Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Implementierung eines automatischen Refactorings zum Entfernen auskommentierten Quellcodes

Justin Kißling

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Stefan Wagner
Betreuer:	Marvin Wyrich, M.Sc.
Beginn am:	17. Oktober 2018
Beendet am:	16. April 2019

Kurzfassung

Im Laufe der Entwicklung von Softwareprojekten werden diese immer komplexer, und die Wartbarkeit verschlechtert sich. Abhilfe schaffen sogenannte Refactorings, bei denen die ursprüngliche Funktionalität von Code beibehalten, seine Form jedoch verbessert wird. Solche Refactorings können jedoch ziemlich aufwändig sein, da sich über die Zeit sehr viele Mängel ansammeln. Viele schlechte Code-Praktiken, sogenannte Code Smells, können bereits automatisch als solche erkannt werden. Es wäre daher wünschenswert, diese auch automatisch beheben zu können.

Der hier vorgestellte Refactoring-Bot tut genau dies, mit einer statischen Code-Analyse als Eingabe führt er automatisch Refactorings durch und stellt diese als Pull-Requests auf GitHub zur Verfügung. Im Gegensatz zu manuellen Refactorings kann dadurch mit sehr geringem Zeitaufwand die Codequalität von Softwaresystemen verbessert werden. Im Rahmen dieser Arbeit wurde der Bot um das Entfernen von auskommentiertem Code erweitert, eines der häufigsten Probleme in Softwareprojekten. Außerdem wurde eine Studie durchgeführt, indem der Refactoring-Bot auf vorhandenen GitHub-Projekten ausgeführt wurde und den Entwicklern automatische Refactorings als Pull-Request vorgeschlagen hat. Das Ergebnis ist, dass der Bot zwar korrekt arbeitet, die Pull-Requests jedoch auf wenig Zustimmung stoßen. Mögliche Gründe dafür werden in der Arbeit diskutiert.

Abstract

During the development of software systems, they grow more complex and their maintainability gets worse. This can be counteracted by so-called refactorings, which improve the appearance of source code, without altering its behaviour. These refactoring can be very time consuming, since a lot of issues accumulate over time. Many bad practices, so-called code smells, can already be detected automatically. It would be desirable to have a way to also remove these automatically.

The Refactoring-Bot presented in this work does exactly that, on the basis of a static code analysis it executes refactorings and creates pull requests on GitHub with the changes. Contrary to manual refactorings, it can increase code quality with very little effort. In this work, the bot was extended to support the automatic removal of commented out code, one of the most common issues in software systems. Furthermore, a study was conducted, in which the Refactoring-Bot was executed on existing GitHub projects and suggested pull requests for refactorings to the developers. The result is that the bot works correctly, but the acceptance rate for pull requests is rather low. Possible reasons for this are discussed in this work.

Inhaltsverzeichnis

1 Einleitung **13**
1.1 Motivation . . . 13
1.2 Aufbau der Arbeit . . . 13

2 Grundlagen **15**
2.1 Auskommentierter Code . . . 15
2.2 Refactoring . . . 18
2.3 Git und GitHub . . . 19
2.4 SonarQube und SonarCloud . . . 19
2.5 Verwandte Arbeiten . . . 20

3 Implementierung **23**
3.1 Refactoring-Bot . . . 23
3.2 Identifizieren von auskommentiertem Code . . . 24
3.3 Entfernen von auskommentiertem Code . . . 25

4 Evaluation **27**
4.1 Ziele . . . 27
4.2 Vorbereitung . . . 28
4.3 Durchführung . . . 29
4.4 Metriken . . . 29
4.5 Auswertungsmethode . . . 30

5 Ergebnisse **31**
5.1 Beschreibung . . . 31
5.2 Diskussion . . . 36
5.3 Limitationen . . . 39

6 Fazit **41**

Literaturverzeichnis **43**

Abbildungsverzeichnis

3.1 Funktionsprinzip des Refactoring-Bots 23
3.2 Das Web-Interface des Refactoring-Bots 24

5.1 Diagramm mit Anzahl der Pull-Requests 32
5.2 Alter und Akzeptanz von Pull-Requests 35
5.3 Länge und Akzeptanz von Pull-Requests 36

Verzeichnis der Tabellen

2.1 Vergleich verschiedener Tools zum automatischen Refactoring 22

5.1 Übersicht der erstellten Pull-Requests nach Softwareprojekt 31
5.2 Größe der Projekte 32
5.3 Übersicht der bearbeiteten Issues 34

Verzeichnis der Listings

2.1 Ein Beispiel für auskommentierten Code 15
2.2 Beispiel für veralteten Code als Kommentar 16
2.3 Beispiel für unfertigen Code als Kommentar 16
2.4 Debug-Code, mittels boolean-Variable deaktiviert 17

3.1 Ein- und Ausgabe mit dem *LexicalPreservingPrinter* 25

1 Einleitung

1.1 Motivation

Auskommentierter Code ist ein häufiges Phänomen in Softwareprojekten. Auf der Code-Analyse Webseite SonarCloud[1] werden verschiedene Probleme von Softwareprojekten festgehalten. Schaut man sich auf der Seite die Probleme von Java-Projekten an, so sieht man, dass von ca. sieben Millionen "Code Smells" etwa 389.000 auf auskommentierten Code entfallen. Damit ist auskommentierter Code das zweithäufigste Problem. Der Begriff *Code Smell* wurde von Kent Beck und Martin Fowler erfunden, sie bezeichnen damit schlecht strukturierten Quellcode [Fow99].

Als Gegenmaßnahme werden sogenannte Refactorings durchgeführt [Fow99]. Dabei handelt es sich um Änderungen am Quellcode, die die Lesbarkeit verbessern, ohne die Funktionalität des Codes zu verändern. Ziel ist es, die Softwarequalität zu erhöhen und die Komplexität zu reduzieren. Refactorings sind jedoch sehr zeitaufwändig, da sich während der Lebenszeit eines Softwareprojekts sehr viele Code Smells ansammeln.

Der Refactoring-Bot[2] soll dieses Problem lösen, indem er automatisch problematische Codestellen erkennt und Refactorings durchführt. Der Refactoring-Bot wurde im Rahmen einer vorherigen Bachelorarbeit entwickelt, in dieser Arbeit wird er um das Entfernen von auskommentiertem Code erweitert, wodurch ein weiterer häufiger Code Smell automatisch behoben werden kann. Dadurch kann die Codequalität einer Software ohne großen Aufwand verbessert werden.

1.2 Aufbau der Arbeit

Zunächst werden in Kapitel 2 verschiedene grundlegende Begriffe für diese Arbeit näher erläutert, darunter Refactorings und auskommentierter Code. Außerdem werden verwandte Arbeiten zum Thema automatisches Refactoring vorgestellt und mit dem Refactoring-Bot verglichen. Die Implementierung des automatischen Entfernens von auskommentiertem Quellcode wird in Kapitel 3 besprochen. Zunächst wird der bereits existierende Refactoring-Bot vorgestellt, danach wird beschrieben, wie dieser um das Entfernen von auskommentiertem Code erweitert wurde und welche anderen Verbesserungen durchgeführt wurden. In Kapitel 4 folgt die Evaluation dieser Implementierung. Dazu wird der Refactoring-Bot, beschränkt auf das Entfernen von auskommentiertem Code, auf verschiedenen Softwareprojekten ausgeführt. Die Änderungen des Bots werden den Entwicklern als Pull-Requests auf GitHub zur Verfügung gestellt, und es wird ausgewertet, wie viele dieser Änderungen abgelehnt werden und ob der Bot diese korrekt durchgeführt hat. Anschließend

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

[2] https://github.com/Refactoring-Bot/Refactoring-Bot

wird diskutiert, aus welchen Gründen vom Bot vorgeschlagene Änderungen abgelehnt werden. Aus den daraus gewonnenen Erkenntnissen werden Schlüsse für die weitere Entwicklung des Bots gezogen.

Kapitel 6 fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf die weitere Entwicklung des Refactoring-Bots.

2 Grundlagen

2.1 Auskommentierter Code

Als auskommentierten Code bezeichnet man Codestellen, die in einen Kommentar gesetzt wurden. Softwareentwickler nutzen dies, um bestimmten Code temporär zu deaktivieren. Mögliche Gründe dafür werden im Verlauf dieses Kapitels besprochen.

Solche auskommentierten Codestellen können die Lesbarkeit eines Programms deutlich verschlechtern, da sie weder etwas zur Funktionalität des Programmes beitragen, noch eine nützliche Form von Dokumentation darstellen. Häufig stiftet auskommentierter Code sogar Verwirrung bei anderen Entwicklern und hindert das Verständnis des Programmcodes. Neue Entwickler, die sich in den Code einarbeiten wollen, werden durch auskommentierten Code abgelenkt. So könnte man in Versuchung kommen, den auskommentierten Code zu analysieren, da er eventuell wichtig sein könnte, oder man seine Funktionsweise verstehen möchte. Daher ist auskommentierter Code nicht nur nutzlos, sondern schadet sogar dem Verständnis der Software.

```
public class Example {
    public int addNumbers(int a, int b) {
        // int c = a + b;
        int c = b + a;
        return c;
    }
}
```

Listing 2.1: Ein Beispiel für auskommentierten Code

2.1.1 Ursachen

Veralteter Code

Einer der häufigsten Gründe für auskommentierten Code ist der Glaube, der auskommentierte Code werde später noch einmal gebraucht [YM12]. Dies geschieht zum Beispiel, wenn alter Code refactored wird, das heißt die Lesbarkeit von Code wird verbessert, ohne dessen Funktion zu verändern. Viele Entwickler kommentieren dazu den alten Code aus, um ihn als Referenz für ihre neue Lösung zu nehmen.

