Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Implementierung eines Bots zum Vorschlagen automatisch durchgeführter Refactorings

Timo Pfaff

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Stefan Wagner
Betreuer:	Marvin Wyrich, M.Sc. Kai Mindermann, M.Sc.
Beginn am:	10. April 2018
Beendet am:	9. Oktober 2018

Kurzfassung

Software altert über die Jahre und verliert an Struktur. Um diesem Prozess entgegenzuwirken, werden in der Softwareentwicklung Refactorings eingesetzt. Refactorings sind Änderungen des Programmcodes, um die Qualität zu steigern, ohne dabei das Verhalten der Software zu verändern. Refactorings stellen somit einen wichtigen Bestandteil dar, um die Softwarequalität auch über mehrere Jahre in einem guten Zustand zu halten. Gleichzeitig werden zum Durchführen von Refactorings aber Ressourcen, wie zum Beispiel Personenstunden, benötigt, die wiederum den Unterschied zwischen Erfolg und Misserfolg eines Softwareprojekts bedeuten können.

Um Ressourcen zu sparen, befasst sich diese Bachelorarbeit mit der Implementierung eines Bots, der automatisch Refactorings durchführt und diese dem Entwickler vor dem Übernehmen der Änderungen in den Hauptentwicklungszweig zum Review anbietet. Die Änderungen sollen nicht automatisch übernommen werden, da ein Fehler während des Refactorings gerade bei sicherheitskritischer Software schwerwiegende Folgen haben kann. Der Bot wurde anschließend anhand eines Beispielsprojekts getestet und gefundene Probleme wurden diskutiert. Des Weiteren wurden zwei Evaluationskonzepte vorgestellt, mit denen der Bot auf verschiedene Kriterien evaluiert werden kann. Das Ergebnis der Arbeit ist die Implementierung des Bots, der die genannte Grundfunktionalität bereitstellt und Refactorings an realen Projekten durchführen kann.

Abstract

Over the years software loses its structure as a part of the aging process. In software development refactorings are used to counteract against this process. Refactorings are changes on the software code to increase the quality of it without changing its behaviour. Therefore refactorings are an important part to keep the quality of the software as high as possible. At the same time refactorings need resources, like personal hours, which can determine the difference between success or failure of the sofware project.

To conserve resources, this bachelor thesis addresses the implementation of a bot, that executes automatic refactorings and provides the changes to the developer for review before applying the changes to the main development branch. The changes should not be taken over automatically, since an error during refactoring, especially with safety-critical software, can have serious consequences. The bot was then tested on a sample project and found problems were discussed. Furthermore, two evaluation concepts were presented, with which the bot can be evaluated on different criteria. The result of the work is the implementation of the bot wich provides the basic functionality and the bot is able to perform refactorings on real projects.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

1 Einleitung

In diesem Kapitel wird die Motivation, Aufgabenstellung und das Ziel dieser Arbeit dargelegt und der Aufbau dieser Arbeit beschrieben.

1.1 Motivation

Refactorings sind ein wichtiger Bestandteil, um langlebigen Programmcode in einem guten Zustand zu halten. Martin Fowler [FB99] definiert Refactorings als einen Prozess, der eine Software so verändert, dass sich die interne Struktur verbessert, das nach außen zu beobachtende Verhalten aber gleich bleibt.

Weiterhin beschreibt Fowler [FB99], welche Vorteile Refactorings mit sich bringen. Diese Vorteile werden in diesem Abschnitt beschrieben. Durch die Veränderungen des Programmcodes, zum Beispiel um neue Funktionalitäten hinzuzufügen oder um Fehler innerhalb des Programms zu beheben, verschlechtert sich die Struktur zunehmend. Refactorings helfen dabei, die verloren gegangene Struktur wieder zu erlangen. Refactorings helfen außerdem dabei, den geschriebenen Code leserlicher zu machen. Das ist vor allem dann von Vorteil, wenn an einem Codeabschnitt gearbeitet wird, an dem schon länger nichts mehr verändert wurde und sich dadurch der Entwickler erneut in diesen Abschnitt einlesen muss. Selbiges gilt auch für Entwickler, die zum ersten Mal mit dem Code konfrontiert werden. Dadurch, dass Refactorings den Code strukturierter und leserlicher machen, helfen sie dabei schneller Code zu schreiben und somit schneller zur Fertigstellung der Software zu gelangen.

Die somit gesparten Ressourcen können in der Folge in die Implementierung oder Verbesserung weiterer Funktionalitäten gesteckt werden. Da manuelle Refactorings aber fehleranfällig sind [GDM12] und je nach Länge und aktuellem Stand des Codes mitunter viel Zeit in Anspruch nehmen können, könnte es sinnvoll sein, die Refactorings mittels eines Tools automatisch durchzuführen.

Knapp 90% aller durchgeführten Refactorings werden dennoch manuell durchgeführt [MPB12]. Laut einer Studie [PK13] ist fehlendes Vertrauen in die Refactoring-Tools ein Grund für das Ablehnen dieser. Aus den genannten Gründen soll in dieser Arbeit ein Bot implementiert werden, der dazu dient, Ressourcen zu sparen, indem er automatisch Refactorings durchführt. Er soll dabei helfen, angestaute Meldungen der statischen Codeanalyse abzuarbeiten, ohne dabei den Workflow der Entwickler zu stören. So soll am Ende ohne erheblich erhöhtem Zeitaufwand für die Entwickler die Codequalität gesteigert werden. Dadurch, dass der Bot vor dem Merge in das Repository die Bestätigung des Entwicklers einholt, soll verhindert werden, dass fehlerhafte Refactorings in das System kommen. Somit hat der Entwickler immer noch die Kontrolle über die Änderungen, was dem fehlenden Vertrauen in Tools entgegenwirken kann.

1.2 Aufgabenstellung und Ziel

Im Rahmen dieser Bachelorarbeit soll die Implementierung eines Bots erfolgen, der vollautomatisch Refactorings durchführt. Damit der Bot weiß, an welchen Stellen im Code Probleme sind und Refactorings benötigt werden, sollen vorher die Ergebnisse einer statischen Codeanalyse ausgelesen werden. Diese statische Codeanalyse soll mittels eines Tools, wie beispielsweise SonarQube[1], erfolgen. Auf Basis der gefundenen Probleme soll der Bot vollautomatisch Refactorings durchführen. Nachdem der Bot das Refactoring durchgeführt hat, müssen die Änderungen den Entwicklern zur Kontrolle vorgelegt werden. Nur im Falle einer Zustimmung sollen die gemachten Änderungen des Bots in das Repository übernommen werden. Um dies zu realisieren, soll eine Verknüpfung mit GitHub[2] bereitgestellt werden, sodass Pull-Requests für die durchgeführten Änderungen erstellt werden können. Sobald die Implementierung abgeschlossen und der Bot einsatzbereit ist, soll er anhand eines Beispielprojekts getestet werden.

1.3 Aufbau der Arbeit

Diese Arbeit ist in fünf Kapitel aufgeteilt. Im folgenden Grundlagen-Kapitel werden wiederkehrende und für die Arbeit wichtige Begriffe definiert. Anschließend werden verwandte Arbeiten vorgestellt, die bereits in diesem Bereich durchgeführt wurden. Auf das Grundlagen-Kapitel folgt die Implementierung des Bots. Hier werden zunächst die Rahmenbedingungen erläutert und wichtige Implementierungsdetails aufgezeigt. Am Ende dieses Kapitels wird erklärt, wie der Bot gestartet werden kann und anhand eines Beispielprojekts getestet wurde. Außerdem werden gefundene Probleme und mögliche Lösungen für diese diskutiert. Um die Nützlichkeit des Bots in zukünftigen Arbeiten zu evaluieren, folgen anschließend zwei Evaluationskonzepte. Diese umfassen die Beschreibungen von möglichen Varianten, um den Bot auf unterschiedliche Kriterien zu evaluieren. Am Ende dieser Arbeit ist eine Zusammenfassung und ein Ausblick zu finden.

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

2 Grundlagen

In diesem Kapitel werden für diese Arbeit wichtige Begriffe beschrieben und verwandte Arbeiten vorgestellt.

2.1 Statische Codeanalyse

Unter statischer Codeanalyse versteht man die Überprüfung von Quellcode auf Fehler, ohne dabei den Code auszuführen [Lou06].

Nach Zhioua et al. [ZSR14] kann eine statische Codeanalyse auf zwei verschiedene Arten durchgeführt werden kann. Bei der ersten Variante wird die Analyse von Entwicklern manuell mittels Code-Reviews durchgeführt. Die andere Möglichkeit ist es, die Analyse automatisch von Tools durchführen zu lassen. Die zuerst genannte Variante kann, je nach Anzahl der Codezeilen viel Zeit in Anspruch nehmen, weshalb es sinnvoll ist, automatische Tools zu verwenden. Novak et al. [NK+10] beschreiben aber, dass nicht alle Befunde von automatischen Tools gefunden werden können. Dementsprechend sollten automatische Analysen nicht ausschließlich, sondern zusätzlich zu manuellen Code-Reviews verwendet werden. Bei diesen sollte in der Folge speziell auf die Befunde geachtet werden, die nicht so einfach von automatischen Tools erkannt werden.

Zhioua et al. [ZSR14] beschreiben außerdem mehrere Techniken, die bei der statischen Codeanalyse eingesetzt werden. Zu diesen Techniken gehören unter anderem Modellprüfung, Kontrollflussanalyse und Datenflussanalyse. Die Datenflussanalyse dient dazu, zu untersuchen, welche Datenabhängigkeiten innerhalb des zu untersuchenden Programms bestehen. Außerdem wird diese Analyse dazu verwendet, um herauszufinden, wie sich das Programm unter verschiedenen Eingabewerten verhält [ZSR14].

Innerhalb dieser Arbeit soll die statische Codeanalyse dazu genutzt werden, um Codestellen zu finden, für die ein Refactoring notwendig ist. Weiterhin soll hierfür das automatisches Tool SonarQube verwendet werden. SonarJava[1], das Analyseprogramm, das in SonarQube integriert ist, verwendet unter anderem die Datenflussanalyse, um mögliche Probleme im Code zu finden. Auf Basis der Befunde bekommt der Bot mitgeteilt, welche Art von Problem vorliegt und an welcher Stelle im Programmcode dieses Problem zu finden ist.

[1] https://www.sonarsource.com/products/codeanalyzers/sonarjava.html

2.2 Refactoring

Wie zu Beginn in Kapitel 1.1 beschrieben, handelt es sich nach Fowler [FB99] bei einem Refactoring um die Veränderung der internen Struktur einer Software, um diese verständlicher und wartbarer zu machen, ohne dabei das Verhalten der Software zu verändern. Refactorings bringen daher mehrere Vorteile mit sich. Zum einen verbessern sie die Struktur von Software, die während des natürlichen Alterungsprozesses verloren geht. Außerdem helfen Refactorings dabei, den Programmcode leichter zu verstehen, was wiederum dabei hilft Software effizienter zu entwickeln. Auch können sie dabei helfen Fehler innerhalb des Programms zu finden. Sinnvolle Zeitpunkte um ein Refactoring anzustoßen sind, wenn eine neue Funktionalität implementiert werden soll, wenn Fehler behoben werden müssen oder im Zuge eines Code-Reviews [FB99].

