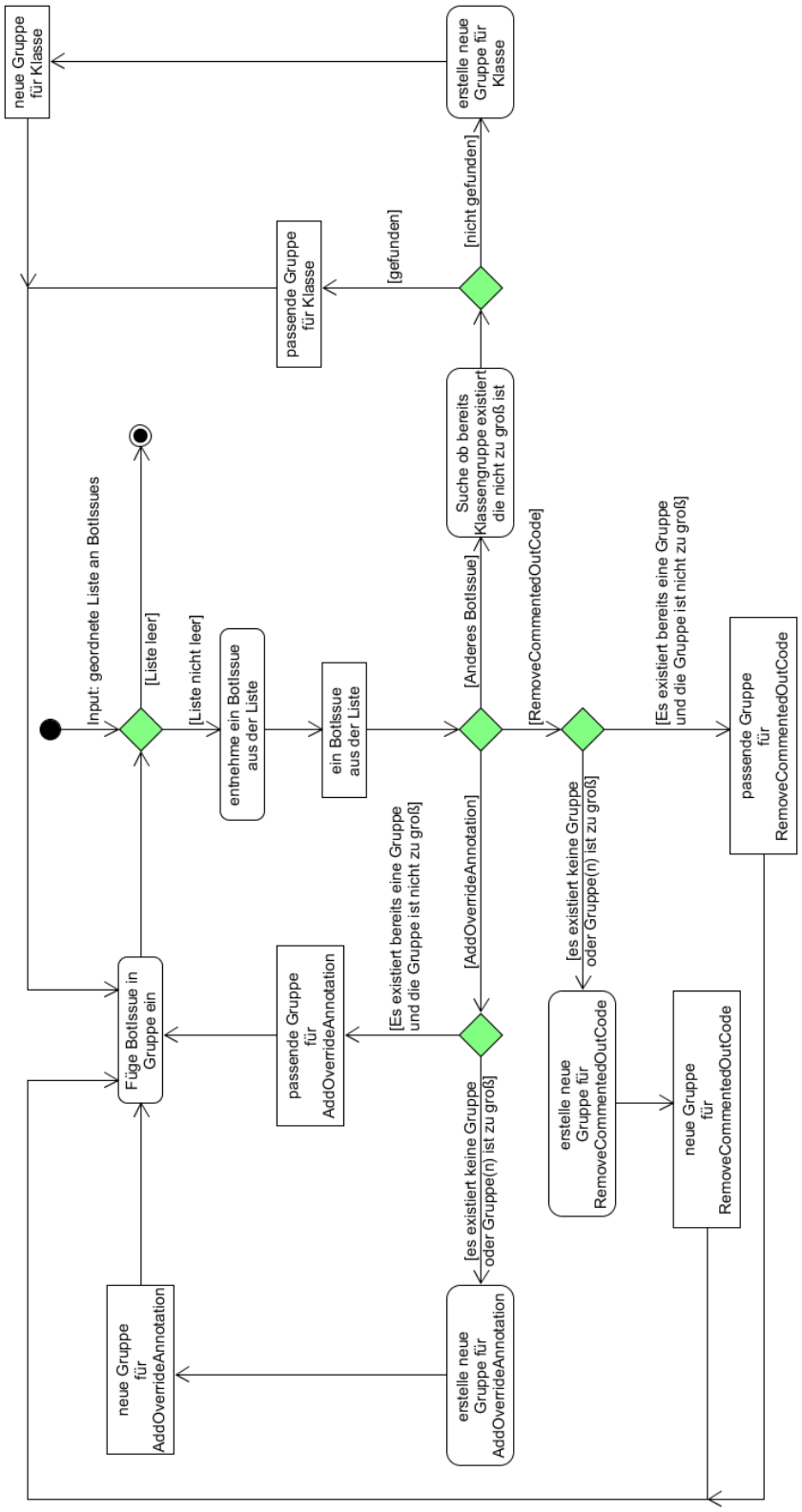

Abbildung 4.2.: Der Gruppierungsalgorithmus zeigt an, wie die Refactorings gruppiert werden können.

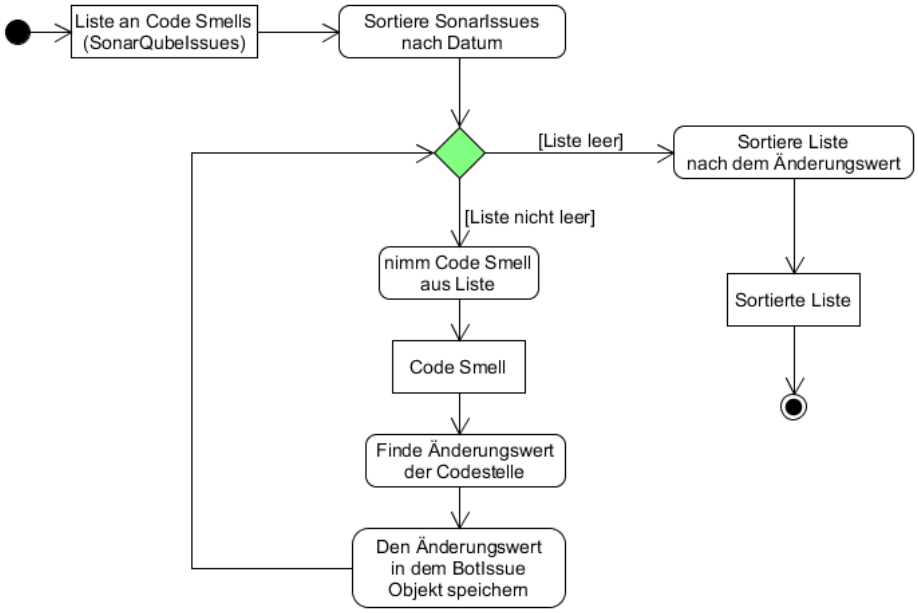

Abbildung 4.3.: Der Priorisierungsalgorithmus für die Code Smells gibt die Reihenfolge an, in der die Code Smells später gruppiert werden.