Jedoch ist es häufig so, dass dieser auskommentierte Code hinterher, nachdem der neue Code bereits fertig implementiert wurde, nicht aufgeräumt wird. Das liegt zum Teil daran, dass Entwickler glauben, den alten Code in Zukunft eventuell doch noch zu brauchen. Falls der neue Code unerwartet Probleme machen sollte, könnte man so zur alten Version zurückkehren.

```
private void printList (List list) {
    /*
    for (int i = 0; i < list.size() - 1; i++) {
        System.out.println(list.get(i);
    }
    */
    list.forEach(item->System.out.println(item));
}
```

Listing 2.2: Beispiel für veralteten Code als Kommentar

In Listing 2.2 sieht man ein typisches Beispiel für archivierten Code als Kommentar. Obwohl bereits eine elegantere Lösung gefunden wurde, wird der alte Code noch als Kommentar beibehalten, wodurch die Lesbarkeit des Codes erheblich beeinträchtigt wird.

Kent C. Dodds, Softwareentwickler bei Paypal, schlägt in seinem Blog vor, als Lösung für diese Art des auskommentierten Codes eine Versionsverwaltung wie Git zu benutzen, anstatt den alten Code auszukommentieren [Dod17] . Sollte der alte Code tatsächlich noch einmal gebraucht werden, könne man diesen stattdessen aus einem Git-Repository wiederherstellen. Sollte man der Meinung sein, auskommentierter Code könnte noch einmal wichtig werden, könnte man nach dessen Entfernung einen Kommentar hinterlassen, wann der Code entfernt wurde und welche Funktion er hatte. So kann man diesen bei Bedarf leichter wiederfinden.

Code in Entwicklung

Ein weiterer Grund für das Auskommentieren von Code sind Codeabschnitte, die noch nicht funktionsfähig sind. Beispielsweise könnten diese fehlerhaft sein, oder benötigte Bibliotheken sind noch nicht verfügbar, weswegen der Code nicht kompiliert. Viele Entwickler kommentieren daher in Entwicklung befindlichen Code aus, damit das gesamte Projekt weiterhin kompiliert.

```
private Object getObject (int index) {
    // TODO: This still throws errors
    // library.getObject(index);
    return null;
}
```

Listing 2.3: Beispiel für unfertigen Code als Kommentar

Listing 2.3 zeigt ein Beispiel für Code, der noch nicht korrekt funktioniert und daher auskommentiert wurde.

Aber auch hier wäre eine Versionsverwaltung die sinnvollere Lösung. Anstatt den in Entwicklung befindlichen Code auszukommentieren, kann man beispielsweise in Git einen neuen Branch erstellen, in dem man das neue Feature implementiert. Sollte dieses noch nicht funktionieren, ist somit nicht die Funktionalität des Master-Branches gestört. Wenn man das Feature fertiggestellt hat, kann man seinen Feature-Branch mergen.

Code zum Debuggen

Viele Softwareprojekte enthalten Codeabschnitte, die nur zum Debuggen benutzt werden. Diese werden dann auskommentiert, wenn sie nicht benutzt werden. Während Debug-Code eine gängige und sinnvolle Praxis ist, ist das Auskommentieren von Code dazu ungeeignet, und bringt einige Nachteile mit sich. Das größte Problem ist die Tatsache, dass der Code *verrottet*. Da der Code auskommentiert ist, wird er vom Compiler nicht mehr erkannt und überprüft. Dies führt dazu, dass sämtliche Refactorings, wie zum Beispiel das Ändern von Variablennamen, nicht auf den auskommentierten Code angewendet werden. Auch Änderungen an Methodensignaturen oder nicht mehr vorhandene Abhängigkeiten können nicht erkannt werden. Somit besteht die Chance, dass auskommentierter Code nicht mehr kompiliert, wenn er wieder einkommentiert wird. In einem solchen Fall müsste man den betroffenen Code manuell beheben und an die aktuelle Programmversion anpassen, was Zeit kostet.

Eine bessere Lösung wäre daher, den Debug-Code durch einen *#ifdef* Befehl vom Präprozessor ignorieren zu lassen [Sta11]. Dies hat jedoch den Nachteil, dass das ganze Projekt neu gebaut werden muss, um zwischen den Codevarianten zu wechseln. Sollte dies zu zeitaufwändig sein, oder man arbeitet in einer Programmiersprache, die keine Befehle wie *#ifdef* anbietet, wäre es auch möglich, den Code mit Hilfe einer speziellen Variable und einer if-Abfrage zu deaktivieren. In Java könnte dies wie folgt aussehen:

```java
boolean debug = false;

if (debug) {
  debugCode();
}
```

Listing 2.4: Debug-Code, mittels boolean-Variable deaktiviert

Dieser Code wird bei der Kompilierung ignoriert und beeinträchtigt somit nicht die Laufzeit, aber er wird weiterhin auf Korrektheit überprüft. Dadurch lässt sich ein Verrotten des Codes verhindern, was bei auskommentiertem Code nicht der Fall ist.

2.1.2 Sinnvolle Anwendungen von auskommentiertem Code

Wie oben bereits aufgezeigt wurde, lässt sich auskommentierter Code durch verschiedene Methoden, wie das Benutzen einer Versionsverwaltung, häufig vermeiden. Nun stellt sich die Frage, ob es überhaupt Situationen gibt, in denen auskommentierter Code sinnvoll sein kann.

Codebeispiele als Dokumentation

Es kann sinnvoll sein, Codebeispiele zur Dokumentation einer Klasse oder einer Methode in einen Kommentar zu integrieren. Solche Dokumentationen sollte der Refactoring-Bot möglichst erkennen und vermeiden, diese zu entfernen. SonarQube, welches dem Refactoring-Bot Codestellen mit auskommentiertem Code liefert, enthält bereits eine Reihe von Überprüfungen, um das Entfernen von

normaler Dokumentation zu verhindern. So wird davon ausgegangen, dass Kommentare am Anfang einer Datei zu einer Beschreibung der Lizenz gehören [1]. Außerdem werden Javadoc-Kommentare ausgeschlossen, da diese für gewöhnlich nicht zum Auskommentieren von Code benutzt werden.

Referenz für Refactoring

Beim Refactoring von größeren Codeabschnitten ist es häufige Praxis, den alten Code auszukommentieren, damit man diesen als Referenz heranziehen kann. Man kann so auch sicherstellen, dass man die gesamte Funktionalität des alten Codes implementiert.

Diese Art von auskommentiertem Code sollte jedoch zeitnah wieder entfernt werden, da er nach dem Abschluss des Refactorings nicht mehr gebraucht wird. Am besten sollte der auskommentierte Code erst gar nicht in einer Versionsverwaltung eingecheckt werden. Daher wäre es denkbar, dem Refactoring-Bot das Alter des auskommentierten Codes als ein Kriterium für die Entfernung zu Grunde zu legen. Ist auskommentierter Code erst ein paar Tage alt, so könnte ein Entwickler ihn noch aktiv als Referenz benutzen. Ist er jedoch mehrere Monate alt, so ist es unwahrscheinlich, dass er noch irgendeinen Zweck erfüllt.

2.2 Refactoring

Als ein Refactoring bezeichnet man im Allgemeinen eine Veränderung eines Softwaresystems, bei der die interne Struktur verbessert wird, ohne das externe Verhalten des Codes zu modifizieren [Fow99]. Während der Entwicklung eines Softwaresystems wird dieses immer komplexer, und entfernt sich mehr und mehr von seinem ursprünglichen Design, wodurch die Qualität des Codes gemindert wird [MT04]. Das Refactoring hat zum Ziel, die Softwarequalität wieder zu erhöhen und die Komplexität zu reduzieren.

Das Entfernen von auskommentiertem Code fällt somit unter die Definition eines Refactorings, da Kommentare keinen Einfluss auf das Verhalten des Programmcodes haben und somit bedenkenlos entfernt werden können.

Tom Mens [MT04] beschreibt folgende Schritte zur Durchführung eines Refactorings:

1. Stellen identifizieren, an denen die Software refactored werden soll
2. Feststellen, welche Refactorings an den identifizierten Stellen angewendet werden sollen
3. Garantieren, dass das angewendete Refactoring das Verhalten des Codes beibehält
4. Das Refactoring anwenden
5. Die Auswirkung des Refactorings auf die Qualitätsmerkmale der Software (z.B. Komplexität, Verständlichkeit, Wartbarkeit) oder den Prozess (z.B. Produktivität, Aufwand, Kosten) feststellen

[1] https://github.com/SonarSource/sonar-java/blob/master/java-checks/src/main/java/org/sonar/java/checks/CommentedOutCodeLineCheck.java

6. Die Konsistenz zwischen dem refactorten Code und den anderen Software-Artefakten (wie Dokumentation, Design-Dokumente, Requirements Specification, Tests, etc..) beibehalten

Der Refactoring-Bot ist in der Lage, viele dieser Aufgaben zu automatisieren. Er kann passende Codestellen selbstständig identifizieren, ein geeignetes Refactoring auswählen, und es anwenden. Die Auswirkung auf die Qualität der Software kann durch eine erneute SonaQube-Analyse festgestellt werden, nachdem Refactorings durchgeführt wurden. Der Bot überprüft nicht, dass das Verhalten des Codes beibehalten wird, dies muss von den Entwicklern selbst durch geeignete Tests sichergestellt werden. Außerdem muss der letzte Schritt ebenfalls von den Entwicklern selbst durchgeführt werden, da der Bot nicht in der Lage ist, Dokumentation selbstständig zu aktualisieren.