Refactorings laufen nach einem bestimmten Muster ab. Mens et al. [MT04] beschreiben in ihrer Arbeit den Ablauf wie folgt: Zunächst muss bestimmt werden, an welcher oder welchen Stellen ein Refactoring durchgeführt werden soll. Nachdem die Stelle im Code festgelegt wurde, muss ausgewählt werden, welche Art von Refactoring durchgeführt werden soll. Vor der tatsächlichen Durchführung des Refactorings muss gewährleistet werden, dass durch die Veränderung des Codes das Verhalten des Programms nicht verändert wird. Ist dies sichergestellt, kann das Refactoring durchgeführt werden. Nach Abschluss des Refactorings sollte noch geprüft werden, ob Aspekte der Software, wie beispielsweise Qualitätsmerkmale, durch das Refactoring verändert wurden.

Wie schon beim Ablauf eines Refactorings im vorherigen Abschnitt beschrieben, existieren mehrere Arten von Refactorings. Um einen kurzen Überblick zu geben, werden im Folgenden häufig eingesetzte Refactorings genannt und beschrieben. Die hier ausgewählten Beispiele gehören zu den meist verwendeten Refactoring-Arten, die von GitHub-Entwicklern benutzt wurden, wenn Entwickler für ihren Code, der in einem GitHub-Repository liegt, Refactoring-Aktivitäten durchgeführt haben [STV16]:

- **Rename:** Eine Variable, Methode oder Klasse wird umbenannt, um die Absicht des jeweiligen Konstrukts besser darzustellen.
- **Move Method:** Eine Methode wird in eine andere Klasse verschoben, weil sie zu der neuen Klasse gehört.
- **Move Class:** Eine Klasse wird in ein anderes Paket verschoben, da sie semantisch zu einem anderen Paket gehört.
- **Extract Method:** Ein Codefragment wird aus einer Methode in eine andere Methode ausgelagert, deren Name den Zweck des Codefragments widerspiegelt.
- **Extract Class:** Teile einer Klasse werden in eine neue Klasse ausgelagert, da die momentane Klasse die Logik für zwei Klassen enthält.
- **Inline Method:** Der Methodenaufruf wird durch den Rumpf der Methode ersetzt und die Methode wird entfernt.
- **Pull Up Method:** Eine Methode wird in eine Superklasse hochgezogen, da zwei oder mehr Kindklassen eine solche Methode mit gleichem Verhalten besitzen.

Die Beschreibungen der Beispiele sind, bis auf „Move Class“, aus [FB99] entnommen. „Move Class“ stammt aus [STV16]. Es gilt anzumerken, dass diese nur ein Bruchteil der durchführbaren Refactorings darstellen. Eine umfangreichere Auflistung ist in den angegebenen Quellen zu finden.

2.3 Bot

Unter einem Bot versteht man ein Programm, das ohne durch einen Menschen gesteuert zu werden, automatisch vorprogrammierte Aktionen ausführt [GXWW11].

Bots werden für unterschiedliche Aufgaben programmiert und eingesetzt. Je nachdem für welchen Zweck sie eingesetzt werden und auf welche Art und Weise sie interagieren, können Bots unterschiedlich charakterisiert werden. Es existieren Bots, die dazu programmiert wurden, einfache Anweisungen auszuführen, vom Nutzer gewünschte Informationen auszugeben oder Transaktionen durchzuführen. Bots können außerdem dazu genutzt werden die Produktivität von Teams zu erhöhen, indem sie beispielsweise Termine in einen Kalender eintragen oder die Kommunikation und Zusammenarbeit fördern [LSZ18].

2.4 Pull-Request

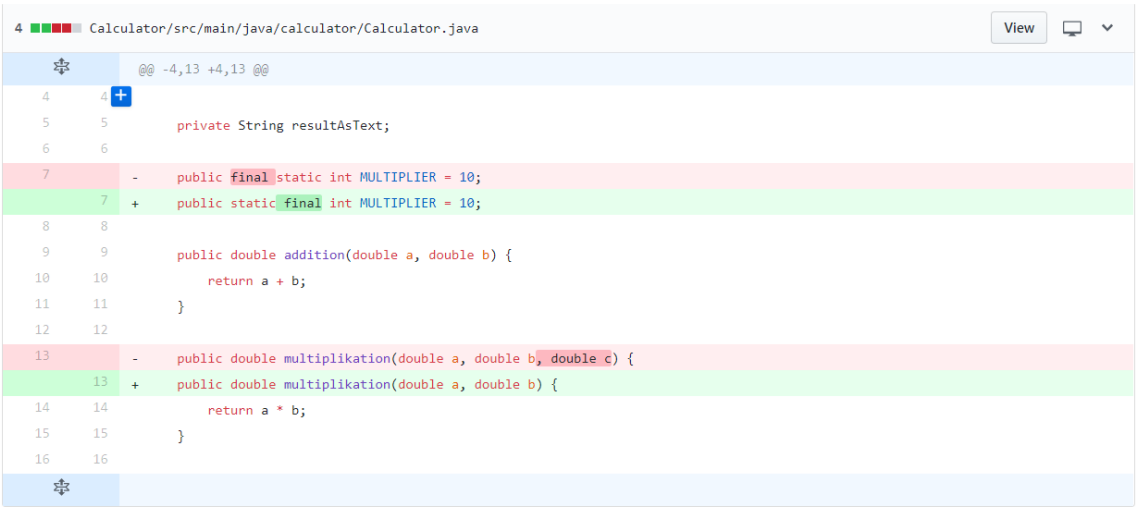

Abbildung 2.1: Beispiel eines Pull-Requests [Git18]. Rot markierte Zeilen zeigen den Stand vor den Änderungen. Grün markierte Zeilen zeigen den Stand nach den Änderungen. Die genauen Änderungen in diesen Zeilen werden durch die Markierung hervorgehoben. Bildschirmfoto vom Autor.

Um die Änderungen der Refactorings dem ursprünglichen Code gegenüberzustellen und so dem Entwickler auf einen Blick zu zeigen welche Stellen verändert wurden sollen nach den durchgeführten Refactorings Pull-Requests auf GitHub erstellt werden. Pull-Requests werden in einer pull-basierten Entwicklung, wie GitHub sie unterstützt, verwendet. Entwickler, die an einem Projekt anderer Entwickler mitarbeiten oder weitere Funktionalitäten für das Projekt beisteuern wollen klonen

besagtes Projekt und können so unabhängig Änderungen vornehmen. Zu jedem Zeitpunkt können gemachte Änderungen mittels eines Pull-Request für das ursprüngliche Repository vorgeschlagen werden. Wie Anhand von Abbildung 2.1 zu sehen ist, zeigt der Pull-Request alle Änderungen des Codes im Vergleich zur aktuellen Version auf dem Master-Branch. Die Besitzer des ursprünglichen Repositories können basierend auf diesem Vergleich entscheiden, ob die Änderungen in ihr Repository übernommen werden oder ob der Pull-Request geschlossen und somit die Änderungen verworfen werden. Im Falle einer Zustimmung wird ein Merge durchgeführt [GPD14].

2.5 Verwandte Arbeiten

Im Bereich Refactoring existieren bereits viele Arbeiten und Tools. In diesem Abschnitt soll darauf eingegangen werden, welche Tools bereits entwickelt wurden. Diese unterscheiden sich nicht nur in der Anzahl durchführbarer Refactorings, sondern auch in der Art der Automatisierung.

Vollautomatische Tools führen Refactorings selbstständig aus, sodass der Nutzer maximal die Teile des Programms, die gerefactored werden sollen, festlegt und den Start initialisiert. Zu den vollautomatischen Tools gehören unter anderem Autorefactor[2], The Spartanizer [GO17] und Code-Imp [MÓ11]. Im Folgenden Abschnitt soll die technische Seite eines solchen automatischen Refactorings anhand von The Spartanizer dargelegt werden.

Dieses Tool ist als Eclipse-Plugin verfügbar und verwendet sogenannte „Tipper“, die wiederum Transformationsregeln darstellen. „Tippers“ werden in verschiedene Kategorien eingeteilt. Eine dieser Kategorien sind beispielsweise die „Nominal Tippers“. Hierzu gehören Transformationsregeln, die zum Beispiel für das Umbenennen von Variablen verwendet werden. Der Nutzer hat die Möglichkeit auszuwählen, welche Arten von „Tippers“ angewendet werden sollen und auf welchen Teil des Programmcodes das Refactoring angewendet wird. Wenn der Nutzer das Refactoring anstößt, werden alle ausgewählten Refactorings auf Basis der Transformationsregeln und den Rahmenbedingungen des Nutzers vollautomatisch durchgeführt [GO17].

Mit diesem Tool ist es außerdem möglich, Refactorings halbautomatisch durchzuführen. Damit ist gemeint, dass anstelle von mehreren Refactorings auf bestimmte Teile des Codes, einzelne Refactorings manuell für eine bestimmte Stelle ausgewählt werden und das Tool anschließend dieses eine Refactoring durchführt. Gegebenenfalls muss der Entwickler noch weitere Rahmenbedingungen, wie zum Beispiel den neuen Namen bei einer Umbenennen-Operation, angeben. Der Nutzer interagiert dementsprechend häufiger mit dem Tool. Ein solches halbautomatisches Refactoring ist in vielen gängigen IDE´s, wie Eclipse[3] oder IntelliJ[4], bereits integriert. Außerdem wurden auch für diesen Zweck Tools implementiert. Zu diesen gehören unter anderem RefactorIT[5], BeneFactor [GM11] und WitchDoctor [FGL12].

Während RefactorIT, ähnlich wie in den Entwicklungsumgebungen, Refactorings mittels eines hinzugefügten Menüs bereitstellt, verfolgen BeneFactor und WitchDoctor eine andere Art der Interaktion zwischen Tool und Entwickler. Diese Tools erkennen anhand der Aktionen des Entwicklers ob dieser

[2] http://autorefactor.org/
[3] https://www.eclipse.org/
[4] https://www.jetbrains.com/idea/
[5] https://sourceforge.net/projects/refactorit/

gerade, egal ob bewusst oder unbewusst, im Begriff ist ein Refactoring manuell durchzuführen und schlägt ihm vor, dieses Refactoring dem Tool zu überlassen. Der Entwickler könnte zum Beispiel gerade eine Variable umbenennen. Das Tool würde in diesem Fall einschreiten und auf Wunsch des Entwicklers diese Umbenennen-Operation übernehmen und automatisch durchführen.

Auch Gábor Szoke [Sző17] beschreibt in seiner Arbeit, wie Refactorings automatisiert werden können. Es handelt sich bei seinem Lösungsansatz ebenfalls um eine halbautomatische Variante, da der Entwickler einzelne Refactorings auswählt und dieses eine Refactoring anschließend automatisch durchgeführt wird. Im Folgenden wird beschrieben, wie seine Lösung aussieht. Um die Stellen zu finden, die ein Refactoring benötigen verwendet er das Tool PMD. Es handelt sich dabei um ein Tool, das zur statischen Codeanalyse verwendet wird. Zusätzlich wird ein abstrakter Syntaxbaum erstellt, durch den mittels eines Algorithmus die betroffenen Elemente des Quellcodes gefunden und verändert werden. Die Änderungen werden mittels einer Datei, welche den Vorher-Nacher-Vergleich der Änderungen zeigt, den Entwicklern präsentiert.