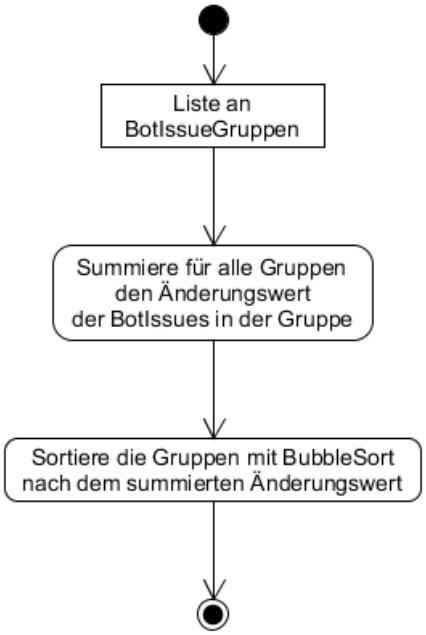

Abbildung 4.4.: Der Priorisierungsalgorithmus für die Gruppen gibt die Reihenfolge an, in der die Pull Requests für die Gruppen erstellt werden.

5. Ergebnisse

In diesem Kapitel werden die Ergebnisse der Studie vorgelegt, hierbei handelt es sich nur um die reinen Ergebnisse, die die Probanden angegeben haben. Eine Diskussion der Ergebnisse erfolgt in Kapitel 6. Dabei wurde dieses Kapitel in die verschiedenen Forschungsfragen untergliedert.

Es haben insgesamt acht Studenten der Universität Stuttgart an der Studie teilgenommen. Die Teilnehmer kommen aus den Studiengängen Softwaretechnik, Informatik und Medieninformatik und befinden sich, bis auf einen Teilnehmer, im Bachelorstudium. Ein Teilnehmer studiert Softwaretechnik im Masterstudium. Alle Teilnehmer haben eine Programmiererfahrung zwischen 1,5 und neun Jahren und haben gute Javakenntnisse, auf einer Skala von 1 bis 10, wobei der Wert 10 sehr gute Kenntnisse bedeutet, liegen die Werte der Probanden zwischen 4 und 9. Dabei hat ein Proband einen Erfahrungswert von 4, einer von 6, drei von 7, zwei von 8 und einer von 9 angegeben. Alle Probanden haben bereits mit GitHub/GitLab gearbeitet, dabei verwenden die meisten Teilnehmer hierbei Git oft, ein Proband hat angegeben Git selten und zwei haben angegeben Git nur gelegentlich zu benutzen, und haben mehr als nur grundlegende Erfahrung mit Git, wobei hier die Erfahrungswerte zwischen 2 und 9 liegen. Auf der Erfahrungsskala von 1 bis 10 hat ein Proband einen Wert von 2, einer von 3, einer von 4, zwei von 6, zwei von 8 und ein Proband einen Wert von 9 angegeben. Alle Teilnehmer benutzen Git an der Uni, vier der Teilnehmer nutzen Git zusätzlich noch für private Zwecke und zwei Teilnehmer nutzen Git sowohl privat als auch beruflich und an der Uni.

5.1. RQ1: Vergleich der Gruppierung

Um die Frage beantworten zu können, welche der beiden Refactoring-Bot Versionen die Probanden bevorzugen würden, haben drei Probanden angegeben die zweite Bot-Version für beide Repositories zu bevorzugen, zwei Probanden haben angegeben die erste Version für beide Repositories zu bevorzugen, drei Probanden haben angegeben bei dem *resilience-simulater* Repository die erste Bot-Version und bei dem *jackson-databind* Repository die zweite Version des Refactoring-Bots zu bevorzugen. Ein Proband, der zuerst die erste Version des Refactoring-Bots für das *resilience-simulater* Repository bekommen hat, war erst davon überzeugt, dass er die einzelnen Pull Requests besser findet, nachdem er dann das Repository mit Version zwei bekommen hat, hat er schon bei dem Anblick über die Pull Request Übersicht seine Meinung geändert und fand die Gruppierung der zweiten Bot-Version besser. Insgesamt bevorzugten zwei Probanden die erste Version des Refactoring-Bots für beide Repositories. Einer