2.3 Git und GitHub

Git ist eine Versionsverwaltung für Dateien, die häufig bei der Entwicklung von Softwaresystemen zum Einsatz kommt. Ins Leben gerufen wurde es von Linux-Entwickler Linus Torvalds im Jahr 2005 für die Entwicklung des Linux-Kernels [Spi12]. Der Vorteil der Verwendung einer Versionsverwaltung ist, dass alle Änderung am Code protokolliert werden, mit Datum und Name des Entwicklers. So sind diese leichter nachvollziehbar und können bei Bedarf auch relativ einfach rückgängig gemacht werden.

Für das Aufsetzen eines öffentlichen Git-Repositories gibt es verschiedene Anbieter. Der größte davon ist GitHub [2], welches auch vom Refactoring-Bot verwendet werden kann, um Zugriff auf den Quellcode zu erhalten und Änderung bereitzustellen. GitHub bietet die Möglichkeit, anderen Entwicklern Änderungen an ihrem Code durch sogenannte *Pull-Requests* vorzuschlagen. Die Entwickler haben dann die Möglichkeit, die vorgeschlagenen Änderung zu überprüfen, mögliche Modifikationen vorzuschlagen, oder den Pull-Request direkt zu *mergen*. Möchte man die Änderungen nicht übernehmen, so schließt man einen Pull-Request. Der Refactoring-Bot nutzt Pull-Requests, um durchgeführte Refactorings bereitzustellen.

2.4 SonarQube und SonarCloud

SonarQube [3] ist ein Programm zur statischen Codeanalyse. Ein solches Programm überprüft Quellcode auf Fehler, ohne diesen auszuführen. So können Fehler im Programmcode schon vor der Kompilierung und Ausführung festgestellt werden. Dabei können sowohl Syntaxfehler, die das kompilieren verhindern würden, als auch Code Smells festgestellt werden. Das Programm SonarQube zielt auf letztere ab, es stellt verschiedene Code Smells fest, und bestimmt damit Qualitätsmetriken für den untersuchten Quellcode. Da Code Smells oftmals bestimmten Mustern folgen, können sie von Programmen wie SonarQube automatisch festgestellt werden. Einige häufig verwendete Refactorings sind beispielsweise duplizierter Code oder lange Methoden, bei denen ein Teil des Codes in eine neue Methode ausgelagert werden.

[2] https://github.com/

[3] https://www.sonarqube.org/

Um die Analyseergebnisse von SonarQube für andere Entwickler leichter zugänglich zu machen, können diese auf der Webseite *SonarCloud* gehostet werden. Diese kann der Refactoring-Bot benutzen, um die statische Codeanalyse eines Softwareprojekts abzurufen und basierend darauf Refactorings durchzuführen. Über die API von SonarCloud ist es möglich, Details über jeden Code Smell abzurufen, darunter die Art des Problems, die Datei, und die Codezeile. Dadurch kann der Refactoring-Bot die passenden Refactorings an den korrekten Stellen durchzuführen.

Der Refactoring-Bot ist ebenso in der Lage, lokale SonarQube-Analysen zu verwenden oder diese von einer anderen Webseite zu beziehen.

2.5 Verwandte Arbeiten

Es gibt bereits eine Reihe von Programmen, die automatisches Refactoring umsetzen. Die meisten werden als Plugins für Entwicklungsumgebungen angeboten und führen Refactorings lokal durch. Im Gegensatz dazu ist der hier vorgestellte Refactoring-Bot eine eigenständige Anwendung, die den Programmcode selbstständig von GitHub herunterlädt und die Änderungen später wieder als Pull-Requests zur Verfügung stellt.

JDeodorant

JDeodorant [4] ist ein Plugin für die Eclipse Entwicklungsumgebung, welches Code Smells erkennt und automatisch Refactorings anwendet. Unterstützt werden *Feature Envy, State Checking, Type Checking, Long Method, God Class, Duplicated Code* und *Refused Bequest* [TCC18]. Das Programm untersucht den Quellcode, identifiziert Stellen an denen Refactorings angewendet werden können, und führt dann das Refactoring durch. Anschließend wird sichergestellt, dass sich das Verhalten des Codes nicht verändert.

Laut den Entwicklern ist JDeodorant eines der populärsten Tools für automatisches Refactoring, so wurden von 2011 bis 2017 35.000 Refactorings mit dem Tool durchgeführt [TCC18]. Außerdem wird es in zahlreichen Arbeiten zitiert und für ähnliche Tools oft als Vergleich herangezogen. Des Weiteren gibt es viele Programme, die ganz oder teilweise auf JDeodorant basieren.

Fontana et al. testeten in einer Studie über verschiedene Refactoring-Tools unter anderem JDeodorant [FMPZ15] . Die Autoren kamen zu dem Fazit, dass JDeodorant *"manchmal Bugs einführt oder Entwickler täuscht"* [FMPZ15]. Außerdem kam es zu Abstürzen des Tools.

The Spartanizer

The Spartanizer [5] ist ein Plugin für die Eclipse Entwicklungsumgebung, welches darauf abzielt, den Quellcode eines Programms durch automatisches Refactoring auf einen sogenannten *spartan style* zu reduzieren. Dies beschreibt *"einen sparsamen Coding-Stil, der die Nutzung von Zeichen,*

[4] https://github.com/tsantalis/JDeodorant
[5] https://github.com/SpartanRefactoring/Main

Variablen, Token, etc.. minimiert" [GO17]. Dabei werden über 150 verschiedene Refactorings angewendet, welche das Plugin automatisch in der richtigen Reihenfolge durchführt. Das Programm soll dabei eine geringe Fehlerrate aufweisen. In einem Test wurden nur in 14 von 422 Dateien Fehler eingeführt, welche sich jedoch leicht beheben ließen [GO17].

Fault Buster

Fault Buster [6] ist ein Toolset zum automatischen Refactoring, welches Plugins für verschiedene bekannte Entwicklungsumgebungen, darunter Eclipse, Netbeans und IntelliJ IDEA, anbietet [SNF+15]. Es besitzt ebenfalls ein eigenständiges Interface. Das Programm unterstützt 40 verschiedene Refactorings, die automatisch an passenden Stellen durchgeführt werden können. Das Tool soll vor allem beim *continuous refactoring* hilfreich sein, das heißt es werden laufend kleinere Verbesserungen durchgeführt, anstatt dass auf einmal größere Änderungen durchgeführt werden [SNF+15]. Teilweise wird auch potenziell unerwünschtes Verhalten von Code behoben, was streng genommen nicht unter die Definition eines Refactorings fällt, da diese das Verhalten des Codes nicht verändern sollen.

Fault Buster kann, ähnlich wie der Refactoring-Bot, automatisch den neuesten Quellcode aus einem Versionskontrollsystem wie Git beziehen. Das Tool ist jedoch nicht in der Lage, durchgeführte Refactorings als Pull-Request zur Verfügung zu stellen. Refactorings können auch über ein Webservice-Interface durchgeführt werden, in dem auch verschiedene Einstellungen konfiguriert werden können.

Code-Imp

Code-Imp ist ein Tool, welches nicht nur automatische Refactorings unterstützt, sondern auch über 25 Qualitätsmetriken des analysierten Codes erheben kann [MÓ11]. Diese fließen auch in das automatische Refactoring ein. Ein Refactoring wird nämlich nur durchgeführt, wenn dadurch die Metriken und somit auch das Design des Codes verbessert werden können. Code-Imp wird über die Kommandozeile bedient und besitzt keine Integration für Entwicklungsumgebungen.

Das Tool unterstützt 14 verschiedene Refactorings.

Vergleich mit Refactoring-Bot

Wie man der obigen Tabelle entnehmen kann, ist das Alleinstellungsmerkmal des Refactoring-Bots die Unterstützung von Versionskontrollsystemen für das automatische Herunterladen von Quellcode und die Bereitstellung von Änderungen als Pull- oder Merge-Request. Dadurch kann der Refactoring-Bot nahtlos in den Softwareentwicklungsprozess eingebunden werden, da eine einzige Ausführung für die Durchführung und Bereitstellung eines Refactorings ausreicht. Bei anderen Programmen muss der Quellcode manuell heruntergeladen werden, woraufhin Refactorings lokal

[6] http://www.sed.inf.u-szeged.hu/FaultBuster

Name	Automatische Refactorings	Anzahl Refactorings	Schnittstelle	Unterstützt Versionskontrolle
JDeodorant	Ja	7	Eclipse Plugin	Nein
The Spartanizer	Ja	Über 150	Eclipse Plugin	Nein
Fault Buster	Ja	40	Eclipse, Netbeans, IntelliJ IDEA Plugin	Nur Download
Code-Imp	Ja	14	Kommandozeile	Nein
Refactoring-Bot	Ja	5	Web-Interface	Download und Upload

Table 2.1: Vergleich verschiedener Tools zum automatischen Refactoring

durchgeführt werden und erneut händisch ein Commit und ein Pull-Request erstellt werden müssen. Hat man die Konfiguration für ein Projekt im Refactoring-Bot einmal aufgesetzt, ist im Idealfall nur ein Klick notwendig, um ein Refactoring für ein ganzes Projekt durchzuführen.