Weiterhin wurden auch Tools entwickelt, die selbst keine Refactorings durchführen, sondern lediglich den Entwickler dabei unterstützen manuelle Refactorings zu realisieren. Eines dieser Tools ist KinEdit [Ter15]. Idee dieser Tools ist es, dass die Entwickler alle Schritte des Refactorings manuell durchführen, aber beispielsweise durch das Anzeigen von Referenzen für bestimmte Elemente, wie es bei KinEdit der Fall ist, unterstützt werden.

Der im Rahmen dieser Arbeit zu entwickelnde Bot gliedert sich dabei zwischen den vollautomatischen und halbautomatischen Tools ein. Einerseits wird das Refactoring vollautomatisch auf Basis einer vorher durchgeführten statischen Codeanalyse durchgeführt, andererseits gibt es aber immer noch die Interaktion mit dem Entwickler, da die Änderungen vor einem Commit erst einmal vorgeschlagen werden und die Bestätigung durch den Entwickler benötigt wird.

Neben den bereits genannten Tools existiert noch ein weiteres Tool, welches am ähnlichsten zu dem Bot dieser Arbeit ist. Der Name des Tools ist FaultBuster [SNF+15]. FaultBuster funktioniert so, dass eine statische Codeanalyse auf dem Code, der verändert werden soll, durchgeführt wird. Der Code selbst wird dafür aus einem Versionsverwaltungssystem geholt. Die Ergebnisse der Analyse können anschließend über eine webbasierte Schnittstelle abgefragt werden. Über diese Schnittstelle ist es außerdem möglich, den Refactoringmechanismus anzustoßen. Die durch die Refactorings gemachten Änderungen werden den Entwicklern, ähnlich einem Pull-Request, mittels einer Datei, die den Vorher-Nacher-Vergleich enthält, gezeigt. Der Entwickler kann die Änderungen in seiner Entwicklungsumgebung übernehmen oder verwerfen.

Der Bot, der in dieser Arbeit entwickelt werden soll, unterscheidet sich von der zuletzt genannten Arbeit hauptsächlich durch die Anbindung an GitHub und die Art, wie die Änderungen den Entwicklern vorgelegt werden.

3 Implementierung

In diesem Kapitel wird zunächst beschrieben, mit welchen Technologien der Bot implementiert wurde, anschließend folgt eine Beschreibung der Umsetzung, wobei auch Designentscheidungen begründet werden, gefolgt von einem Abschnitt zum Test des Bots. Der letzte Teil dieses Kapitels beschreibt Probleme, die unter anderem während des Tests gefunden wurden. Mögliche Lösungen für die gefundenen Probleme werden ebenfalls in diesem Teil diskutiert. Der Programmcode des Bots und das Script für die Erstellung des Pull-Requests liegen in einem GitHub-Repository[1]. Das Script zum Starten des Bots und eine Beispiel-Properties-Datei liegen in einem eigenen GitHub-Repository[2].

3.1 Implementierungsgrundlage

Für die Implementierung des Bots wurde die Entwicklungsumgebung Eclipse[3] verwendet und die verwendete Programmiersprache ist Java. Als Build-Management-Tool und zum Verwalten der benötigten Abhängigkeiten wurde Apache Maven[4] eingesetzt. Weitere verwendete Bibliotheken sind JavaParser[5], um die Refactorings zu implementieren, JSON[6], um die Antworten der Requests zu parsen und Git[7] für die Versionsverwaltung. Für das Erstellen der Pull-Requests wurde außerdem hub[8] verwendet. Dabei handelt es sich um ein Kommandozeilen-Tool mit dem Git-Befehle für GitHub erweitert werden.

3.2 Umsetzung des Bots

Dieser Abschnitt beschäftigt sich mit der Umsetzung der Kernelemente des Bots. Zunächst wird das Konzept der Umsetzung mithilfe einer dafür erstellten Abbildung erläutert. Anschließend wird die Implementierung aller Kernelemente nacheinander beschrieben und die getroffenen Designentscheidungen begründet.

[1] https://github.com/Refactoring-Bot/Refactoring-Bot
[2] https://github.com/Refactoring-Bot/RefactoringScripts
[3] https://www.eclipse.org/
[4] https://maven.apache.org/
[5] https://github.com/javaparser/javaparser
[6] https://www.json.org/
[7] https://git-scm.com/
[8] https://github.com/github/hub

3.2.1 Konzept der Umsetzung

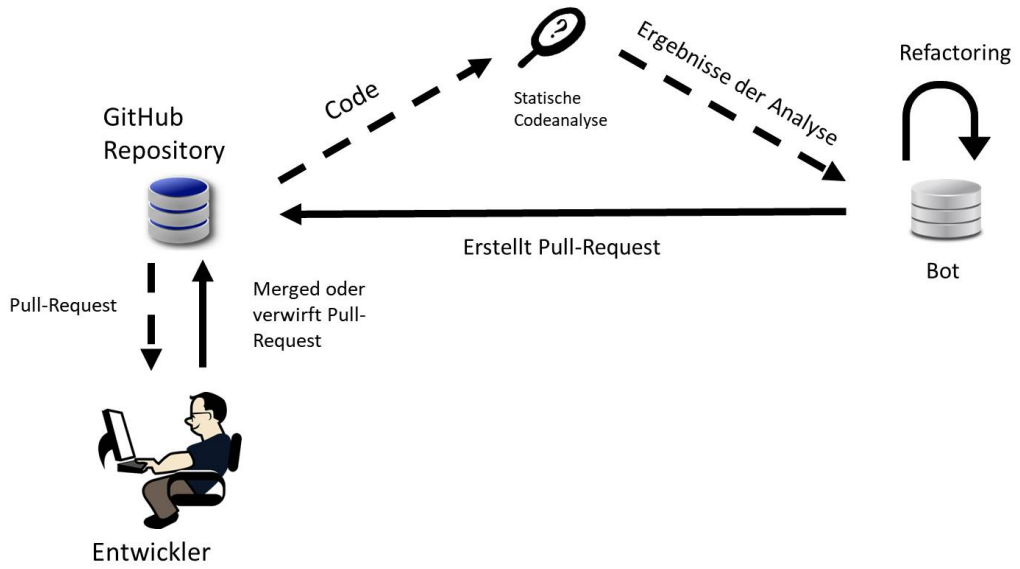

Abbildung 3.1: Konzept der Umsetzung. Gestrichelte Linien zeigen Datenfluss an. Durchgezogene Linien sind Aktionen, die durchgeführt werden.

Anhand von Abbildung 3.1 kann man sehen, wie ein Refactoring abläuft. Der Code, der sich in einem GitHub-Repository befindet, wird mittels einer statischen Codeanalyse auf Befunde geprüft. Die Ergebnisse dieser Analyse stellen wiederum den Input für den Bot dar. Auf Basis dieser Ergebnisse führt der Bot das Refactoring durch und erstellt einen Pull-Request mit den Änderungen auf GitHub. Der Entwickler kann jetzt den Pull-Request überprüfen und diesen entweder übernehmen oder verwerfen. Die Kernelemente des Bots sind somit das Auslesen der statischen Codeanalyse, das Refactoring selbst und das Erstellen des Pull-Requests. Die Implementierung dieser Kernelemente wird in den nächsten Kapiteln erläutert.

3.2.2 Auslesen der statischen Codeanalyse

Damit der Bot weiß, an welchen Stellen er das Refactoring ansetzen muss und welche Art von Refactoring er durchführen muss, werden zunächst die aktuellen Befunde von Sonarcloud[9] für das Projekt mittels eines Http-Get-Requests an die API abgefragt. Der Request ist dabei so aufgebaut, dass lediglich offene, also noch nicht bearbeitete, Befunde berücksichtigt werden. Die Antwort besteht aus einem JSON mit allen Befunden, aus dem die benötigten Parameter für die Refactorings extrahiert werden:

- **Komponente (component):** Dieser Parameter enthält den Pfad zur Java-Klasse, in der das Problem vorliegt. Vor dem Pfad ist außerdem der Sonarcloud Projektname angehängt.

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

- **Projekt (project):** Dieser Parameter ist der Name des Sonarcloud Projekts. Er wird verwendet, um aus dem Parameter Komponente den Pfad zu extrahieren. Dies geschieht, indem mittels einer „Substring“-Operation der Projektname aus der Komponente entfernt wird.
- **Codezeile (line):** Eine oder mehrere Zeilen innerhalb der betroffenen Klasse, in denen das Problem vorliegt.
- **Regel (rule):** Jeder Sonarcloud Befund hat eine bestimmte Regel, die festlegt, um welche Art von Befund es sich handelt. Über diese wird entschieden, welche Art von Refactoring notwendig ist.
- **Nachricht (message):** Beschreibung des Problems. Über diesen Parameter kann je nach Art des Problems beispielsweise der Name einer Variablen oder Methode entnommen werden.

Der Antwort JSON enthält noch weitere Parameter. Die Genannten sind dabei die, die zum aktuellen Zeitpunkt benötigt und verwendet werden. Für weitere Refactorings könnten wiederum andere Parameter wichtig sein. So ist bei der Implementierung dieser zu entscheiden, welche Informationen notwendig sind. Die notwendigen Informationen müssen dementsprechend aus dem JSON entnommen werden.

Zum Testen wurde in der Anfangszeit der Implementierung SonarQube lokal ausgeführt. Später wurde auf Sonarcloud umgestellt. Sonarcloud ist ein Cloud Service, der auf SonarQube basiert. Der Bot führt dabei die statische Codeanalyse nicht selbst durch, sondern es wird davon ausgegangen, dass das Ergebnis der Analyse auf Sonarcloud vorliegt. Diese Ergebnisse müssen dabei zum aktuellen Stand des Codes, den der Bot verändern soll, passen, da ansonsten Codezeilen, zu denen SonarQube-Befunde erstellt hat, gar nicht mehr existieren könnten. Auch könnte das festgestellte Problem bereits vom Entwickler gelöst worden sein. Für den Prototyp wurde vorläufig davon ausgegangen, dass die Analyse und der Programmcode auf demselben Stand sind. Für das Testen an einem Beispielprojekt wurde die Analyse selbstständig durchgeführt und somit sichergestellt, dass dies der Fall ist. Mögliche Lösungen für dieses Synchronisierungsproblem werden in Abschnitt 3.4 diskutiert.

3.2.3 Durchführung des Refactorings

Auf Basis der im vorherigen Kapitel angesprochenen statischen Codeanalyse führt der Bot die Refactorings durch, die er beherrscht. Hierbei kommt der JavaParser zum Einsatz. Er wird dazu verwendet, eine Java-Datei zu parsen. Dabei wird ein abstrakter Syntaxbaum erstellt, um durch die Elemente der Java-Klasse navigieren zu können, die vom Befund angegebenen Stellen zu finden und so zu verändern, wie es durch das Refactoring definiert ist. Für jedes Refactoring wurde dabei eine eigene Java-Klasse erstellt.

Die Refactorings wurden ohne das Verwenden bereits bestehender Refactoring-Tools selbst mithilfe des JavaParsers implementiert. Der Grund für diese Entscheidung ist, dass der Bot spezielle Refactorings durchführt, die auf Regeln und Befunde von SonarQube zugeschnitten sind.