der Probanden der die zweite Version abgelehnt hat würde eine Gruppierung der Refactorings komplett ablehnen, während der andere Proband angegeben hat eine Gruppierung zu bevorzugen, dann jedoch nach einem anderen Gruppierungskriterium. Beide Probanden die für beide Repositories die erste Bot-Version bevorzugen, haben jedoch angegeben, dass sie die gruppierten *ReorderModifier* Refactorings in Version zwei besser finden. Die Probanden, die für das *resilience-simulater* Repository die erste Bot-Version und bei dem *jackson-databind* Repository die zweite Version des Refactoring-Bots besser finden, haben angegeben, dass es sie stört, wenn viele große *RemoveCommentedOutCode* Blöcke, die über mehrere Dateien verteilt sind in einem Pull Request stehen, wie es beim *resilience-simulater* Repository der Fall ist sie haben aber kein Problem damit, wenn kleinere *RemoveCommentedOutCode* Blöcke über mehrere Dateien verteilt in einer Gruppe stehen, so wie es im *jackson-databind* Repository steht, solange es keine anderen Gruppen mit *RemoveCommentedOutCode* Refactorings in einer dieser Klassen gibt.

Zwei Probanden haben angegeben, dass ihnen die Pull Requests aus der ersten Version zu viel sind, einer dieser zwei Probanden hat daraufhin nicht mehr alle Pull Requests der ersten Version angeschaut.

Alle Probanden haben angegeben, dass sie die *ReorderModifier* Gruppe der zweiten Bot-Version im *jackson-databind* Repository viel besser finden, als die drei einzelnen Pull Requests aus der ersten Version.

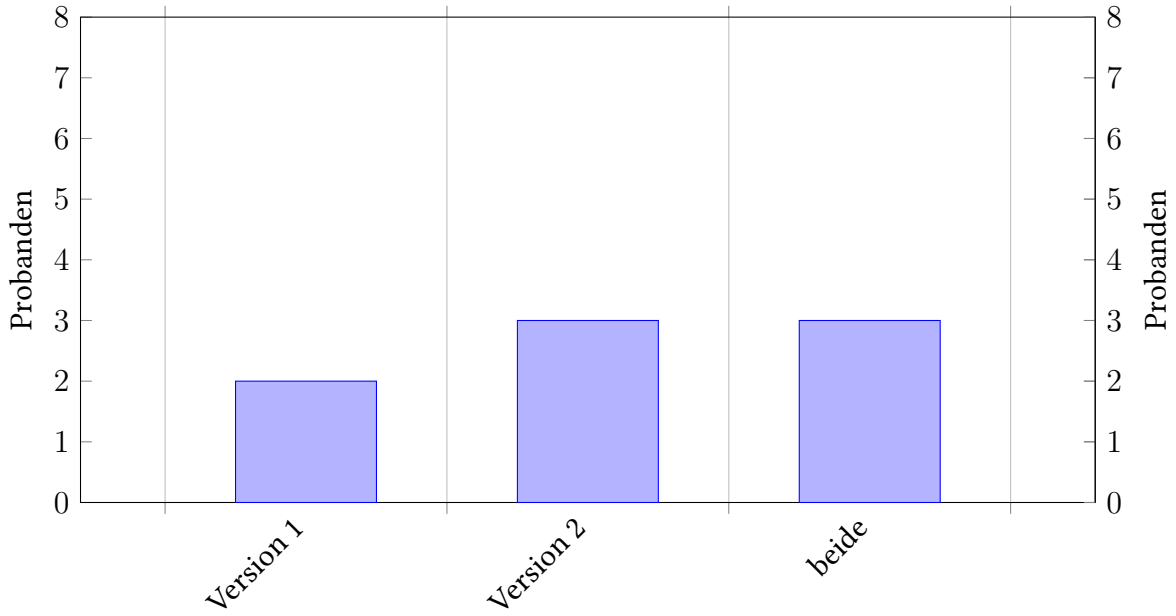

Abbildung 5.1.: Darstellung wieviele Probanden welche Version bevorzugen.

5.2. RQ2: Sinnvolle Reihenfolge der Pull Requests

Auf die Frage, ob die Probanden die Reihenfolge aus der zweiten Bot-Version gut finden, haben sie folgendes angegeben: Drei Probanden ist die Reihenfolge in der die Pull Requests stehen egal. Ein Proband würde es besser finden, wenn man den Arten der Refactorings eine Priorität zuteilt und die gefundenen Code Smells dann nach ihrem Refactoring priorisiert werden. Für Refactorings der gleichen Art gibt es, nach ihm, dann keine weitere Priorisierung. Ein Proband hat angegeben die Pull Requests erst nach dem letzten Änderungsdatum, neuestes Datum eines Code Smells der Gruppe, und dann nach der Anzahl an Codeänderungen zu priorisieren, die Gruppengröße spiele dabei keine Rolle. Ein Proband würde die Priorität einer Gruppe von der Anzahl an enthaltenen Refactorings abhängig machen, so dass Pull Requests mit mehr Refactorings eine höhere Priorität besitzen. Ein Proband würde die Refactorings erst nach der Art priorisieren, dann nach dem letzten Änderungsdatum und dann nach der Anzahl der Änderungen im Code. Ein Proband würde die Refactorings nach Packages gruppieren und nach den Packages priorisieren.