Die Anzahl der unterstützten Refactorings ist derzeit im Vergleich zu den anderen vorgestellten Tools noch gering, wird jedoch stetig erweitert.

3 Implementierung

3.1 Refactoring-Bot

Der *Refactoring-Bot*, welcher derzeit am Institut für Softwaretechnologie (ISTE)[1] entwickelt wird, führt automatisch Refactorings durch, welche dann als Pull-Request auf GitHub zur Verfügung gestellt werden. So können Entwickler bequem einzelne Refactorings annehmen oder ablehnen. Der Bot basiert auf dem Spring Framework und besitzt eine Benutzeroberfläche, auf die man mit einem Webbrowser zugreifen kann. Konfigurationen werden in einer SQL-Datenbank gespeichert, wodurch man mehrere Softwareprojekte speichern kann und dann selektiert, auf welchem man Refactorings durchführen möchte.

Zunächst erstellt der Nutzer eine Konfiguration für sein Projekt, für die ein auf GitHub gehostetes Projekt und eine von SonarQube erstellte Qualitätsanalyse benötigt werden. Nachdem der Bot das Projekt von GitHub heruntergeladen hat, führt er lokal Refactorings durch. Dabei wird die in der Konfiguration gespeicherte Analyse des Quellcodes von SonarQube als Basis für die Erkennung der Code Smells benutzt. Die von SonarQube vorgegebenen Issues werden mit entsprechenden Refactoring-Klassen verknüpft, in denen dann der Code Smell entfernt wird. Schließlich wird nach dem Ausführen der Refactorings für jedes durchgeführte Refactoring ein separater Pull-Request auf GitHub erstellt.

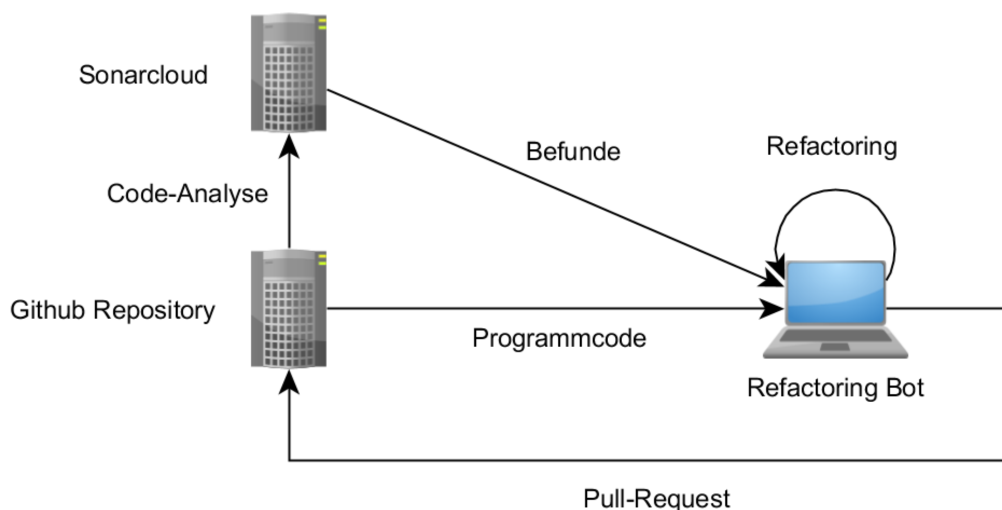

Figure 3.1: Funktionsprinzip des Refactoring-Bots

[1] https://www.iste.uni-stuttgart.de/

Abbildung 3.1 verdeutlicht die Arbeitsweise des Refactoring-Bots. Der Bot erhält eine SonarQube-Analyse und ein Git-Repository als Eingabe, und stellt dann die fertigen Refactorings wieder als Pull-Request auf demselben Repository zur Verfügung.

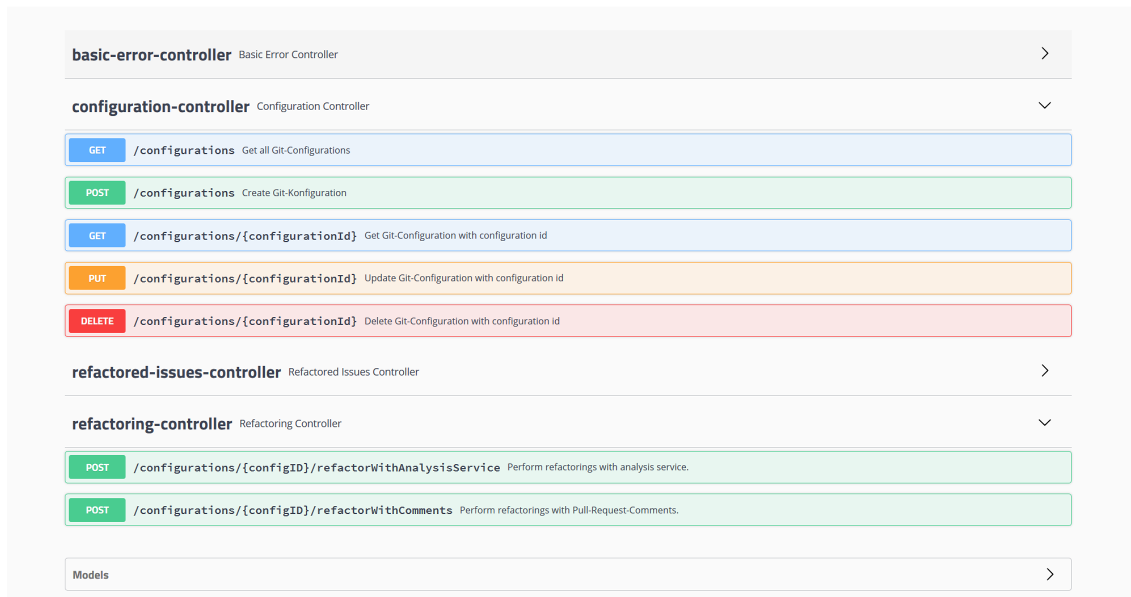

Figure 3.2: Das Web-Interface des Refactoring-Bots

In Abbildung 3.2 ist das Interface des Refactoring-Bots zu sehen. Beim Start des Programms wird automatisch das Web-Interface im Standard-Browser des Nutzers aufgerufen. Dort kann er verschiedene Einstellungen vornehmen, wie das Hinzufügen und Löschen von Konfigurationen für verschiedene Softwareprojekte. Sobald eine Konfiguration erstellt ist, kann mit Angabe der Konfigurations-ID direkt ein Refactoring gestartet werden.

Jedes Refactoring wird als eigene Klasse implementiert, wodurch der Bot um beliebige Refactorings erweitert werden kann. Jede dieser Klassen ist dabei mit der ID eines SonarQube-*Issues* verknüpft, auf Basis derer der Bot das passende Refactoring auswählen kann und die entsprechende Codestelle übergeben bekommt. Im Rahmen dieser Arbeit soll der bestehende Bot um das Entfernen von auskommentiertem Quellcode erweitert werden.

3.2 Identifizieren von auskommentiertem Code

Sämtliche Befunde liest der Bot aus einer SonarQube Analyse ein. Dazu kann die Web-Schnittstelle von SonarCloud[2] benutzt werden, wenn dort eine Analyse des Projekts hinterlegt ist. Alternativ kann auch eine lokale SonarQube-Analyse verwendet werden.

Wird eine Analyse von SonarCloud benutzt, so werden mit einer Reihe von HTTP-Anfragen alle Issues abgefragt und in einer Liste gespeichert. Diese Funktionalität war im Refactoring-Bot bereits vorhanden, wurde für diese Arbeit jedoch in einigen Punkten verbessert. Insbesondere werden nun

[2] https://SonarCloud.io/web_api

alle Issues eines Projekts von SonarCloud abgefragt, wenn nötig mit mehreren HTTP-Anfragen, da jede Anfrage auf 500 Issues beschränkt ist. Nachdem alle Issues gesammelt wurden, übersetzt der Bot diese in *BotIssue*-Objekte. Dies ist eine Datenstruktur, die Information zu den Issues enthält, die für das Refactoring benötigt werden. Dazu gehören unter anderem der Pfad der Datei, in der das Problem auftritt, und die entsprechende Codezeile. Außerdem wird den Issues, basierend auf der SonarQube Issue-ID, ein passendes Refactoring zugewiesen. Im Falle von auskommentiertem Code wird die Issue-ID *squid:CommentedOutCodeLine* mit der *RemoveCommentedOutCode*-Klasse verknüpft.

Die SonarQube-Analyse muss von den Entwicklern des Softwareprojekts aufgesetzt und durchgeführt werden. Anschließend muss man die Projekt-ID von SonarCloud in die Bot-Konfiguration eintragen, oder eine lokale SonarQube-Analyse spezifizieren. Dabei ist es wichtig, dass die SonarQube-Analyse mit der aktuellsten Version des Quellcodes durchgeführt wurde. Der Bot überprüft dies nicht, also kann es eventuell zu Fehlfunktionen kommen, falls die Versionen von Analyse und Quellcode nicht übereinstimmen. Da es nur schwer möglich wäre dies automatisch zu überprüfen, wird es dem Benutzer überlassen, die Korrektheit der eingegebenen Informationen sicherzustellen.