Der Bot unterstützt zum aktuellen Zeitpunkt zwei verschiedene Refactorings:

- **Hinzufügen der @Override Annotation:** Dieses Refactoring fügt einer Methode, die eine Methode der Superklasse überschreibt, die @Override Annotation hinzu, falls diese vergessen wurde.

- **Umsortieren der Modifier nach der Java-Sprachspezifikation:** Die Java-Sprachspezifikation legt eine Reihenfolge fest, nach der die Modifier sortiert werden sollen. Beispielsweise kommt der Modifier „static“ stets vor den Modifier „final“ und nicht umgekehrt. Das Refactoring strukturiert den Modifier, sodass er den Regeln entspricht.

Das Umsortieren der Modifier stellt hierbei einen besonderen Fall dar. Beim Einlesen der Klasse, die gerefactored werden soll, mit der Compilerunit des JavaParsers werden die Modifier bereits richtig sortiert. Das heißt jedes andere Refactoring beinhaltet dieses Refactoring bereits. Es ist trotzdem zusätzlich als eigenes Refactoring implementiert, da der Fall auftreten kann, dass das der einzige Befund von SonarQube ist. Wenn dieser Fall eintritt und der Bot somit kein anderes Refactoring durchführen würde, würde dieser Befund ignoriert und nicht gerefactored werden.

Es wurden außerdem zwei weitere Refactorings implementiert und ausprobiert:

- **Entfernen einer nicht verwendeten Variable:** Variablen, die nicht gelesen werden, werden durch dieses Refactoring entfernt. Es kommt zu solchen Befunden, indem beispielsweise vergessen wurde, eine angelegte Variable, die im Endeffekt nicht verwendet wird, oder eine Variable, die durch Änderungen am Code nicht mehr benötigt wird, zu entfernen.
- **Entfernen nicht verwendeter Parameter von Methoden:** Hier wird ein Parameter von einer Methode entfernt, wenn dieser nicht verwendet und somit nicht benötigt wird.

Diese beiden Refactorings sind aber noch nicht als komplett fertig einzustufen, da beim späteren Testen Probleme aufgetaucht sind, die vorher nicht bedacht wurden. Um sagen zu können, dass der Bot diese richtig unterstützt, muss die Implementierung erweitert werden. Mehr dazu ist in Abschnitt 3.4 zu finden.

Die Auswahl der Refactorings beruht mit einer Ausnahme auf den am häufigsten auftretenden Befunden von Sonarcloud für die Programmiersprache Java. Sonarcloud bietet die Möglichkeit, alle öffentlichen Befunde nach bestimmten Suchkriterien zu filtern[10]. In der angegebenen Url wurde als Filter bereits Java als Programmiersprache ausgewählt. Über die weiteren Filter kann eine Liste mit den unterschiedlichen Arten von Befunden angezeigt werden. Diese Liste ist absteigend nach der Anzahl des Auftretens der Befunde sortiert.

Die Ausnahme stellt dabei das Entfernen einer nicht verwendeten Variable dar. Dieses Refactoring wurde zu Beginn der Implementierung aus der Liste der möglichen Befunde ausgewählt, ohne dabei auf die Häufigkeit des Auftretens zu achten. Es wurde testweise ausgewählt, um zu überprüfen, ob es in einem akzeptablen zeitlichen Rahmen möglich ist, Refactorings selbst zu implementieren ohne dabei auf bereits bestehende Codezeilen von veröffentlichten Refactoring-Tools zurückzugreifen.

Außer den drei anderen realisierten Refactorings existieren Befunde, die noch häufiger auftreten. Einer der häufigsten Befunde ist beispielsweise, dass Variablennamen einer gewissen Namenskonvention folgen sollen. Dieses Refactoring wurde nicht implementiert, da Projekte oder Teams mitunter ihre eigene Namenskonvention haben und dementsprechend nicht wollen, dass ihre Namensgebung verändert wird. Ein weiteres Beispiel ist das Entfernen von bestimmten Kommentaren oder Kommentarblöcken. Es existieren dabei Befunde, bei denen im Kommentar beispielsweise geschrieben steht, unter welcher Lizenz das Projekt veröffentlicht ist. Solche Kommentare wurden

[10] https://sonarcloud.io/explore/issues?languages=java&resolved=false

bewusst erstellt und sollten somit nicht entfernt werden. Aus diesem Grund und weil es schwer zu unterscheiden ist, ob der Kommentar sinnvoll oder unnötig ist, wurde auch dieses Refactoring nicht implementiert.

3.2.4 Automatischer Commit und Bereitstellen des Pull-Requests

```
changeDir(){
        cd "$location"
}
commit(){
 git add -A
 git commit -a -m "$commitMessage"
 git push -u origin "$branchName"
}
pullRequest() {
  hub pull-request -h "$repoOwner":"$branchName" "$commitMessage"
}

changeDir
git checkout -b "$branchName"
commit
pullRequest
git checkout "master"
```

Listing 3.1: Script zum Erstellen der Pull-Requests auf GitHub. Die Variablen werden vom Bot an das Script übergeben.

Nachdem das Refactoring durchgeführt wurde, sollen die Änderungen committed werden und ein Pull-Request mit den Änderungen auf GitHub erstellt werden. Listing 3.1 zeigt das für diesen Zweck erstellte Script. Über das Script wird zunächst in den richtigen Ordner, in dem das lokale Git-Repository liegt, navigiert. Anschließend wird der Commit durchgeführt. Die Commit-Nachricht wird dabei in den jeweiligen Refactoring-Klassen in einer Methode festgelegt. Alle Refactoring-Klassen implementieren zu diesem Zweck das Interface „Refactoring“, welches eine Methode „getCommitMessage“ besitzt. Nach dem Commit erstellt das Script einen neuen Branch. Der Name der Branches setzt sich aus dem Wort „RefactoringBranch“ und dem Schlüssel des Sonarqube-Befundes zusammen. Auf diese Weise ist garantiert, dass jeder Branch einen einzigartigen Namen hat und keine Probleme bei der Erstellung auftreten können. Der letzte Schritt des Scripts ist es, den Pull-Request zu erstellen. Hier kommt das in Abschnitt 3.1 genannte Kommandozeilen-Tool hub zum Einsatz. Mittels des Befehls „hub pull-request“ wird der Pull-Request für den eben erstellten Branch auf GitHub erstellt. Der Entwickler kann sich die Änderungen anschauen und entscheiden ob der Pull-Request akzeptiert werden soll und somit die Änderungen in den Master-Branch übernommen werden oder ob die Änderungen verworfen werden sollen.

Für ein durchgeführtes Refactoring wird jeweils genau ein Pull-Request erstellt. Der Grund dafür ist, dass der Bot den Entwickler möglichst wenig in seinem Workflow stören soll. Daher wäre es kontraproduktiv, wenn erst alle Refactorings durchgeführt werden und dann ein Pull-Request für alle Refactorings erstellt wird. Durch mehrere Pull-Requests mit wenigen Änderungen, kann der Entwickler innerhalb kurzer Zeit und ohne dabei in seinem Workflow gestört zu werden Änderungen akzeptieren oder verwerfen.

Eine weitere Designentscheidung an dieser Stelle ist die Anzahl gleichzeitig offener Pull-Request vom Bot dieser Arbeit. Jedes Team soll dabei selbst entscheiden können, wie viele Pull-Request für ihr GitHub-Repository akzeptabel sind. Vor jedem neuen Refactoring wird über einen Request an GitHub abgefragt, wie viele offene Pull-Requests vom Bot gerade existieren. Falls die maximale Anzahl erreicht ist, führt der Bot vorerst keine weiteren Refactorings für dieses Projekt durch. Der Wert für die maximale Anzahl an offenen Pull-Requests wird derzeit in einer Properties-Datei festgelegt. Mehr dazu ist im nächsten Abschnitt zu finden.

3.2.5 Starten des Bots

Bevor der Bot gestartet werden kann, muss eine Properties-Datei angelegt werden:

```
maxNumberOfOpenPullRequests = 5
sonarCloudProjectName = Test:Test:master
fileLocation = c://Users/XYZ/Test/git/Calculator/
gitHubProject = Refactoring-Bot/RefactoringTest
bashLocation = c:/Programme/Git/bin/bash.exe
pullRequestScriptLocation = c:/Users/XYZ/pull-request.sh
gitHubLoginName = MaxMustermann
listOfDoneIssuesLocation = c://Users/XYZ
```

Listing 3.2: Beispiel einer Properties-Datei.

Wie anhand von Listing 3.2 zu erkennen ist werden in dieser Properties-Datei wichtige Parameter festgelegt. Zu diesen gehören zum einen Pfade zu den Files oder Scripts und zum anderen Werte, wie die maximale Anzahl an offenen Pull-Requests oder der Sonarcloud Projektname, der benötigt wird, um die richtigen Befunde für das Projekt zu finden.

```
pullRepoCode(){
 cd c://Users/XYZ/Test/git/
 git pull
}
pullBotCode(){
 cd c://Users/XYZ/git/Refactoring-Bot/
 git pull
}

run(){
mvn install
mvn exec:java -Dexec.args="C:\Users\XYZ\TestProperties.properties"
}

pullRepoCode
pullBotCode
run
```

Listing 3.3: Script zum Starten des Bots. Die Pfade sind dabei Musterpfade, die für jedes Projekt angepasst werden müssen.

Um den Bot komfortabel starten zu können, wurde das Script in Listing 3.3 erstellt. Es wird davon ausgegangen, dass sowohl das Repository des Projekts, das gerefactored werden soll, als auch das Respository des Bots geklont wurden und jeweils in einem lokalen Git-Repository verfügbar sind. Des Weiteren sollte eine für das Projekt passende Properties-Datei angelegt worden sein.

Das Script führt mehrere Schritte durch. Zunächst wird in das lokale Git-Repository des Projekts navigiert und die aktuelle Version gepullt. Anschließend wird in das lokale Git-Repository des Bots navigiert und auch hier die aktuelle Version gepullt. Dies hat den Vorteil, dass mögliche Änderungen oder Erweiterungen des Bots bei jedem Start des Bots enthalten sind. Dabei ist aber zu beachten, dass in den Master-Branch des GitHub-Repositories vom Bot keine Änderungen gepusht werden dürfen, die dazu führen, dass der Bot nicht mehr korrekt läuft oder nicht mehr kompiliert werden kann. Als nächsten Schritt führt das Script für den Bot den Befehl „mvn install“ aus, wodurch das Bot-Projekt gebaut wird. Dieser Schritt wird benötigt, da der Code zunächst kompiliert werden muss, um den Bot starten zu können. Nachdem der Code kompiliert wurde, wird der Bot mithilfe des Befehls „mvn exec:java“ und dem Übergeben der vorher angelegten Properties-Datei gestartet. Beim ersten Start des Bots wird geschaut, ob bereits eine Textdatei, die die einzigartigen Schlüssel der bereits durchgeführten Refactorings enthält, an dem in der Properties-Datei angegebenen Ort existiert. Falls die Datei existiert, werden die Schlüssel in ein Array aus Strings eingelesen. Der Bot überprüft anhand dieser Liste, ob ein Refactoring bereits durchgeführt und vorgeschlagen wurde, und führt es kein zweites Mal durch. Sobald die maximale Anzahl an Pull-Requests erreicht wurde oder keine Befunde mehr übrig sind, die der Bot refactoren kann, wird die Liste in die Textdatei geschrieben und der Bot terminiert. Beim nächsten Start des Bots wird die geschriebene Liste wieder eingelesen.