5.3. RQ3: Sinnvolle Gruppierung in Version 2

Um die Forschungsfrage 3, ob in der zweiten Bot Version die Gruppierung sinnvoll gewählt wurde, zu beantworten, haben die Probanden während der Studie selbstständig gesagt, was ihnen an den Gruppen gefällt und was sie ändern würden. Wenn sie dies nicht getan haben, wurden sie gefragt, wie sie die Refactorings gruppiert hätten. Hierbei hat ein Proband erst gemeint, dass er die einzelnen Pull Requests aus der ersten Bot-Version bevorzugt, nachdem er dann aber die Ergebnisse der zweiten Bot-Version gesehen hat, fand er die Gruppen besser. Der Proband hat anschließend noch gesagt, dass er die Refactorings nach Klasse gruppieren würde und im nächsten Schritt alle Refactorings die diese Klasse betreffen in die Gruppe speichern, ein weiterer Proband würde dieses Vorgehen auch bevorzugen. Ein Proband hat angegeben, dass er es in Ordnung findet, wenn die *RemoveCommentedOutCode* Pull Request Gruppen über mehrere Dateien gehen, bei anderen Refactorings würde er das ablehnen. Der Proband hat dann noch angegeben, dass er die Refactorings nach Art und Klasse gruppieren würde und zu große Gruppen so aufteilen würde, dass aufeinander folgende Refactorings in derselben Gruppe stehen. Ein Proband, der für beide Repositories die erste Bot-Version bevorzugt hat, hat angegeben, dass er es besser finden würde, wenn alle Refactorings nach demselben Gruppierungsschema gruppiert werden, aber selbst dann noch die einzelnen Pull Requests bevorzugen würde.

Ein Proband hat angegeben, dass er es gut findet, wenn die ganzen *RemoveCommentedOutCode* Refactorings in einer Gruppe stehen, findet es aber nicht gut wenn die Pull Request Gruppen über mehrere Dateien gehen. Vier Probanden haben dann noch angegeben, dass sie zusätzlich zu der Gruppierung nach der Klasse die Gruppen noch zusätzlich nach der Art des Refactorings abgrenzen wollen. Zwei Probanden würden die Gruppen nur nach der Klasse bilden und dann

alle Refactorings für diese Klasse in einer Klassengruppe speichern. Ein Proband hat angegeben, die *RemoveCommentedOutCode* Refactorings nach Klasse und die restlichen Refactorings nach Art zu gruppieren, also klassenübergreifend.

Alle Probanden, selbst der Proband der von sich aus die Pull Requests nicht gruppieren würde, finden es sinnvoll, dass die drei *ReorderModifier* Refactorings in einer Gruppe stehen.

Insgesamt haben drei Probanden angegeben, dass sie keine Probleme damit haben, dass die *RemoveCommentedOutCode* Refactorings über mehrere Dateien hinweg gehen, vier Probanden haben gesagt, dass sie es bevorzugen würden, wenn ein Pull Request nur eine Datei bearbeitet und ein Proband würde es in Ordnung finden, wenn ein Pull Request mehrere Dateien bearbeitet, solange diese sich im selben Package befinden.

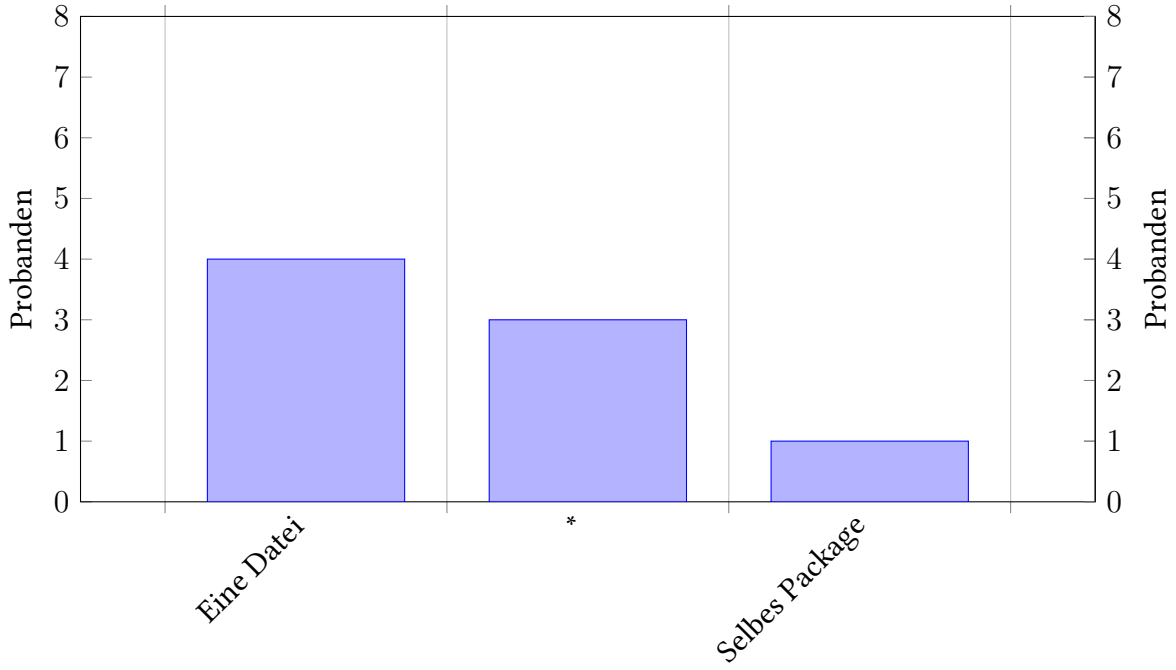

Abbildung 5.2.: Darstellung wieviele Probanden es bevorzugen würden, wenn die *RemoveCommentedOutCode* Pull Requests nur über eine Datei gehen. *Eine oder Mehrere Dateien.