3.3 Entfernen von auskommentiertem Code

Das automatische Entfernen von auskommentiertem Code findet in der Klasse *RemoveCommented-OutCode.java* statt. Diese ruft der Bot jedes Mal auf, wenn ein entsprechendes Issue von SonarQube bearbeitet werden soll. Als Eingabe erhält die Klasse ein *BotIsssue*-Objekt und das *GitConfiguration*-Objekt der dazu passenden Projektkonfiguration. Basierend auf diesen Informationen wird der Pfad zum Quellcode zusammengesetzt und die Datei eingelesen. Da in der SonarQube-Analyse nur die Anfangszeile eines Blocks von auskommentiertem Code angegeben wird, stellt der Bot selbstständig fest, wie weit dieser Block geht, und entfernt alle betroffenen Zeilen aus dem Quellcode.

Zur Entfernung eines Kommentars wird die Bibliothek *JavaParser*[3] benutzt. Diese übersetzt den Code aus der Eingabedatei in einen Syntaxbaum, über den man Änderungen am Code durchführen kann. Dadurch wird sichergestellt, dass Kommentare korrekt identifiziert und sauber entfernt werden.

```java
// Prepare data
String path = gitConfig.getRepoFolder() + "/" + issue.getFilePath();

// Read file
FileInputStream in = new FileInputStream(path);
CompilationUnit compilationUnit =
    LexicalPreservingPrinter.setup(StaticJavaParser.parse(in));
{...}
PrintWriter out = new PrintWriter(path);
out.println(LexicalPreservingPrinter.print(compilationUnit));
out.close();
```

Listing 3.1: Ein- und Ausgabe mit dem *LexicalPreservingPrinter*

[3] https://github.com/javaparser/javaparser

Um die Eingabedatei einzulesen und später wieder zurückzuschreiben, wird der sogenannte *LexicalPreservingPrinter* des JavaParsers benutzt. Dieser hat den Vorteil, dass später bei der Ausgabe die ursprüngliche Formatierung des Codes beibehalten wird.

Eine Herausforderung bei der Implementierung war, dass der JavaParser bisher nicht das Entfernen von Kommentaren zusammen mit dem *LexicalPreservingPrinter* unterstützt hat. Benutzt man stattdessen *pretty printing*, das heißt man formatiert den Code bei der Ausgabe neu, bekommt man in Git keine saubere Versionshistorie, da die komplette Datei umgeschrieben wird. Dies ist für eine Anwendung wie den Refactoring-Bot äußerst ungünstig, da Entwickler sehen müssen, welche Zeilen vom Bot verändert wurden. Würde man Dateien bei jedem Refactoring neu formatieren, so wären Änderungen später nicht mehr nachvollziehbar.

Zur Lösung dieses Problems wurde die entsprechende Funktionalität im JavaParser implementiert und den Entwicklern als Pull-Request übermittelt[4]. Dieser wurde schließlich ab Version 3.13.0 integriert.

[4]https://github.com/javaparser/javaparser/pull/2082

4 Evaluation

4.1 Ziele

In dieser Studie soll der Refactoring-Bot mit dem automatischen Entfernen von auskommentiertem Quellcode auf vorhandenen Softwareprojekten ausgeführt werden. Es soll geklärt werden, ob das automatische Refactoring von auskommentiertem Code sinnvoll ist und von Entwicklern akzeptiert wird. Zusätzlich soll festgestellt werden, in welchen Fällen diese automatischen Refactorings abgelehnt werden, und ob man diese verhindern kann, beispielsweise durch Überprüfung des Alters oder der Länge des auskommentierten Codes.

Außerdem soll die Korrektheit der durchgeführten Refactorings überprüft werden. Hier wird zwischen drei Fällen unterschieden:

- **Fall 1**: Korrekt. Der komplette auskommentierte Codeabschnitt wird fehlerfrei und restlos entfernt
- **Fall 2**: Unvollständig. Es verbleiben einzelne Abschnitte des auskommentierten Quellcodes
- **Fall 3**: Fehlerhaft. Der Quellcode kompiliert nach dem Refactoring nicht mehr, da das Entfernen fehlerhaft durchgeführt wurde

Fall 2 wäre zwar ungünstig, aber nicht sehr problematisch. Verbleibende Kommentare könnten hinterher manuell entfernt werden. Fall 3 wäre jedoch kritisch, da der Refactoring-Bot nie die Kompilierbarkeit des Codes beeinträchtigen sollte. Sollte ein solcher Fall auftreten, müsste genau überprüft werden, wodurch dieses Problem verursacht wird und wie man es in Zukunft verhindern kann.

Somit sollen folgende Forschungsfragen geklärt werden:

- **RQ1a** Welcher Anteil der von dem Bot durchgeführten Refactorings wird angenommen?
- **RQ1b** Aus welchen Gründen werden Refactorings abgelehnt?
- **RQ2** Lässt sich ein Bezug zwischen Alter und Länge des auskommentierten Codes und der Rate der Akzeptanz herstellen?
- **RQ3:** Führt der Bot die Refactorings fehlerfrei durch? Ist der Code weiterhin kompilierbar?

4.2 Vorbereitung

Um die Aussagekraft der Resultate zu erhöhen, wird der Bot vor der Evaluation in einigen Punkten modifiziert. Zunächst werden die vom Bot zu bearbeitenden Issues auf auskommentierten Code beschränkt, da andere Refactorings für diese Evaluation uninteressant sind. Des Weiteren wird die Reihenfolge der Issues abgeändert. Normalerweise arbeitet der Bot diese der Reihe nach ab, d.h. nach der Reihenfolge wie sie in SonarCloud sortiert sind. Da im Rahmen dieser Evaluation aber nur wenige Issues pro Projekt bearbeitet werden, würde dies dazu führen, dass viele der Refactorings in den gleichen Dateien durchgeführt werden. Um aussagekräftigere Ergebnisse zu erhalten, wird der Bot so modifiziert, dass Issues in einer zufällig gewählten Reihenfolge bearbeitet werden. So erhält man eine zufällige Auswahl an bearbeiteten Issues in verschiedenen Dateien und mit verschiedenem Alter.

Zusätzlich wurde die Überprüfung von Kommentaren auf auskommentierten Code nach der von SonarQube angegebenen Zeile deaktiviert. Dies führt dazu, dass der Bot, ausgehend von der von SonarQube gelieferten Zeile, alle zusammenhängenden Kommentare löscht. Die Studie soll klären, inwieweit dieses Verhalten erwünscht ist.

Die vom Bot an die Pull-Requests angehängte Nachricht wird angepasst, um einen kurzen Überblick über den Zweck dieser Arbeit zu geben. Insbesondere wird erwähnt, dass es sich bei den Pull-Requests um automatisch generierte Refactorings handelt. Außerdem werden die Entwickler gebeten, gegebenenfalls ihre Gründe für die Ablehnung eines Pull-Requests zu beschreiben. Die folgende Nachricht wird an jeden Pull-Request angehängt:

"Hello! We are currently testing a bot that automatically removes commented out code based on a SonarQube analysis (`https://github.com/Refactoring-Bot/Refactoring-Bot`). This is an automated pull request created by the bot. Please accept or deny it, based on its usefulness. If you decide to reject it, please briefly describe your reason (e.g. false positive). Thank you very much for your cooperation!
Best regards,
Justin Kissling"

4.2.1 Auswahl der Projekte

Softwareprojekte, die für die Evaluation des Refactorings geeignet sind, müssen folgende Kriterien erfüllen:

- Das Projekt muss auf GitHub gehostet sein, damit der Bot Zugriff auf den Code hat und Pull-Requests erstellen kann
- Es muss in Java programmiert worden sein, da der Bot keine anderen Programmiersprachen unterstützt
- Die letzte Projektaktivität darf höchstens zwei Wochen zurückliegen, da bei verlassenen Projekten Pull-Requests wahrscheinlich nicht mehr bearbeitet werden

Auf Basis dieser Kriterien werden passende Projekte auf der Übersichtsseite von SonarCloud ermittelt [1]. Da diese bereits absteigend nach dem Datum der letzten Analyse sortiert sind, werden nur aktive Projekte angezeigt.

Zusätzlich wird nach Java-Projekten gefiltert, und Projekte mit weniger als 1000 Codezeilen werden ausgeschlossen, da diese häufig nicht genügend Befunde haben. Somit wird mit folgender Query gesucht: https://sonarcloud.io/explore/projects?languages=java&size=2&sort=-analysis_date

Außerdem wird die SonarQube-Analyse von jedem Projekt auf das Vorhandensein von auskommentiertem Code untersucht, da der Bot sonst keine Befunde zum Bearbeiten hätte. Wird ein passendes Projekt identifiziert, so wird eine Konfiguration im Refactoring-Bot aufgesetzt. Die passenden GitHub-Repositories sind dabei entweder direkt auf der SonarCloud-Seite des Projekts verlinkt, oder werden durch eine Google-Suche ermittelt.

Um zu überprüfen, ob ein Projekt noch aktiv ist, wird das jeweilige GitHub-Repository betrachtet. Da eine Analyse auch von anderen Personen auf SonarCloud hochgeladen werden kann, bedeutet eine neuere Analyse nicht unbedingt, dass das Projekt selbst noch aktiv ist. Somit wird stattdessen der letzte Commit auf GitHub betrachtet, dieser sollte maximal zwei Wochen zurückliegen.