Die Implementierung der Kernelemente des Bots ist damit abgeschlossen und der Bot ist bereit Refactorings durchzuführen. Er soll jetzt anhand eines Beispielprojekts getestet werden.

3.3 Testen des Bots

Beim Testen des Bots und auch bei der Durchführung einer Evaluation wird davon ausgegangen, dass der Code, der gerefactored werden soll, zur statischen Codeanalyse passt. Das heißt, seit der letzten Analyse wurden keine Änderungen am Code vorgenommen oder nach einem neuen Push auf den Master-Branch wurde eine neue Analyse durchgeführt. Das ist daher wichtig, da der Bot ansonsten beispielsweise versucht eine Annotation an eine Methode hinzuzufügen, die nach einem neuen Commit auf Master gar nicht mehr existiert. Der Befund der statischen Codeanalyse existiert aber noch, da die Analyse auf einem alten Stand des Codes basiert. Dieses Synchronisierungsproblem wurde in Unterabschnitt 3.2.2 bereits erwähnt und wird in Abschnitt 3.4 weiter diskutiert.

Nachdem der Bot implementiert wurde, wurde er anhand eines Beispielprojekts getestet. Das Beispielprojekt stellt dabei einen sehr vereinfachten Taschenrechner dar, der die Basisoperationen Multiplikation, Addition, Division und Subtraktion beherrscht. Es wurden absichtlich bestimmte Fehler eingebaut, sodass Sonarcloud bei der Analyse genau diese Stellen markiert. Die Fehler basieren dabei genau auf den Refactorings, die der Bot zum aktuellen Stand unterstützt. Auf folgende Kriterien wurde der Bot getestet:

- **Korrektheit:** Werden die richtigen Refactorings an den richtigen Stellen durchgeführt und ist das Ergebnis des Refactorings korrekt.
- **Zuverlässigkeit:** Führt der Bot die Refactorings durch, ohne dabei Fehler zu machen.
- **Einhalten der vorgegebenen Grenzen:** Erstellt der Bot nicht mehr als die vorgegebene maximale Anzahl an Pull-Request.

Der Bot wurde dabei mehrmals mit einer unterschiedlichen Anzahl an maximal offenen Pull-Requests getestet. Alle Refactorings wurden vom Bot ohne Fehler durchgeführt. Auch die Anzahl der Pull-Requests wurde stets eingehalten. Es gilt aber hier, wie bereits erwähnt, noch mal deutlich zu machen, dass das Beispielprojekt genau auf die Refactorings des Bots zugeschnitten ist und somit ist dieses Testergebnis mit Vorsicht zu genießen. Der Bot wurde außerdem auf einem weiteren Projekt getestet und dabei sind Probleme aufgetreten, die mit dem Beispielprojekt nicht zu erkennen waren. Diese Probleme werden im nächsten Kapitel diskutiert.

3.4 Diskussion offener Probleme

Wie bereits erwähnt wurden während des Testens und bei der Implementierung des Bots Probleme gefunden, die in diesem Kapitel diskutiert werden sollen. Eines der genannten Probleme ist die fehlerhafte Durchführung von zwei Refactorings. Zum einen „das Entfernen nicht verwendeter Methodenparameter“ und zum anderen „das Entfernen von nicht verwendeten Variablen“. Das Problem, das aufgetreten ist, besteht darin, dass eine Variable entfernt wurde, die aber im Konstruktor der Klasse verwendet wird. Sonarcloud hat diese Variable markiert und einen Befund erstellt, in dem empfohlen wird, diese Variable zu entfernen. Da der Bot als Input das Ergebnis der statischen Analyse bekommt und im Befund die Variable als zu entfernen markiert wurde, hat der Bot die Variable entfernt. Um dieses Problem zu umgehen, muss der Bot so erweitert werden, dass er den gesamten Code durchgeht, nach weiterem Auftreten dieser Variable sucht und diese nur dann entfernt, falls sie tatsächlich an keiner anderen Stelle verwendet wird. Falls sie an einer anderen Stelle verwendet wird, muss entschieden werden, ob die Variable beibehalten wird oder ob sie an allen vorkommenden Stellen im Code entfernt wird.

Das Problem beim Entfernen von nicht verwendeten Parametern von Methoden ist im Grunde dasselbe. Ein Methodenparameter könnte zwar in der Methode selbst nicht verwendet werden und somit von Sonarcloud markiert werden, aber der Parameter könnte an einer anderen Stelle in einer anderen Klasse vom Entwickler verwendet werden. In der Folge würden durch das Entfernen Compilefehler auftreten. Auch hier wäre die Lösung den Bot so zu erweitern, dass er den gesamten Code durchsucht und den Parameter an allen Stellen entfernt.

Während des Testens ist außerdem noch ein weiterer Punkt aufgefallen, der bei bestimmten Fällen negativ auffällt. Es handelt sich dabei um das Format des Programmcodes. Unter Format sind dabei beispielsweise Einzugsgröße oder Anzahl Spalten pro Codezeile vor einem Zeilenumbruch zu verstehen. JavaParser ändert das Format des Programmcodes, falls das Projekt ein eigenes Format besitzt. Das ist zum einen ein Problem, wenn die Entwickler ihr Format beibehalten wollen, und zum anderen wird der Pull-Request dadurch unübersichtlich, da sich durch das unterschiedliche Format alle Zeilen geändert haben und die tatsächlichen Änderungen durch die Refactorings nicht sofort sichtbar werden. Es existiert zu diesem Zweck eine Klasse mit dem Namen „LexicalPreservationPrinter“ für JavaParser, die dafür benutzt werden soll, um die Formatierung beizubehalten. Dies wurde getestet, führt aber wiederum zu Problemen beim Entfernen von Variablen. Es wird eine „UnsupportedOperationException“ geworfen, wenn der Typ der zu entfernenden Variable kein primitiver Datentyp ist. Zu diesem Problem wurde bereits auf der GitHub-Seite des JavaParser ein Issue erstellt, das zum aktuellen Zeitpunkt noch nicht gelöst wurde[11]. Sollte dieses

[11] https://github.com/javaparser/javaparser/issues/1667

Problem gelöst werden, kann für jede Refactoring-Klasse des Bots der „LexicalPreservationPrinter“ verwendet werden, um das Format beizubehalten. In Unterabschnitt 3.2.3 wurde beschrieben, dass das Umsortieren der Modifier in jedem anderen Refactoring automatisch enthalten ist. Auch diese Anomalie würde durch die Verwendung des „LexicalPreservationPrinters“ aufgehoben werden und lediglich das dafür extra erstellte Refactoring würde die Modifier umsortieren.

Wie in Unterabschnitt 3.2.2 erwähnt besteht ein Synchronisierungsproblem, das auftritt, wenn der Stand der Analyse und der Stand des Codes unterschiedlich sind. Dieses Problem könnte umgangen werden, indem der Bot in die Build-Pipeline des Projekts, auf dem er arbeiten soll, integriert wird. Die Idee ist es, in dieser Pipeline einen Job zu erstellen, bei dem die statische Analyse durchgeführt wird, bevor der Bot ausgeführt wird. Sind die Ergebnisse der Analyse dabei öffentlich auf Sonarcloud, kann der Bot ohne Probleme darauf zugreifen. Ist die Analyse dagegen nicht öffentlich, benötigt der Bot einen Authentifizierungstoken, um ihm das Recht zu geben, die Ergebnisse der Analyse abzufragen. Die Konfiguration des Bots müsste in diesem Fall um diesen Parameter erweitert und die Methode, die für das Abfragen der Ergebnisse zuständig ist, angepasst werden.

Eine andere Option wäre die Analyse selbst durchzuführen und dabei entweder SonarQube lokal zu benutzen oder eine eigene Sonarcloud-Organisation zu erstellen, in der der Bot die Analyse für alle seine Projekte durchführt. Die letztere Möglichkeit setzt voraus, dass die Entwickler erlauben die Analyse ihres Codes öffentlich zugänglich zu machen. Den Bot die Analyse selbst durchführen zu lassen hat allerdings den Nachteil, dass von vornherein klar sein muss, welches Build-Management Tool für das Projekt verwendet wird. Je nachdem, welches Tool verwendet wird, wird ein anderer Befehl zum Starten der Analyse benötigt. Das ausschlaggebende Kriterium gegen diese Option ist aber, das die Entwickler in diesem Fall nicht die Möglichkeit haben festzulegen, welche Regeln bei der Analyse durch SonarQube berücksichtigt werden sollen. Auch die Möglichkeit eigene Regeln zu definieren ist dadurch nicht mehr gegeben. Aus den genannten Gründen ist die erste Option die sinnvollere.

4 Evaluationskonzepte

Dieses Kapitel beschreibt die theoretische Durchführung der Evaluation auf zwei verschiedene Arten. Aufgrund des zeitlich begrenzten Rahmens der Arbeit werden ausschließlich Evaluationskonzepte aufgestellt und die Evaluation selbst nicht durchgeführt. Ein weiterer Grund hierfür ist, dass es sich bei dem Bot noch um einen Prototypen handelt und der Bot erweitert werden muss, um sinnvoll evaluiert zu werden. Bevor die beiden Evaluationskonzepte vorgestellt werden, wird zunächst beschrieben, welche Erweiterungen nötig sind, dass der Bot einen evaluationsfähigen Stand erreicht.

4.1 Diskussion des evaluationsfähigen Stands

Wie zuvor erwähnt, befindet sich der Bot in einem Zustand, in dem er noch nicht als evaluationsfähig einzustufen ist. Im Folgenden werden die beiden wichtigsten Kriterien genannt, die der Bot mindestens erfüllen sollte, um eine aussagekräftige Evaluation durchzuführen.

Anzahl durchführbarer Refactorings Die Anzahl der durchführbaren Refactorings ist mit zwei, beziehungsweise vier, wenn die Refactorings gezählt werden, die fehlerhaft durchgeführt werden, noch zu gering. Dementsprechend würden bei einem Großteil der Projekte eine geringe Anzahl oder keine Refactorings durchgeführt werden, was für eine aussagekräftige Evaluation nicht ausreicht. Eine Mindestanzahl von zehn durchführbaren Refactorings wird empfohlen, um den Bot unter diesem Kriterium als evaluationsfähig einzustufen. Diese Zehn sollen dabei aber Refactorings sein, die in der Liste der am häufigsten auftretenden Befunde auf Sonarcloud zumindest unter den ersten dreißig liegen. Dadurch ist gewährleistet, dass der Bot in den meisten Projekten eine aussagekräftige Anzahl an Befunden refactoren kann.

Format Das Problem der Formatierung des Codes wurde bereits in Abschnitt 3.4 erläutert. Für eine sinnvolle Evaluierung ist es wichtig, dieses Problem zu lösen. Wenn ein Refactoring das Format ändert, wird der Pull-Request mit hoher Wahrscheinlichkeit abgelehnt. Dem zugrunde liegt nicht nur, dass die gemachten Änderungen unübersichtlich im Pull-Request dargestellt werden, sondern dass die Entwickler ihr Format beibehalten wollen. Durch das Akzeptieren des Pull-Requests würden aber einzelne Java-Klassen entstehen, die eine Abweichung vom festgelegten Format darstellen. Die gemachten Änderungen würden also aufgrund der Formatierung abgelehnt werden, auch wenn die Refactorings korrekt durchgeführt wurden. Dies hat zur Folge, dass die Ergebnisse einer Evaluation verfälscht werden würden.