5.4. RQ4: Verbesserungsvorschläge

Von den Probanden wurden folgende Verbesserungsvorschläge gemacht:

- Die Namen der Pull Request Gruppen verkürzen.
- Die Refactorings nur nach Klasse gruppieren.

- Die Refactorings nach Klassen und Art gruppieren.
- Bei größeren Gruppen, die in mehrere Gruppen gesplittet werden, den Split so aufteilen, dass aufeinanderfolgender Code in einer Klasse in derselben Gruppe steht.
- In den Namen der Pull Requests nur die Art des Refactorings und in den Commit-Messages die genauere Beschreibung mit Klasse und Zeile.
- Für alle Refactoringarten ein einheitliches Gruppierungsschema verwenden.

Für die Änderung des Namens, der Commit-Message für den Pull Request, haben die Probanden angegeben, soll der Dateiname mit der Art der Refactorings stehen. Ein Proband hat angegeben, dass er den Namen so gestalten würde: Klassenname, Art des Refactorings, [betroffene Code-Zeilen]*. Wobei hier [betroffene Code-Zeilen]* bedeutet, dass nacheinander alle Code-Zeilen aufgelistet werden, sollen in denen was geändert wurde, Code-Blöcke dürfen dabei zusammengefasst werden. Ein Proband hat angegeben in den Namen des Pull Requests nur den Namen des Refactorings und die geänderten Zeilen, aber nicht den Klassennamen zu schreiben.

Bei der Gruppierung der Refactorings nach der Refactoringart haben viele Probanden angegeben, dass sie dies bevorzugen würden, da man dann in einem Pull Request sich nur mit einer Art des Refactorings befasst und sich keine Gedanken darüber machen muss wie die anderen Refactoringarten funktionieren und somit effizienter arbeiten kann, da man nicht umdenken muss. Dieses Argument wurde ebenfalls von den Probanden verwendet, die die Refactorings noch zusätzlich nach Klassenzugehörigkeit gruppieren würden, dabei hat man dann laut den Probanden dann noch den Vorteil, dass man sich nur mit der Funktion einer Klasse auseinandersetzen muss und die Pull Request noch übersichtlicher sind.

6. Diskussion

Dieses Kapitel widmet sich der Frage wie man mit den Ergebnissen aus dem Kapitel 5 die verschiedenen Forschungsfragen beantworten kann. Dieses Kapitel ist ebenso wie Kapitel 5 in die verschiedenen Forschungsfragen unterteilt, wobei sich die Kapitel 6.1 bis Kapitel 6.4 mit den Forschungsfragen 1 bis 4 befassen und Kapitel 6.5 sich den Limitationen widmet, welche die Studie betroffen haben.

6.1. RQ1: Vergleich der Gruppierung

Für die erste Forschungsfrage sollte beantwortet werden, ob die Probanden die zweite Bot-Version der ersten Bot-Version vorziehen. Da drei Probanden angegeben haben, die zweite Version des Refactoring-Bots für beide Repositories zu bevorzugen, stehen 37,5% für die volle Nutzung der zweiten Refactoring-Bot Version. Weitere 37,5% haben angegeben die zweite Version für das *jackson-databind* Repository zu bevorzugen und für das *resilience-simulater* Repository die erste Version. Die Ablehnung der zweiten Refactoring-Bot Version liegt bei 25%.

Obwohl die Akzeptanz der zweiten Bot Version größer als die komplette Ablehnung ist, würde ich die Forschungsfrage eins mit nein beantworten, da 25% gegen die Anwendung der zweiten Version sind und weitere 37,5% die zweite Version nicht vollständig unterstützen. Bevor die Gruppierung übernommen wird, würde ich das Gruppierungskriterium noch einmal überarbeiten. Dies hat den Grund, dass viele der Probanden, die die zweite Version nicht vollständig unterstützen oder ganz abgelehnt haben, angegeben haben, dass sie die zweite Version mit einem anderen Gruppierungskriterium unterstützen würden. Auch wegen weiteren Gründen aus den Kapiteln 6.3 und 6.4 lässt sich ableiten, dass eine Gruppierung von den Probanden erwünscht ist, diese aber anders gestaltet werden sollte.

6.2. RQ2: Sinnvolle Reihenfolge der Pull Requests

Forschungsfrage zwei befasst sich damit, ob die Probanden die Priorisierung der Pull Requests sinnvoll finden.

Obwohl hier das Verhältnis zwischen der Annahme und der Ablehnung der zweiten Version gleich wie in Kapitel 6.1 ist, 37,5% nehmen die Version zwei an und 25% lehnen sie ab, würde ich hier das Priorisierungskriterum ohne Änderungen annehmen, da ebenfalls 37,5% der Probanden angegeben haben, dass ihnen die Reihenfolge in der die Pull Requests stehen egal ist und deshalb auch kein Problem mit der implementierten Priorisierung haben.

6.3. RQ3: Sinnvolle Gruppierung in Version 2

Ob die Gruppierung in der zweiten Bot-Version als sinnvoll empfunden wurde, soll mit der dritten Forschungsfrage beantwortet werden.