4.3 Durchführung

Bei der Durchführung der Evalution wird der Bot auf den oben genannten Projekten und mit den vorher beschriebenen Modifikationen ausgeführt. Dazu wird für jedes Projekt eine Konfiguration im Refactoring-Bot erstellt, mit den Daten des GitHub-Repositories und der SonarCloud-Analyse. Mit dieser Konfiguration wird der Bot anschließend ausgeführt. Der Bot entfernt eine bestimmte Anzahl von auskommentierten Codestellen, und erstellt für jede einen neuen Pull-Request auf dem GitHub-Repository.

Nachdem die Pull-Requests erstellt worden sind, wird deren Status nach spätestens drei Wochen überprüft. Sind Pull-Requests zu diesem Zeitpunkt noch nicht angenommen oder abgelehnt, so gelten sie für diese Studie als unbearbeitet.

4.4 Metriken

Für jedes Projekt werden folgende Daten aufgezeichnet:

- Name des Projekts
- Lines of Code
- Anzahl vom Bot erstellter Pull-Requests
- Davon angenommene Pull-Requests
- Davon abgelehnte Pull-Requests

[1] https://sonarcloud.io/explore/projects?sort=-analysis_date

Zudem werden für jedes Projekt folgende Daten zur Größe und Aktivität festgehalten:

- Anzahl mitwirkender Entwickler
- Anzahl Sterne (Nutzer, die das Projekt auf GitHub favorisieren)
- Anzahl bearbeiteter Pull-Requests in den letzten drei Monaten

Zudem werden folgende Daten für jeden Pull-Request festgestellt:

- Status des Pull-Requests (angenommen oder abgelehnt)
- Alter der Codezeile
- Anzahl entfernter Zeilen
- Korrektheit des auskommentierten Codes
- Grund für Ablehnung (falls von Entwicklern angegeben)

4.5 Auswertungsmethode

Zur Auswertung werden die vom Bot erstellten Pull-Requests auf GitHub beobachtet und dokumentiert, ob diese angenommen oder abgelehnt werden. Pull-Requests, die unbearbeitet bleiben, werden im Ergebnis nicht berücksichtigt. Am Ende werden jeweils alle angenommenen und abgelehnten Pull-Requests von allen Projekten aufaddiert, und basierend auf dem Verhältnis eine Akzeptanzquote ermittelt.

Für jeden einzelnen angenommenen oder abgelehnten Pull-Request wird das entsprechende SonarQube-Issue identifiziert, auf Basis dessen das Refactoring durchgeführt wurde. Von diesem wird das Alter und die Länge der auskommentierten Codestelle ermittelt.

5 Ergebnisse

5.1 Beschreibung

5.1.1 Untersuchte Projekte

Name	Lines of Code	Pull-Req.	Ang.	Abg.	Anmerkungen
Dependency Check [1]	26.000	3	0	3	Kein Kommentar
Dependency Track [2]	33.000	3	0	3	Kein Interesse
OnlineCodex [3]	50.000	3	3	0	
JRadio [4]	43.000	2	2	0	
fess-suggest [5]	4.600	2	0	0	
SlideshowFX [6]	20.000	3	0	0	
Lance [7]	13.000	3	0	0	
Survey [8]	2.400	3	0	0	
SIC-API [9]	17.000	2	0	2	Kein Kommentar
Cresco Dashboard [10]	3.100	3	0	0	
Encyclopedia Metallum API [11]	5.200	3	0	3	Kein Kommentar
Clarity [12]	11.000	3	0	3	Kein Kommentar
Powsybl Incubator [13]	6.500	3	0	3	Kein Kommentar
ebdj-liseuse [14]	8.200	2	0	0	
flatpack [15]	4.600	3	2	1	Grund: Zu viel entfernt
Gesamt		41	7	15	

Table 5.1: Übersicht der erstellten Pull-Requests nach Softwareprojekt

[1] https://github.com/jeremylong/DependencyCheck
[2] https://github.com/DependencyTrack/dependency-track
[3] https://github.com/OnlineCodex/OnlineCodex
[4] https://github.com/dernasherbrezon/jradio
[5] https://github.com/codelibs/fess-suggest
[6] https://github.com/twasyl/SlideshowFX
[7] https://github.com/cloudiator/lance
[8] https://github.com/markoniemi/survey
[9] https://github.com/globo-technology/sic-api
[10] https://github.com/CrescoEdge/dashboard
[11] https://github.com/Loki-Afro/metalarchives
[12] https://github.com/skadistats/clarity
[13] https://github.com/powsybl/powsybl-incubator
[14] https://github.com/ngendron/ebdj-liseuse
[15] https://github.com/Appendium/flatpack

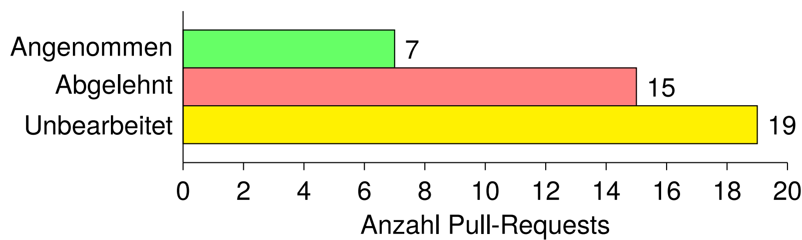

Figure 5.1: Diagramm mit Anzahl der Pull-Requests

Insgesamt wurden 41 Pull-Requests auf 15 verschiedenen Projekten erstellt. Tabelle 5.1 enthält eine Übersicht aller Projekte. Wie man in Abbildung 5.1 sieht, machen unbearbeitete Pull-Requests mit einer Anzahl von 19 (46,3%) den größten Anteil aus. Darauf folgen 15 (36,6%) abgelehnte und lediglich sieben (17,1%) angenommene Pull-Requests. Betrachtet man nur tatsächlich bearbeitete Pull-Requests, wurden sieben von 22 angenommen, was einem Anteil von 31,8% entspricht. Somit wurde der Großteil der bearbeiteten Pull-Requests abgelehnt.

Bei den meisten Projekten blieben die Pull-Requests von den Entwicklern unkommentiert, sowohl bei angenommenen als auch bei abgelehnten. Lediglich ein Entwickler von *flatpack* hinterließ Kommentare zu den Pull-Requests, in denen er konkrete Begründungen für die Annahme oder Ablehnung angab.

Name	Contributors	Sterne	Pull-Requests bearbeitet (letzte 3 Monate)	Ang./Abg.
Dependency Check	105	1582	49	0/3
Dependency Track	7	288	4	0/3
OnlineCodex	5	6	32	2/0
JRadio	1	20	2	2/0
fess-suggest	3	5	1	0/0
SlideshowFX	1	41	0	0/0
Lance	5	1	27	0/0
Survey	1	6	0	0/0
SIC-API	2	0	17	0/0
Cresco Dashboard	2	0	0	0/0
Encyclopedia Metallum API	1	10	0	0/3
Clarity	9	378	0	0/3
Powsybl Incubator	8	2	43	0/3
ebdj-liseuse	1	0	0	0/0
flatpack	5	39	3	2/1

Table 5.2: Größe der Projekte

Tabelle 5.2 enthält weitere Informationen zu den untersuchten Projekten, um deren Größe und Aktivität besser einschätzen zu können. Es wurden sowohl große als auch kleine Projekte untersucht. Einige haben nur einen einzigen Entwickler, während andere Beiträge von vielen verschiedenen Leuten enthalten. Auch die Anzahl der von den Entwicklern bearbeiteten Pull-Requests in den

letzten drei Monaten ist recht unterschiedlich. Bei einigen Projekten wurden in diesem Zeitraum gar keine Pull-Requests bearbeitet, während andere im gleichen Zeitraum bis zu 49 bearbeitet haben. Bei den drei populärsten Projekten (Dependency Check, Dependency Track, Clarity) wurden alle Pull-Requests abgelehnt, während angenommene Pull-Requests eher bei kleineren Projekten zu finden sind.

5.1.2 Eigenschaften von Pull-Requests

Projekt	Korrektheit	Länge (Zeilen)	Alter (Tage)	Status
Dependency Check	Korrekt	8	4	Abgelehnt
	Korrekt	1	1882	Abgelehnt
	Korrekt	8	279	Abgelehnt
Dependency Track	Korrekt	1	16	Abgelehnt
	Korrekt	1	16	Abgelehnt
	Korrekt	1	16	Abgelehnt
OnlineCodex	Korrekt	1	15	Angenommen
	Korrekt	1	15	Angenommen
	Korrekt	1	28	Angenommen
JRadio	Korrekt	1	12	Angenommen
	Korrekt	3	12	Angenommen
fess-suggest	Korrekt	9	1441	Unbearbeitet
	Korrekt	5	1292	Unbearbeitet
SlideshowFX	Korrekt	8	1309	Unbearbeitet
	Korrekt	5	1447	Unbearbeitet
	Korrekt	1	779	Unbearbeitet
Lance	Korrekt	109	265	Unbearbeitet
	Korrekt	8	265	Unbearbeitet
	Korrekt	1	265	Unbearbeitet
Survey	Korrekt	4	924	Unbearbeitet
	Korrekt	1	924	Unbearbeitet
	Korrekt	1	924	Unbearbeitet
SIC-API	Korrekt	5	419	Abgelehnt
	Korrekt	52	132	Abgelehnt
Cresco Dashboard	Korrekt	13	212	Unbearbeitet
	Korrekt	13	212	Unbearbeitet
	Korrekt	13	212	Unbearbeitet
Encyclopedia Metallum API	Korrekt	1	748	Abgelehnt
	Korrekt	1	748	Abgelehnt
	Korrekt	3	748	Abgelehnt
Clarity	Korrekt	6	1294	Abgelehnt
	Korrekt	1	915	Abgelehnt
	Korrekt	1	360	Abgelehnt
Powsybl Incubator	Korrekt	4	16	Abgelehnt
	Korrekt	5	1	Abgelehnt
	Korrekt	1	16	Abgelehnt
ebdj-liseuse	Korrekt	1	455	Unbearbeitet
	Korrekt	7	609	Unbearbeitet
flatpack	Korrekt	1	142	Angenommen
	Korrekt	1	142	Angenommen
	Korrekt	4	238	Abgelehnt

Table 5.3: Übersicht der bearbeiteten Issues

Wie man Tabelle 5.3 entnehmen kann, variieren die entfernten Stellen von auskommentiertem Code in Alter und Länge. Der neueste Kommentar war einen Tag alt, während der älteste 1882 Tage (ca. fünf Jahre) alt war. Die kürzesten Kommentare enthielten nur eine einzige Codezeile, der längste erstreckte sich über 109 Zeilen.