4.2 Evaluation der Quantität

Im Folgenden soll ein Evaluationskonzept vorgestellt werden, mit dem der Bot auf einer Vielzahl von Projekten evaluiert werden kann. Idee dieser Evaluationsvariante ist es, den Bot selbstständig anhand von bestehenden Softwareprojekten zu testen. Es sollen dabei zwei Forschungsfragen untersucht werden:

- **Forschungsfrage 1:** Ist der Bot in der Lage, einen Mehrwert für ein Projekt darzustellen? Mehrwert bedeutet in diesem Kontext, dass der Bot in einem realen Projekt den Arbeitsaufwand für das Entwicklerteam reduzieren kann und somit Ressourcen eingespart werden.
- **Forschungsfrage 2:** Werden die durchgeführten Refactorings von den Entwicklern akzeptiert?

4.2.1 Vorbereitung der Evaluation

Bevor die Evaluation durchgeführt werden kann, müssen Vorbereitungen getroffen werden. Dazu gehört das Festlegen der Kriterien, also die Metrik nach denen die Evaluation ausgewertet werden kann und die Suche nach geeigneten Projekten für die Evaluation. Außerdem muss der Einsatz des Bots und die Auswertung vorbereitet werden.

Evaluationskriterien

Zunächst müssen Kriterien festgelegt werden, die untersucht werden sollen:

- **Reduzierung der technischen Schuld:** Der Begriff technische Schuld wurde von Ward Cunningham [Cun92] eingeführt. Er beschreibt, dass unausgereifter Code dem Aufnehmen von Schulden ähnelt. Die Schulden werden durch Überarbeitung des Codes zurückgezahlt. Bei dieser Metrik soll untersucht werden, ob die Zeit, die benötigt wird, um die technische Schuld auszugleichen, durch den Einsatz des Bots reduziert wird.
- **Anzahl der akzeptierten Pull-Requests:** Hier soll untersucht werden, wie viele der vom Bot vorgeschlagenen Pull-Request akzeptiert werden.

Nachdem die Kriterien, die zu untersuchen sind, festgelegt sind, werden Projekte gesucht, auf denen der Bot evaluiert werden kann.

Kriterien bei der Auswahl der Projekte

Nicht jedes Softwareprojekt ist für die Evaluierung des Bots geeignet. Da in dieser Evaluation geplant ist den Bot selbst auf den Projekten zu testen, beschränkt sich die Auswahl auf Open-Source-Projekte. Der Grund dafür ist, dass der Bot Zugriff auf das Repository und die Ergebnisse der statischen Codeanalyse benötigt. Bei proprietären Softwareprojekten sind diese beiden Elemente meistens privat und nicht für den Bot zugänglich. Somit sind diese Projekte für diese Evaluation nicht geeignet. Kriterien, die die Projekte erfüllen müssen, da der Bot ansonsten keine Refactorings durchführen kann, lauten dementsprechend wie folgt:

- Der Code des Projekts liegt auf einem öffentlichen GitHub-Repository.

- Das Projekt verwendet SonarQube für die statische Codeanalyse und die Ergebnisse der Analyse sind auf Sonarcloud öffentlich verfügbar.
- Die Ergebnisse der statischen Codeanalyse müssen zum aktuellen Stand des Codes auf dem Master-Branch passen.
- Das Projekt muss in der Programmiersprache Java geschrieben sein.
- An dem Projekt muss noch gearbeitet werden. Wenn die Bearbeitung eingestellt wurde, ist davon auszugehen, dass erstellte Pull-Requests ignoriert werden.

Wenn ein Projekt alle genannten Kriterien erfüllt, ist es für die quantitative Evaluation geeignet. Bei der Auswahl sollte neben den Kriterien, die erfüllt sein müssen, darauf geachtet werden welche Größe die Projekte haben. In diesem Kontext ist unter der Größe die Anzahl der Codezeilen und die Anzahl der offenen Befunde zu verstehen. Häufig hängen diese beiden Faktoren miteinander zusammen. Je mehr Codezeilen, desto höher ist die Wahrscheinlichkeit eine größere Anzahl an Befunden vorzufinden [Lip82]. Wünschenswert wären Projekte mit 50 oder mehr Befunden. Eine Evaluation auf einem solchen Projekt ist aussagekräftiger und realitätsnäher als auf einem Projekt mit 5 oder weniger Befunden. Dabei ist zu beachten, dass nicht vorgesehen ist speziell Projekte zu finden, die nur Befunde haben, die der Bot refactoren kann, da dieser Ansatz das Ergebnis verfälschen würde.

Suche nach Projekten

Auf Basis der im letzten Abschnitt definierten Auswahlkriterien sollen Projekte gesucht werden, die diese erfüllen. Das Vorgehen bei der Suche soll in diesem Abschnitt erklärt werden.

Da eines der Auswahlkriterien ein vorhandenes GitHub-Repository ist, wurde zunächst versucht, die Suche auf GitHub durchzuführen. Da aber unter Eingabe verschiedener Suchbegriffe keine Projekte gefunden wurden, die ihre statische Analyse mittels Sonarcloud durchführen, wurde die Suchstrategie geändert. Statt nach Projekten auf GitHub zu suchen, wurde stattdessen auf der Sonarcloud-Seite nach Projekten gesucht. Dabei wurde so vorgegangen, dass als Filter zunächst Java als Programmiersprache ausgewählt wurde und die Größe der Projekte auf 1000 oder mehr Codezeilen festgelegt wurde. Anschließend wurden sowohl die Namen der Projekte, als auch die Namen der Organisationen in die GitHub-Suche kopiert. Dabei wurden Projekte gefunden. Dieser Weg zum Finden von passenden Projekten wird somit empfohlen. Um eine aussagekräftige Evaluation durchführen zu können, sollten dabei mindestens 20 Projekte gefunden werden.

Vorbereitung des Bots

Um die Evaluierung durchzuführen, muss der Bot für alle gefundenen Projekte gestartet werden. Zunächst müssen die Repositories von GitHub geklont werden, sodass sie lokal im Dateisystem vorhanden sind. Auch der Code des Bots selbst wird aus dem GitHub-Repository geklont. Zuletzt wird noch das Script zum Starten des Bots heruntergeladen. Um den Bot starten zu können, muss wie in Unterabschnitt 3.2.5 beschrieben, eine Properties-Datei für jedes Projekt erstellt werden. Die dazu benötigten Parameter werden sowohl aus dem GitHub-Repository, als auch aus dem Sonarcloud-Projekt extrahiert und in die Poperties-Datei übernommen. Die Pfade werden dabei

über das lokale Dateisystem gewonnen. Sind diese Vorkehrungen getroffen worden, ist der Bot bereit gestartet zu werden. Der letzte Schritt, der nötig ist, um die Vorbereitung abzuschließen, ist die Vorbereitung der Auswertung.

Vorbereitung der Auswertung

Um die Ergebnisse später auswerten zu können, wird eine Tabelle erstellt, in der die nachfolgend aufgelisteten Werte dokumentiert werden. Jedes Projekt bekommt in dieser Tabelle eine eigene Zeile. Die Tabelle hat folgende Spalten:

- Name des Projekts
- Anzahl erstellter Pull-Requests
- Anzahl angenommener Pull-Requests
- Anzahl abgelehnter Pull-Requests
- Technische Schuld vor dem Refacoring
- Technische Schuld nach dem Refactoring

Die momentane technische Schuld kann bereits vor der Durchführung notiert werden. Der aktuelle Wert ist über die Sonarcloud-Seite des Projekts zu finden. Hier ist die technische Schuld für das Projekt in Stunden oder Tagen angegeben. In Sonarcloud kann außerdem der Verlauf der technischen Schuld betrachtet werden. Es sollte aber darauf geachtet werden, dass die Durchführung zeitnah geschieht, da es ansonsten vorkommen kann, dass bereits eine neue Analyse durchgeführt wurde und der notierte Wert somit nicht mehr aktuell ist. Über den Verlauf kann herausgefunden werden, ob sich die technische Schuld vor der Durchführung der Evaluation im Vergleich zum bereits notierten Wert verändert hat. Alle weiteren Werte werden während der Durchführung in der Tabelle notiert. Die Vorbereitung ist abgeschlossen und die Evaluation kann durchgeführt werden.

4.2.2 Durchführung der Evaluation

Für die Durchführung der Evaluation wird nacheinander für jedes Projekt der Bot gestartet. Der Bot führt jetzt seine Refactorings durch und erstellt die jeweiligen Pull-Requests. Anschließend wird die Zahl der erstellen Pull-Requests durch den Bot für jedes Projekt in der vorher erstellten Tabelle notiert. Nach einer gewissen Zeitspanne, die nötig ist, um den Entwicklern die Zeit zu geben den Pull-Request anzuschauen und zu akzeptieren oder abzulehnen, wird geschaut, wie viele Pull-Requests für die einzelnen Projekte akzeptiert wurden und wie viele abgelehnt wurden. Auch diese Werte werden in die Tabelle übernommen. Es wird außerdem überprüft, ob für die jeweiligen Projekte bereits eine neue statische Codeanalyse durchgeführt wurde. Sobald dies geschehen ist, kann auch hier der neue Wert für die technische Schuld in der Tabelle notiert werden.

Nachdem diese Prozedur mit allen zu testenden Projekten durchgeführt wurde, werden die Ergebnisse ausgewertet und analysiert.

4.2.3 Auswertung und Interpretation der Evaluationsergebnisse

Um die Ergebnisse auszuwerten, werden zunächst die notierten Werte in der Tabelle verrechnet. Dazu wird für jedes Projekt der Prozentsatz der angenommenen Pull-Requests auf Basis der vorhandenen Werte in der Tabelle berechnet. Außerdem wird die Differenz der technischen Schuld in Prozent berechnet. Aus den Ergebnissen der einzelnen Projekte wird zusätzlich der Prozentsatz der akzeptierten Pull-Requests und die Reduzierung der technischen Schuld über alle Projekte bestimmt und der Durchschnitt berechnet. Es ist dann von einem hohen Prozentsatz akzeptierter Pull-Requests zu sprechen, wenn dieser größer als 60% beträgt. Liegt er unter 60%, ist der Prozentsatz gering. Für die technische Schuld wird bereits bei einer Reduzierung um 20% oder mehr von einem hohen Prozentsatz gesprochen. Grund dafür ist, dass bei Projekten mit einer großen technischen Schuld eine Reduzierung um beispielsweise 60% sehr unwahrscheinlich ist. Um diesen Fall zu erreichen, müsste der Bot eine große Anzahl an Befunden refactoren und bei diesen Befunden müsste es sich fast ausschließlich um Befunde handeln, die von Sonarcloud mit einer großen technischen Schuld markiert wurden. Daher sind 20% oder mehr bereits ein hoher Prozentsatz. Unter 20% ist von einem geringen Prozensatz zu sprechen.