Insgesamt finden es sechs Probanden sinnvoller, die Refactorings nach der Refactoringart und der Klassenzugehörigkeit zu gruppieren.

Der Proband, der lieber einzelne Pull Requests den Gruppen bevorzugt, hat noch angegeben, dass er Gruppen sinnvoll finden kann, wenn alle Refactorings nach einem einheitlichen Kriterium gruppiert werden, hierbei ist ihm das Kriterium jedoch egal. Da dies bei der implementierten Version nicht der Fall ist, findet er die implementierte Gruppierung auch nicht sinnvoll.

Diese Ergebnisse führen dazu, dass die dritte Forschungsfrage mit nein, die Probanden finden die Gruppierung der zweiten Bot-Version nicht sinnvoll, beantwortet werden muss. Wie in Kapitel 6.1 erwähnt sollte die Gruppierung der zweiten Bot-Version nur mit Änderungen akzeptiert werden. Die Änderungen die hierfür vorgesehen werden, wie in Kapitel 6.4 und 7 auch nochmals erwähnt wird, sind die Gruppierung nach der Art und der Klasse der Refactorings, da sich hierfür 75% der Probanden ausgesprochen haben.

6.4. RQ4: Verbesserungsvorschläge

Die Probanden haben viele Verbesserungsvorschläge gemacht:

- Alle Probanden würden eine Namensänderung der gruppierten Pull Requests bevorzugen: dabei hatten alle den Wunsch, dass die Namen deutlich kürzer sein sollten, damit sie besser lesbar sind. So kann man den Namen so abändern, dass er für eine Gruppe von Refactorings, die über mehrere Dateien geht, die Art des Refactorings angibt und die Zeilen, die betroffen sind. Bei Gruppen, die nur eine Datei bearbeiten, kann man zusätzlich noch angeben, um welche Datei es sich handelt. Dies ist meiner Ansicht nach sehr nützlich und sollte unbedingt vor dem Übernehmen der Gruppierung und Priorisierung implementiert werden.

- Außerdem finden es die meisten Probanden besser, wenn man die Refactorings nach der Refactoringart und nach den Klassen aufteilt. Dies wurde in vorherigen Kapiteln bereits erwähnt und hat ebenfalls meine Unterstützung, da ich wie in Kapitel 3 bereits geschrieben habe keine Literatur für die Gruppierung von Code Smells bzw. Refactorings gefunden habe. Dies lässt sich jedoch nicht immer erfüllen, da Methoden auch klassenübergreifend aufgerufen werden können, aber man kann dies so lösen, indem man dieses Refactoring in die Gruppe speichert, in der die Methode steht.
- Große Gruppen sollten so aufgeteilt werden, dass Refactorings, die im Code hintereinander stehen in derselben Gruppe stehen und diese nicht wild gemixt sind. Auch dies ist meiner Meinung nach sehr hilfreich für die Lesbarkeit der Pull Requests und sollte auch übernommen werden.

6.5. Limitationen

Eine technische Limitation zum Zeitpunkt der Studie bestand darin, dass GitHub seine Richtlinien geändert hatte und man dann mit den aktuellen Versionen des Refactoring-Bots keine Pull Requests mehr erstellen konnte. Diesem Problem wurde entgegengewirkt, indem man den Refactoring-Bot laufen lies und dann anhand der Ausgabe auf der SwaggerUI die Pull Requests manuell erstellte. Dies wurde sowohl für die erste als auch für die zweite Version des Bots durchgeführt.

Aufgrund der Beschränkung auf maximal 15 Pull Requests in der ersten Version, damit die Probanden nicht unnötig lange vor einem Repository sitzen, lieferte der Bot nicht immer optimale Ergebnisse zum Gruppieren. Da es aber genügend Refactorings gab, die nach den in Kapitel 4.1.2 genannten Kriterien gruppiert werden konnten, wurde dies so beibehalten. Da für die Gruppierten Pull Requests mehr Refactorings durchgeführt werden konnten, da ein Limit von 15 Pull Requests hier bedeutet, dass 15 Refactoring Gruppen als Pull Requests ins Repository geladen werden, mussten bei dem *resilience-simulator* Repository aus den Pull Request Gruppen die Refactorings entfernt werden, die in der ersten Version nicht vorhanden waren um die Konsistenz der beiden Repositories beizubehalten, da diese Pull Requests beide Male noch automatisch durch den Refactoring-Bot erstellt wurden.

Mögliche Störvariablen des Experimentes sind:

- Probanden haben ein gutes Verhältnis zum Studienleiter und geben daher überwiegend positives Feedback.
- Probanden haben einen schlechtes Verhältnis zum Studienleiter und geben daher überwiegend negatives Feedback.
- Probanden wissen bereits, um was es in der Bachelorarbeit geht und können deswegen ihre Antworten so manipulieren, dass ein gewolltes Ergebnis herauskommt.

Diese Störvariablen lassen sich dadurch kontrollieren, dass allen Probanden für beide Repositories sowohl die erste als auch die zweite Version des Refactoring-Bots gezeigt wird und keiner der Probanden weiß welche Version von mir implementiert wurde. Die dritte Störvariable lässt sich dadurch kontrollieren, indem nur Personen eingeladen wurden, die nicht wissen um was es in der Bachelorarbeit geht.