Von den durchgeführten Refactorings war keines fehlerhaft.

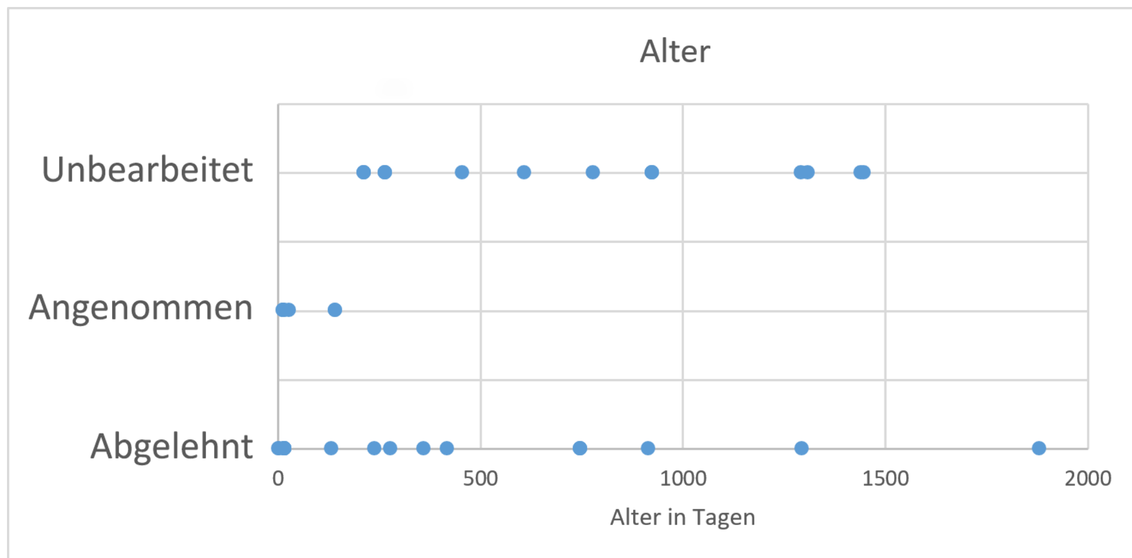

Figure 5.2: Alter und Akzeptanz von Pull-Requests

Abbildung 5.2 zeigt das Alter des bearbeiteten Issues und den Status des dazugehörigen Pull-Requests. Wie man sieht, wurde das älteste angenommene Refactoring bei einem 142 Tage alten Issue durchgeführt, während einige abgelehnte Refactorings auf wesentlich älteren Issues durchgeführt wurden. Somit ist das Durchschnittsalter bei angenommenen Refactorings deutlich geringer, was bei dieser geringen Datenmenge jedoch nicht statistisch aussagekräftig ist.

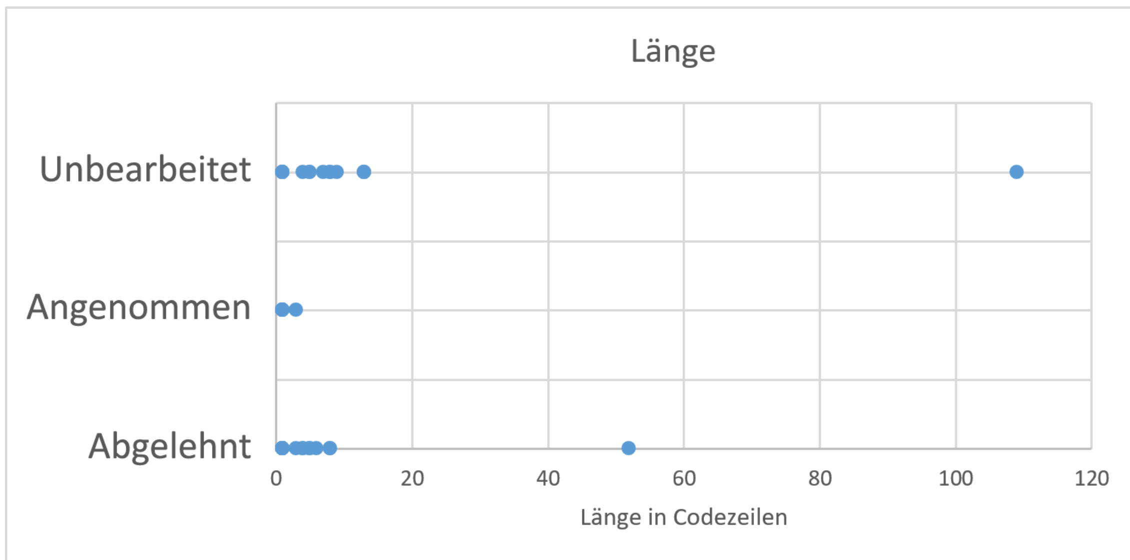

Figure 5.3: Länge und Akzeptanz von Pull-Requests

Abbildung 5.3 zeigt den Zusammenhang zwischen der Anzahl der entfernten Codezeilen und der Akzeptanz der Pull-Requests. Abgelehnte Pull-Requests enthielten tendenziell mehr Codezeilen als angenommene. Die längste entfernte Codestelle umfasste 109 Zeilen, die kürzesten Refactorings betrafen nur einzelne Zeilen.

Bei abgelehnten Pull-Requests enthalten die Refactorings tendenziell mehr Codezeilen als bei angenommenen, das längste abgelehnte Refactoring entfernte 52 Zeilen von auskommentiertem Code.

5.2 Diskussion

In Abschnitt 5.2.1 werden die Ergebnisse bezüglich Forschungsfrage 1 diskutiert, in 5.2.2 Forschungsfrage 2 und in Abschnitt 5.2.3 Forschungsfrage 3.

5.2.1 Akzeptanz

Die Rate der akzeptierten Pull-Requests fällt mit 31,8% eher gering aus. Bei den meisten abgelehnten Pull-Requests wurde kein Kommentar angegeben, weswegen die Bewertung dieses Ergebnisses größtenteils auf eigenen Erfahrungen und Spekulation beruht.

Nach manueller Überprüfung aller erstellten Pull-Requests wurde festgestellt, dass keine davon fehlerhaft waren oder die Kompilierbarkeit des Codes beeinträchtigt haben. Ein fehlerhaftes Durchführen von Refactorings durch den Bot kann somit als Ablehnungsgrund ausgeschlossen werden. Stattdessen werden die folgenden Hypothesen als mögliche Gründe in Betracht gezogen.

Hypothese: Pull-Requests von Bots haben eine niedrigere Akzeptanz als von Menschen erstellte

Ein großer Faktor bei der Ablehnung vieler Pull-Requests könnte die kritische Betrachtung von automatisch generierten Code-Änderungen sein. Durch die in 4.2 beschriebene Nachricht, die jedem Pull-Request angehängt wurde, wird den Entwicklern deutlich gemacht, dass es sich um automatisch generierte Refactorings handelt. Dadurch könnte ein gewisses Misstrauen entstehen, da die Änderungen nicht von einem Menschen durchgeführt wurden, und deshalb aus Sicht der Entwickler vielleicht unsinnig oder fehlerhaft sein könnten. Murgia et al. [MJDV16] führten eine Studie durch, bei der die Reaktionen auf automatisch generierte Antworten auf der Seite Stack Overflow [16] getestet wurden. Stack Overflow ist eine Frage-und-Antwort-Plattform für verschiedene Themen der Softwareentwicklung. Für die Studie wurde ein Bot entwickelt, der automatisch Antworten auf Fragen bezüglich der Behebung von Git-Fehlermeldungen gibt. Beim ersten Durchlauf wurde vorgetäuscht, dass der Bot ein menschlicher Nutzer sei, beim zweiten Mal wurde deutlich gemacht, dass es sich um maschinell generierte Antworten handelt. Der als menschlicher Nutzer getarnte Bot erhielt dabei öfters positives Feedback (sogenannte *upvotes*) und konnte 90 Tage lang aktiv bleiben. Im zweiten Durchlauf wurde der als Bot gekennzeichnete Account innerhalb von 25 Tagen gebannt, und erhielt nie positives Feedback für seine Antworten, obwohl die Software zum Generieren der Antworten in beiden Durchläufen gleich war [MJDV16]. Basierend auf den Ergebnissen dieser Studie liegt es nahe, dass das Misstrauen gegenüber Bots auch bei der Evaluation des Refactoring-Bots eine Rolle gespielt haben könnte. So wurden häufig Pull-Requests kommentarlos abgelehnt, und in einem Fall der Bot sogar direkt von einem Repository gebannt.