Interpretation der Extremfälle

Sollte der Fall eintreten, dass keine Pull-Requests für ein Projekt erstellt wurden, muss der Grund hierfür gefunden werden. Mögliche Gründe hierfür können zum einen ein Fehler des Bots während der Refactorings sein oder für das Projekt existieren keine Befunde, die der Bot refactoren kann.

In Fällen, in denen Pull-Requests erstellt wurden, aber der Prozentsatz der akzeptierten Pull-Requests bei 0% liegt hat der Bot automatisch auch keinen Einfluss auf die technische Schuld. Denn wenn die Pull-Requests nicht akzeptiert werden, werden die gemachten Änderungen nicht in das Repository übernommen und somit bei einer erneuten Analyse nicht berücksichtigt. In diesem Fall muss hinterfragt werden, was der Grund für die Ablehnung der Pull-Requests war. Ein möglicher Grund könnte eine fehlerhafte Durchführung der Refactorings sein. Um die Ursache zu finden, sollten in diesem Extremfall die erstellten Pull-Requests angeschaut werden und untersucht werden ob beispielsweise im Kommentarfeld der Pull-Requests erwähnt wurde, was der Grund der Ablehnung war.

Forschungsfrage 1

Um eine Antwort auf die erste Forschungsfrage geben zu können, ob der Bot einen Mehrwert darstellen kann, indem er Ressourcen für die Entwickler einspart, wird die Reduzierung der technischen Schuld untersucht. Im Falle einer hohen Reduzierung kann die Forschungsfrage mit „Ja“ beantwortet werden. Sollte dagegen die Reduzierung gering ausfallen, ist die Forschungsfrage mit „Nein“ zu beantworten.

Forschungsfrage 2

Um eine Antwort auf die zweite Forschungsfrage geben zu können, ob die durchgeführten Refactorings von den Entwicklern akzeptiert werden, wird von den Ergebnissen der Prozentsatz der angenommenen Pull-Requests verwendet. Ist dieser Prozentsatz hoch, kann davon ausgegangen werden, dass die Entwickler die durchgeführten Refactorings als korrekt und sinnvoll eingestuft haben. Somit kann in diesem Fall die Forschungsfrage mit „Ja“ beantwortet werden. Ist der Prozentsatz dagegen tief, ist die Forschungsfrage mit „Nein“ zu beantworten.

Kombination beider Metriken

Wenn die beiden Metriken miteinander kombiniert werden, entstehen dabei vier mögliche Fälle:

- **Fall 1:** Hoher Prozentsatz akzeptierter Pull-Requests und hohe Reduzierung der technischen Schuld.
- **Fall 2:** Hoher Prozentsatz akzeptierter Pull-Requests aber geringe Reduzierung der technischen Schuld
- **Fall 3:** Geringer Prozentsatz akzeptierter Pull-Requests und geringe Reduzierung der technischen Schuld
- **Fall 4:** Geringer Prozentsatz akzeptierter Pull-Requests aber hohe Reduzierung der technischen Schuld

Fall 1 stellt dabei den Idealfall dar, Fall 3 das Gegenteil. Der Prozentsatz der akzeptierten Pull-Requests steht dabei in direktem Verhältnis zur Reduzierung der technischen Schuld. Werden viele Pull-Requests angenommen ist die Reduzierung dementsprechend hoch und umgekehrt. Fall 2 und Fall 4 sind unterschiedlich zu interpretieren. Fall 2 kann so interpretiert werden, dass die Refactorings zwar korrekt durchgeführt wurden, aber dabei wurden überwiegend Befunde abgearbeitet, die von SonarQube mit einem geringen Zeitaufwand eingestuft wurden. Hier sollten die implementierten Refactorings nochmals überprüft werden und überlegt werden, ob es Sinn ergibt, weitere komplexere Refactorings hinzuzufügen. Fall 4 ist das Gegenstück zu Fall 2. Die akzeptierten Änderungen haben Befunde abgearbeitet, die mit einem hohen Zeitaufwand eingestuft wurden. Zu beachten ist hier, dass dennoch ein Großteil der Refactorings abgelehnt wurde. In diesem Fall gilt zu untersuchen, was der Grund für die Ablehnung war.

4.3 Evaluation der Qualität

Im Vergleich zum ersten Konzept, wird in diesem Abschnitt ein Evaluationskonzept beschrieben, mit dem der Bot unter Hinzuziehung von Probanden evaluiert werden soll. Idee hier ist es nicht, den Bot selbst auf bestehenden Projekten arbeiten zu lassen, sondern Entwickler oder ganze Entwicklerteams den Bot auf ihren Projekten testen zu lassen. Diese sollen anschließend Feedback geben. Folgende Forschungsfrage soll dabei beantwortet werden: Stellt der Bot einen Störfaktor während der Entwicklung von Software dar?

4.3.1 Vorbereitung der Evaluation

Um diese Evaluation vorzubereiten, müssen auch hier Kriterien festgelegt werden, die untersucht werden, um die Forschungsfrage zu beantworten. Das Suchen von Probanden und die Erstellung eines Bewertungsbogens für das Feedback der Probanden sind ebenfalls ein Bestandteil der Vorbereitung.

Evaluationskriterien

Auch bei dieser Variante müssen zunächst Kriterien festgelegt werden nach denen der Bot evaluiert wird:

- **Gebrauchstauglichkeit:** Wie gut kommen die Probanden mit der Bedienung des Bots klar? Dieses Kriterium ist an den Unterpunkt „Operability“ der Charakteristik „Usability“ des Qualitätsmodels in ISO 25010 angelehnt [ISO11].
- **Nützlichkeit:** Empfinden die Entwickler den Bot als nützlich oder störend?

Suchen nach Probanden

Zur Vorbereitung gehört das Suchen nach geeigneten Probanden. Zielgruppe sollen dabei Studierende sein, die bereits Programmiererfahrung haben. Dazu eignen sich vor allem Studierende, die bereits in fortgeschrittenen Semestern sind und gerade an einem Studienprojekt teilnehmen. Um die Probanden zu finden, werden in den Rechner-Pools der Universität die Studierenden gefragt, ob sie im Moment an einem Studienprojekt teilnehmen. Idealfall wäre, wenn die Software der Studienprojekte selbst bereits in einem GitHub-Repository liegt. Falls dies nicht der Fall ist, soll gefragt werden, ob die Möglichkeit besteht, dass die Studierenden ihr Studienprojekt auf ein GitHub-Repository ablegen können und eine statische Analyse auf Sonarcloud durchführen können. Falls es nicht möglich ist, die Software der Studienprojekte zu verwenden, können die Studierenden alternativ nach einem eigenen Projekt, an dem sie gerade arbeiten und für diese Evaluation verwenden können, gefragt werden. Dabei besteht im Vergleich zum ersten Evaluationskonzept kein Zwang zu einem öffentlichen Repository. Wichtig ist nur zu beachten, dass die Software in der Programmiersprache Java geschrieben ist. Ziel ist es, mindestens 10 Probanden zu finden.

Erstellung des Bewertungsbogens

Für das Feedback der Probanden sollen diese, nachdem sie den Bot getestet haben, mittels eines Interviews zu ihrer Meinung befragt werden. Zu diesem Zweck soll ein Bewertungsbogen erstellt werden, der auf einer Likert-Skala basiert. Diese wurde von Rensis Likert 1932 entwickelt [Lik32]. Dabei werden Aussagen aufgestellt, die die Probanden mit einer Zahl zwischen 1 und 4 bewerten sollen. 1 steht dabei für „Trifft zu“ und 4 für „Trifft nicht zu“. Es wird bewusst eine gerade Zahl gewählt, sodass es nicht möglich ist, neutral auf eine der Aussagen zu antworten. Die Probanden haben also nur wertende Antwortmöglichkeiten. Beispiele für Aussagen auf diesem Bogen sind:

- Die Bedienung des Bots war einfach.

- Das Ausführen des Bots hat mich bei meiner Arbeit abgelenkt.
- Die erstellten Pull-Requests waren verständlich.
- Die Überprüfung der Pull-Requests war störend.

Zusätzlich zu diesen Aussagen, sollen im Interview weitere Fragen gestellt werden, die die Probanden frei, also nicht mittels einer Bewertungsskala, beantworten sollen. Mögliche Fragen sind folgende:

- Würden Sie den Bot weiter benutzen?
- Lohnt sich Ihrer Meinung nach der Mehraufwand des Reviews der Pull-Requests?
- Hatten Sie das Gefühl durch den Bot in ihrem normalen Workflow gestört worden zu sein?
- Haben Sie weitere Anmerkungen oder Verbesserungsvorschläge?

Die Vorbereitung der Evaluation ist damit abgeschlossen und kann durchgeführt werden.

4.3.2 Durchführung der Evaluation

Für die Durchführung werden die Probanden eingeladen und bekommen eine kurze Einführung. Zur Einführung gehört eine Erklärung, wie der Bot für ein Projekt konfiguriert wird und anschließend gestartet werden kann. In Unterabschnitt 3.2.5 wurde bereits beschrieben, welche Schritte dafür nötig sind. Nach der Einweisung der Probanden sollen diese den Bot für eine Woche benutzen, während sie an ihren Projekten weiterarbeiten. Nach Ablauf dieser Woche werden die Probanden ein weiteres Mal eingeladen, um in einem Interview den in der Vorbereitung erstellten Bewertungsbogen auszufüllen und die darin enthaltenen Fragen zu beantworten. Sobald die Interviews mit allen Probanden durchgeführt wurden, ist die Durchführung der Evaluation beendet. Die Bewertungsbögen werden im Anschluss an die Durchführung ausgewertet.

4.3.3 Auswertung und Interpretation der Evaluationsergebnisse

Die Bewertungsbögen werden ausgewertet, indem für jede Aussage die durchschnittliche Antwortpunktzahl berechnet wird. Auch die Antworten auf die Fragen mit freier Antwortmöglichkeit werden ausgewertet, indem geschaut wird, ob sie positiv oder negativ für den Bot ausfallen. Je nach Ergebnis dieser Auswertung kann die Forschungsfrage unterschiedlich beantwortet werden. Bewerten die Probanden den Bot über alle Aussagen überwiegend als störend, ist die Forschungsfrage mit „Ja“ zu beantworten. Tritt dieser Fall ein, ist das weitere Vorgehen, den Grund für diese Einschätzung zu finden. Über die einzelnen Aussagen kann analysiert werden, welche Elemente des Bots ein Störfaktor sind. Auch die Antworten auf die gestellten Fragen können Aufschluss darüber geben und sollten berücksichtigt werden. Beispielsweise könnten die Pull-Requests für die Probanden störend sein. Als Folge sollten die Elemente, die von den Probanden negativ oder als Störfaktor bewertet wurden, noch mal überdacht und dementsprechend angepasst werden.