7. Zusammenfassung und Ausblick

Die Motivation dieser Arbeit war, dass automatische Refactorings bessere und schnellere Resultate erzielen als manuelle Refactorings und das Ziel war durch das Gruppieren und Priorisieren der Code Smells die Codequalität effizienter zu verbessern. In Kapitel 2 wurden dann die Grundlagen, von Bots, Refactorings und Code Smells, die für diese Arbeit benötigt werden beschrieben. Im weiteren Verlauf der Arbeit habe ich untersucht, wie sich die Code Smells priorisieren und gruppieren lassen. Hierfür habe ich einige wissenschaftliche Veröffentlichungen gelesen und in Kapitel 3 zusammengefasst. Dabei bin ich zu dem Entschluss gekommen, dass ich die Code Smells erst nach dem letzten Änderungsdatum und dann nach der Anzahl an Änderungen priorisiere, so dass der Code Smell mit den meisten Änderungen die höchste Priorität bekommt und bei Code Smells mit der gleichen Anzahl an Änderungen das letzte Änderungsdatum entscheidet. Als Gruppierungskriterien habe ich gewählt, dass *RemoveCommentedOutCode* und *AddOverrideAnnotation* Refactorings jeweils eine eigene Gruppe bekommen. Bei den restlichen Refactorings habe ich mich dann dazu entschieden, dass sie klassenweise gruppiert werden. Da es wichtig ist, dass die Gruppen nicht zu groß werden, da sie sonst zu unübersichtlich werden, habe ich noch die Gruppengrößen beschränkt. Dabei habe ich mich dafür entschieden, dass *AddOverrideAnnotation* Gruppen auf 20 Refactorings, *RemoveCommentedOutCode* Gruppen auf 10 Refactorings und Klassengruppen auf 20 Refactorings zu begrenzen. Für die Studie wurden mehrere Studenten der Universität Stuttgart eingeladen und insgesamt 8 Studenten haben zugesagt an der Studie teilzunehmen. Jeder dieser Studenten musste dann insgesamt 2 GitHub Repositories analysieren, beide Repositories wurden mit beiden Bot-Versionen analysiert. Alle Studenten haben beide Repositories mit beiden Bot-Versionen bekommen, jedoch in unterschiedlicher Reihenfolge. Für die Studie mussten sie dann die Pull Requests der Repositories analysieren und dabei laut denken was sie an den Repositories gut oder schlecht finden. Dabei mussten die Studenten entscheiden, welche Bot-Version sie bevorzugen, ob die Priorisierung und Gruppierung meiner Bot-Version sinnvoll ist und welche weiteren Verbesserungsvorschläge sie für den Refactoring-Bot haben. Durch die Studie bin ich zu den Ergebnissen gekommen, dass die Priorisierung sinnvoll gewählt wurde, bei der Gruppierung jedoch ein paar Änderungen vorgenommen werden müssen, da sich viele Probanden für eine andere Art der Gruppierung, nämlich nach Art des Refactorings und der Klassenzugehörigkeit, ausgesprochen haben.

Ausblick

Der Bot kann nach den Wünschen der Probanden aus Kapitel 5.4 angepasst werden, damit er die Refactorings nach Art und Klasse gruppiert. Diese Vorgehensweise vereinfacht es auch wenn neue Refactorings für den Bot implementiert werden. Bei SonarQube-Analysen die Code Smells über mehrere Zeilen hinweg entdecken, muss man aufpassen, dass das Refactoring dieser Code Smells keine Auswirkungen auf die weitere Analyse hat, so kann es zum Beispiel passieren, dass wenn ein Refactoring mehrere Zeilen so manipuliert, dass neue Zeilen hinzu kommen oder alte Zeilen wegfallen, dass dann die Analyse von SonarQube nicht mehr mit dem aktuell vom Refactoring-Bot betrachteten Quellcode übereinstimmt und der Refactoring-Bot daher Probleme bei der weiteren Ausführung bekommt. Bei Refactorings die Code-Stellen löschen oder so manipulieren, dass Code-Zeilen entfernt oder hinzugefügt werden, muss man darauf achten, dass:

1. Diese Code Smells als eine der letzten gerefactored werden, um Wechselwirkungen zu anderen Refactorings zu vermeiden, siehe Kapitel 4.1.3 und
2. diejenigen Refactorings, die die größte Zeilennummer betreffen zuerst gerefactored werden.

Eine weitere nützliche Funktion, die noch nicht implementiert wurde ist das Entfernen von Refactorings aus Pull Request Gruppen. Hier könnte der User mit Hilfe der Kommentare in GitHub die unerwünschten Refactorings aus den Pull Request Gruppen entfernen.

Außerdem könnte der Bot von den Usern noch lernen, indem er analysiert welche Pull Request Gruppen die User bevorzugt annehmen oder ablehnen und dann seine zukünftige Gruppenerstellung und Priorisierung darauf anpassen, dass er erst die Gruppen erstellt, die die User bevorzugt annehmen. Dieses Botlernen geschieht dann für jedes Entwicklerteam separat.

A. Anhang

A.1. Einverständniserklärung

Einverständniserklärung

Marvin Knodel

Februar 2020

Datenerhebung

Die für diese Studie erhobenen Daten, wie E-Mail Adresse, Fragebogen, Aufzeichnung der Stimme und die Einverständniserklärung werden nur im Rahmen der Bachelorarbeit von Marvin Knodel verwendet, sie werden nicht an Dritte weitergegeben. Die Aufnahmen der Stimme werden nach der Studie anonym in ein schriftliches Protokoll übertragen und die Aufnahmen werden nach der Auswertung der Studie sofort gelöscht, sie werden nicht Dritten vorgespielt. Der Fragebogen und das aufgenommene Protokoll enthalten eine Probanden Nummer, die nicht mit den persönlichen Daten des Probanden in Verbindung gebracht werden kann, der Proband bleibt anonym. Die Fragebögen sowie die abgetippten Protokolle werden anonym aufbewahrt. Diese Einverständniserklärung wird ebenfalls aufbewahrt. Ergebnisse und Inhalte der Studie und Befragung werden nach der Studie anonym in der Bachelorarbeit verwendet. Die Teilnahme an der Studie ist freiwillig und der Proband hat das Recht jederzeit die Studie abzubrechen. Im Falle eines Abbruches der Studie werden alle bis dahin erhobenen Daten sofort gelöscht und nicht weiter verwertet.

Ich erkläre mich damit einverstanden, dass oben beschriebene Daten in dem Umfang der oben beschrieben wurde erhoben, ausgewertet und gespeichert werden dürfen und ich wurde über meine Rechte und den Umfang der Datenerhebung, Datenverwertung und Datenspeicherung Informiert. (Bitte schreiben Sie die Jahreszahl voll aus, 2020)

.....
(Ort und Datum)(Unterschrift)

A.2. Fragebogen

Probanden Nummer: ..... (wichtig damit nachher das richtige Protokoll mit dem dazugehörigen Fragebogen ausgewertet werden kann, Proband bleibt anonym)

Bitte beantworten Sie folgende Fragen wahrheitsgemäß. Kreuzen Sie die Antwort an, die am ehesten auf Sie zutrifft.

1. Wie viel Programmiererfahrung haben Sie (in Jahren)?
   .....

2. Wie gut schätzen Sie Ihre Erfahrung in Java ein?
   nicht gut sehr gut
   O O O O O O O O O O O

3. Was studieren Sie?
   .....

4. Haben Sie schon einmal mit GitHub / GitLab gearbeitet?
   Ja Nein
   O O

   (a) Wenn Sie Frage 4 mit Ja beantwortet haben, wie häufig arbeiten Sie mit GitHub/GitLab?
   nie selten gelegentlich oft sehr oft
   O O O O O

   (b) Wenn Sie Frage 4 mit Ja beantwortet haben, wo arbeiten Sie mit GitHub/GitLab? (mehrere Kreuze möglich)
   Beruflich Privat An der Uni
   O O O

   (c) Wenn Sie Frage 4 mit Ja beantwortet haben, wie schätzen Sie Ihre Erfahrung mit GitHub/GitLab ein?
   nicht gut sehr gut
   O O O O O O O O O O O

Literaturverzeichnis

[Asi42] I. Asimov. „Runaround. Astounding science fiction“. In: *New York, NY: Street and Smith Pub* (1942) (zitiert auf S. 16).

[BFB99] K. Beck, M. Fowler, G. Beck. „Bad smells in code“. In: *Refactoring: Improving the design of existing code* (1999), S. 75–88 (zitiert auf S. 16).

[FFZR15] F. A. Fontana, V. Ferme, M. Zanoni, R. Roveda. „Towards a prioritization of code debt: A code smell intensity index“. In: *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. IEEE. 2015, S. 16–24 (zitiert auf S. 13, 18).

[Fow18] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018 (zitiert auf S. 13, 16, 18, 23).

[GPD14] G. Gousios, M. Pinzger, A. v. Deursen. „An exploratory study of the pull-based software development model“. In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, S. 345–355 (zitiert auf S. 24).

[LSZ17] C. Lebeuf, M.-A. Storey, A. Zagalsky. „Software bots“. In: *IEEE Software* 35.1 (2017), S. 18–23. DOI: 10.1109/MS.2017.4541027 (zitiert auf S. 15, 16).

[MB10] E. Murphy-Hill, A. P. Black. „An interactive ambient visualization for code smells“. In: *Proceedings of the 5th international symposium on Software visualization*. ACM. 2010, S. 5–14 (zitiert auf S. 18, 19, 23).

[SYA+12] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, T. Dybå. „Quantifying the effect of code smells on maintenance effort“. In: *IEEE Transactions on Software Engineering* 39.8 (2012), S. 1144–1156 (zitiert auf S. 13, 18, 23).

[VMD16] S. A. Vidal, C. Marcos, J. A. Dıaz-Pace. „An approach to prioritize code smells for refactoring“. In: *Automated Software Engineering* 23.3 (2016), S. 501–532 (zitiert auf S. 17, 26).

[WB19] M. Wyrich, J. Bogner. „Towards an autonomous bot for automatic source code refactoring“. In: *Proceedings of the 1st International Workshop on Bots in Software Engineering*. IEEE Press. 2019, S. 24–28. DOI: 10.1109/BotSE.2019.00015 (zitiert auf S. 13, 15, 17, 22–24, 26).

Alle URLs wurden zuletzt am 01. 04. 2020 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