Sind die Bewertungen dagegen zugunsten des Bots, ist die Forschungsfrage mit „Nein“ zu beantworten. In diesem Fall stellt der Bot keinen Störfaktor dar und es ist anzunehmen, dass die Bedienung des Bots verständlich ist. Aber auch diesem Fall kann, durch das Untersuchen der

Antworten auf die Fragen, geschaut werden, ob die Probanden Verbesserungsvorschläge beigetragen haben, die sinnvoll und umsetzbar sind. Diese können in der weiteren Entwicklung des Bots berücksichtigt und umgesetzt werden.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde beschrieben, wie ein Prototyp eines Bots, der vollautomatisch Refactorings durchführt, implementiert wurde. Der Bot ist in der Lage auf Basis einer statischen Codeanalyse Code zu refactoren und die Änderungen mittels eines Pull-Requests dem Entwickler vorzuschlagen. Die geforderte Grundfunktionalität ist somit gegeben. Bisher unterstützt er dabei zwei Refactorings komplett und für zwei weitere Refactorings wurde der Grundstein gelegt. Um diese als einsatzfähig zu bewerten, bedarf es aber noch einer Erweiterung. Während des Testens des Bots sind Probleme festgestellt worden, die während der Implementierung nicht bedacht wurden. Dazu gehören zum einen die nicht bedachten Aspekte der nicht komplett unterstützen Refactorings und zum anderen das Problem des Formats. Für die genannten Probleme wurden zudem mögliche Lösungen diskutiert, die in der weiteren Entwicklung des Bots berücksichtigt werden können. Des Weiteren wurden Evaluationskonzepte beschrieben, auf deren Basis eine Evaluation durchgeführt werden kann, sobald der Bot einen evaluationsfähigen Stand erreicht hat.

Ausblick

Wie die Arbeit zeigt, ist es möglich, einen solchen Bot zu implementieren. Da es sich zum aktuellen Stand aber eher um einen Prototyp handelt, muss der Bot erweitert werden. In erster Linie müssen die genannten Probleme behoben werden. Dazu gehört zum einen das Beibehalten des ursprünglichen Formats und zum anderen das Fertigstellen der beiden Refactorings „Entfernen einer nicht verwendeten Variable“ und „Entfernen eines nicht verwendeten Methodenparameters“. Mögliche Lösungen für diese Probleme wurden bereits in Abschnitt 3.4 diskutiert.

Um weitere Probleme dieser Art vorzubeugen oder im Allgemeinen zu verhindern, dass die durchgeführten Refactorings zu Compilefehlern führen, besteht die Möglichkeit zwischen der Durchführung der Refactorings und dem Erstellen des Pull-Requests einen weiteren Schritt einzubauen. Dieser Schritt soll sicherstellen, dass keine Compilefehler durch die Änderungen verursacht werden. Eine Möglichkeit wäre, dass der Bot in diesem Schritt schaut, ob für das Projekt Unit-Tests erstellt wurden. Findet er Unit-Tests, werden diese ausgeführt und die Testergebnisse werden in der Beschreibung des Pull-Requests bereitgestellt. So kann der Entwickler, bevor er den Pull-Request akzeptiert, sehen, ob die Tests fehlgeschlagen sind und überprüfen, ob der Grund dafür die Änderungen durch den Bot sind oder ob die Tests eventuell davor schon fehlgeschlagen sind. Sollten keine Unit-Tests vorhanden sein, wäre die andere Option, dass der Bot versucht, das Projekt mit den gemachten Änderungen zu bauen. Sollte dabei festgestellt werden, dass der Build fehlschlägt, werden die Änderungen verworfen.

Neben dem Lösen der bestehenden Probleme gilt es, für den Bot weitere Refactorings zu implementieren. Hierbei sollten Refactorings hinzugefügt werden, die häufig auftreten und vollautomatisch realisierbar sind. Zum Finden solcher Refactorings kann die Liste der am häufigsten auftretenden Befunde bei Sonarcloud durchgeschaut werden. Beispiele hierfür sind:

- Variablen innerhalb von Klassen, die als „public“ deklariert worden sind, sollten entweder „private“ , „protected“ oder eine „static final“ Konstante sein.
- Felder, die mit „public static“ deklariert wurden, sollen zusätzlich mit „final“ deklariert werden.
- Häufig auftretende gleiche Literale sollten als Konstanten implementiert werden.

Neben diesen trivialen Refactorings wäre es außerdem interessant den Bot um komplexe Refactorings zu erweitern. Ein Beispiel hierfür wäre das Extrahieren einer Methode. Bei einem solchen Refactoring ist allerdings zu beachten, dass es nicht trivial ist, dieses vollautomatisch durchführen zu lassen. Der Grund dafür ist, dass die neue Methode einen Namen benötigt. Man müsste sich also zunächst überlegen, wie man diesen festlegt. Mögliche Ansätze wären ein generischer Namen, der später noch vom Entwickler angepasst werden kann oder den Bot so zu erweitern, dass er Issues von GitHub ausliest, in denen die Entwickler den Namen definieren.

Nachdem der Bot, wie zuvor beschrieben, einen evaluationsfähigen Stand erreicht hat, sollte er nach den vorgestellten Evaluierungskonzepten evaluiert werden, um festzustellen, ob er einen Mehrwert darstellt.

Zum aktuellen Stand ist der Bot lediglich lokal ausführbar. In Zukunft könnte der Bot auf einem Server laufen. Zusätzlich könnte eine grafische Oberfläche implementiert werden. Diese kann unter anderem eine Konfigurationsseite anbieten, auf der die Entwickler alle Parameter festlegen können, die momentan in der Properties-Datei festgelegt werden. Die Konfiguration selbst könnte dabei ebenfalls erweitert werden, sodass die Entwickler in der Lage sind, auszuwählen, welche Refactorings für ihr Projekt durchgeführt werden sollen und welche nicht.

Literaturverzeichnis

[Cun92] W. Cunningham. „The WyCash Portfolio Management System“. In: *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*. OOPSLA '92. Vancouver, British Columbia, Canada: ACM, 1992, S. 29–30. ISBN: 0-89791-610-7. DOI: 10.1145/157709.157715. URL: http://doi.acm.org/10.1145/157709.157715 (zitiert auf S. 30).

[FB99] M. Fowler, K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999 (zitiert auf S. 11, 14, 15).

[FGL12] S. R. Foster, W. G. Griswold, S. Lerner. „WitchDoctor: IDE support for real-time auto-completion of refactorings“. In: *2012 34th International Conference on Software Engineering (ICSE)*. Juni 2012, S. 222–232. DOI: 10.1109/ICSE.2012.6227191 (zitiert auf S. 16).

[GDM12] X. Ge, Q. L. DuBose, E. Murphy-Hill. „Reconciling Manual and Automatic Refactoring“. In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012, S. 211–221. ISBN: 978-1-4673-1067-3. URL: http://dl.acm.org/citation.cfm?id=2337223.2337249 (zitiert auf S. 11).

[Git18] GitHub, Inc. 2018. URL: https://github.com/Refactoring-Bot/RefactoringTest/pull/89/files (zitiert auf S. 15).

[GM11] X. Ge, E. Murphy-Hill. „BeneFactor: a flexible refactoring tool for eclipse“. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM. 2011, S. 19–20 (zitiert auf S. 16).

[GO17] Y. Gil, M. Orrù. „The Spartanizer: Massive automatic refactoring“. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Feb. 2017, S. 477–481. DOI: 10.1109/SANER.2017.7884657 (zitiert auf S. 16).

[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. 16).

[GXWW11] S. Gianvecchio, M. Xie, Z. Wu, H. Wang. „Humans and Bots in Internet Chat: Measurement, Analysis, and Automated Classification“. In: *IEEE/ACM Transactions on Networking* 19.5 (Okt. 2011), S. 1557–1571. ISSN: 1063-6692. DOI: 10.1109/TNET.2011.2126591 (zitiert auf S. 15).

[ISO11] ISO/IEC JTC 1/SC 7. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. Englisch. Standard ISO/IEC 25010:2011(en). International Organization for Standardization, März 2011. URL: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en (zitiert auf S. 35).

[Lik32] R. Likert. „A technique for the measurement of attitudes.“ In: *Archives of psychology* (1932) (zitiert auf S. 35).

[Lip82] M. Lipow. „Number of Faults per Line of Code“. In: *IEEE Transactions on Software Engineering* SE-8.4 (Juli 1982), S. 437–439. ISSN: 0098-5589. DOI: 10.1109/TSE.1982.235579 (zitiert auf S. 31).

[Lou06] P. Louridas. „Static code analysis“. In: *IEEE Software* 23.4 (2006), S. 58–61 (zitiert auf S. 13).

[LSZ18] C. Lebeuf, M. A. Storey, A. Zagalsky. „Software Bots“. In: *IEEE Software* 35.1 (Jan. 2018), S. 18–23. ISSN: 0740-7459. DOI: 10.1109/MS.2017.4541027 (zitiert auf S. 15).

[MÓ11] I. H. Moghadam, M. Ó Cinnéide. „Code-Imp: A tool for automated search-based refactoring“. In: *Proceedings of the 4th Workshop on Refactoring Tools*. ACM. 2011, S. 41–44 (zitiert auf S. 16).

[MPB12] E. Murphy-Hill, C. Parnin, A. P. Black. „How We Refactor, and How We Know It“. In: *IEEE Transactions on Software Engineering* 38.1 (Jan. 2012), S. 5–18. ISSN: 0098-5589. DOI: 10.1109/TSE.2011.41 (zitiert auf S. 11).

[MT04] T. Mens, T. Tourwé. „A survey of software refactoring“. In: *IEEE Transactions on software engineering* 30.2 (2004), S. 126–139 (zitiert auf S. 14).

[NK+10] J. Novak, A. Krajnc et al. „Taxonomy of static code analysis tools“. In: *MIPRO, 2010 Proceedings of the 33rd International Convention*. IEEE. 2010, S. 418–422 (zitiert auf S. 13).

[PK13] G. H. Pinto, F. Kamei. „What Programmers Say About Refactoring Tools?: An Empirical Investigation of Stack Overflow“. In: *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools*. WRT '13. Indianapolis, Indiana, USA: ACM, 2013, S. 33–36. ISBN: 978-1-4503-2604-9. DOI: 10.1145/2541348.2541357. URL: http://doi.acm.org/10.1145/2541348.2541357 (zitiert auf S. 11).

[SNF+15] G. Szőke, C. Nagy, L. J. Fülöp, R. Ferenc, T. Gyimóthy. „FaultBuster: An automatic code smell refactoring toolset“. In: *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Sep. 2015, S. 253–258. DOI: 10.1109/SCAM.2015.7335422 (zitiert auf S. 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*. FSE 2016. Seattle, WA, USA: ACM, 2016, S. 858–870. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2950305. URL: http://doi.acm.org/10.1145/2950290.2950305 (zitiert auf S. 14, 15).

[Sző17] G. Szőkea. „Automating the Refactoring Process“. In: *Acta Cybernetica* 23 (2017), S. 715–735 (zitiert auf S. 17).

[Ter15] J. Terrell. „KinEdit: A Tool to Help Developers Refactor Manually“. In: *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. SPLASH Companion 2015. Pittsburgh, PA, USA: ACM, 2015, S. 71–72. ISBN: 978-1-4503-3722-9. DOI: 10.1145/2814189.2815366. URL: http://doi.acm.org/10.1145/2814189.2815366 (zitiert auf S. 17).

[ZSR14] Z. Zhioua, S. Short, Y. Roudier. „Static Code Analysis for Software Security Verification: Problems and Approaches“. In: *2014 IEEE 38th International Computer Software and Applications Conference Workshops* (2014), S. 102–109 (zitiert auf S. 13).

Alle URLs wurden zuletzt am 03. 10. 2018 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