Für zukünftige Studien mit dem Refactoring-Bot wäre es daher interessant, eine ähnliche Methode wie Murgia et al. [MJDV16] zu verwenden. Da der Refactoring-Bot mit einem beliebigen GitHub-Account verbunden werden kann, könnte man auch diesen als menschlichen Nutzer tarnen, indem man einen echt wirkenden Namen erfindet und ein menschliches Profilbild verwendet. Außerdem müsste man eine Nachricht für Pull-Requests wählen, die so wirkt, als käme sie von einem echten Nutzer.

Zusätzlich ist es bei vielen der untersuchten Projekte so, dass alle bisherigen Beiträge von der Entwicklergruppe selbst stammen, und es entweder wenige oder gar keine Pull-Requests von anderen Personen gab. Dies könnte darauf hindeuten, dass viele Entwickler GitHub nur benutzen, um intern in ihrem Team zu arbeiten, jedoch kein Interesse an Beiträgen von fremden Leuten haben.

Hypothese: Die Unterteilung in einzelne Pull-Requests ist unerwünscht

Ein weiteres Problem der derzeitigen Implementierung könnte sein, dass Entwickler ihre Versionshistorie sauber halten wollen, der Bot jedoch für jede einzelne Änderung einen separaten Commit, Branch, und Pull-Request erstellt. Dies ist auf GitHub eher unüblich, es ist eher gebräuchlich, alle durchgeführten Änderungen in einem Pull-Request zusammenzufassen.

[16] https://stackoverflow.com/

Durch die Generierung von mehreren Pull-Requests erzeugt man mehr Aufwand für die Entwickler, da jeder einzeln bearbeitet werden muss. In SonarQube-Analysen kommen teilweise hunderte oder tausende von Issues vor, und es wäre nicht praktikabel, als Entwickler diese Anzahl von Pull-Requests zu bearbeiten. Auf der GitHub-Webseite müssen diese Pull-Requests nämlich einzeln bearbeitet und anschließend gemerged werden.

Außerdem wird die Versionshistorie des Projekts bei vielen einzelnen Pull-Requests schnell unübersichtlich, da oftmals nur einzelne Zeilen entfernt werden, wodurch es zu einer großen Anzahl von einzelnen Änderungen kommt.

In einer ähnlichen Studie, bei der ebenfalls automatische Refactorings als Pull-Requests erstellt wurden, wurde eine höhere Akzeptanzquote von 44,4% erzielt [DCM+18]. Ein ausschlaggebender Faktor könnte gewesen sein, dass dort nur ein Pull-Request erstellt wurde, der das Refactoring an allen geeigneten Stellen im gesamten Projekt angewendet hat. Entwickler könnten dadurch eher geneigt sein, den Pull-Request anzunehmen, da die Änderungen umfangreicher sind.

Eine Lösung für dieses Problem wäre, dem Bot die Funktionalität hinzuzufügen, mehrere oder sogar alle Issues einer SonarQube Analyse auf einmal zu bearbeiten. Dadurch würde man den Arbeitsaufwand für Entwickler deutlich reduzieren, und eine übersichtliche Versionshistorie behalten.

Hypothese: Manchmal sollen nur Teile des auskommentierten Codeblocks entfernt werden

Einer der Pull-Requests wurde mit dem Hinweis "*trying to delete too much.*" abgelehnt. Es handelt sich dabei um insgesamt vier Kommentarzeilen, davon enthielt eine auskommentierten Code, und drei weitere waren Teil eines dazugehörigen Kommentars. Während der Evaluation arbeitete der Bot so, dass ausgehend von einer von SonarQube angegebenen Zeilennummer alle zusammenhängenden Zeilenkommentare gelöscht werden. Die Annahme ist hier, dass Kommentare, die sich direkt über oder unter auskommentiertem Code befinden, zu diesem gehören. Damit wären es unsinnig, diese im Code zu belassen. Dieses Verhalten scheint aber nicht in allen Situationen erwünscht zu sein.

Letztendlich wurde entschieden, den Refactoring-Bot dennoch alle zusammenhängenden Kommentare unter der von SonarQube gelieferten Zeile entfernen zu lassen. Dies mag zwar in einigen wenigen Fällen ungünstig sein, ist aber allgemein dennoch die bessere Lösung. Es wurde versucht, den Refactoring-Bot normale Kommentare von auskommentiertem Code unterscheiden zu lassen, dies stellte sich aber als unzuverlässig heraus. Außerdem kam es mit dieser Lösung zu vielen "verwaisten" Kommentaren, die sich auf entfernten auskommentierten Code bezogen. Somit würde es keinen Sinn machen, diese alleine stehen zu lassen.

5.2.2 Korrelation von Akzeptanz und Alter/Länge der Codestelle

Aufgrund der hohen Anzahl von kommentarlos abgelehnten und unbearbeiteten Pull-Requests lässt sich hier keine statistisch aussagekräftige Messung durchführen. Dafür ist die Datenmenge zu gering.

Es wird empfohlen, den Refactoring-Bot auskommentierten Code in jedem Fall entfernen zu lassen, unabhängig von Alter und Länge. Wie in Kapitel 2 bereits besprochen wurde, ist auskommentierter Code eine schlechte Praxis und sollte nie in einem Versionskontrollsystem eingecheckt werden. Somit ist es sinnvoll, diesen in jedem Fall zu entfernen.

5.2.3 Korrektheit

Von den 41 durchgeführten Refactorings war keines fehlerhaft, das heißt in jedem Fall war der Code nach Durchführung des Refactorings weiterhin kompilierbar. Dies war zu erwarten, da der Bot mit Hilfe der *JavaParser* Bibliothek Codestellen identifizieren kann, die tatsächlich Kommentare enthalten, und auch nur diese entfernt. Somit wird sichergestellt, dass nie die Funktionsfähigkeit des Codes beeinträchtigt wird. Selbst im Ausnahmefall, wenn eine SonarQube Analyse angegeben wird, die nicht zur aktuellen Codeversion passt und für Code Smells andere Zeilennummern liefert, würden schlimmstenfalls fälschlicherweise andere Kommentare gelöscht.

5.3 Limitationen

Die zuvor beschriebenen Ergebnisse sollten unter Berücksichtigung einiger Einschränkungen betrachtet werden. Diese werden im Folgenden beschrieben.

5.3.1 Das Entfernen von auskommentiertem Code ist im Gegensatz zu anderen Refactorings nicht immer erwünscht

Die Ergebnisse dieser Studie sind nicht unbedingt auf andere Arten von Refactorings übertragbar, da das Entfernen von auskommentiertem Code für manche Entwickler unerwünscht sein könnte. Wie in Kapitel 2 erläutert, ist das Auskommentieren von Code eine schlechte Praxis, für die es bereits genügend bessere Alternativen gibt. Jedoch kann es sein, dass manche Entwickler diese Alternativen nicht nutzen können oder wollen, da sie ihre Arbeitsweise nicht umstellen wollen. Somit ist nicht immer eindeutig, ob das Refactoring durchgeführt werden soll oder nicht, da manche Entwickler in auskommentiertem Code kein Problem sehen.

Es ist daher anzunehmen, dass bei anderen Refactorings eine höhere Akzeptanzrate erzielt werden kann, wenn der Bedarf nach einem Refactoring eindeutiger ist. Beispielsweise unterstützt der Refactoring-Bot auch das Hinzufügen einer fehlenden Override-Annotation. Eine Override-Annotation sollte immer verwendet werden, wenn eine Methode einer Superklasse überschrieben wird. Lässt man diese Annotation weg, so verschlechtert sich die Lesbarkeit des Codes, und die Methode wird nicht korrekt vom Compiler überprüft. In diesem Fall gibt es keinerlei Gründe dafür, das Refactoring nicht durchzuführen, wodurch die Akzeptanz wahrscheinlich höher wäre als beim Entfernen von auskommentiertem Code.

5.3.2 Nur Projekte mit einer Analyse auf SonarCloud wurden betrachtet

Die in dieser Studie ausgewählten Projekte sind nicht unbedingt allgemein repräsentativ, da immer vorausgesetzt wurde, dass eine auf SonarCloud gehostete Code-Analyse vorhanden war. Daher ist davon auszugehen, dass die Entwickler dieser Softwareprojekte bereits ein gewisses Interesse an der Erkennung und Behebung von Code Smells haben. Eventuell lassen sich andere Ergebnisse erzielen, wenn Projekte von GitHub ausgewählt werden und die SonarQube-Analyse selbst durchgeführt wird.

5.3.3 Geringe Anzahl an bearbeiteten Pull-Requests

Durch die geringe Anzahl von tatsächlich bearbeiteten Pull-Requests sind die Ergebnisse nicht unbedingt repräsentativ, da es hier eine hohe Varianz gibt. Letztendlich wurden 22 Pull-Requests bearbeitet, während 19 innerhalb von drei Wochen unbeantwortet blieben.

5.3.4 Generelle Ablehnung von Bots oder fremden Beiträgen

Es besteht die Möglichkeit, dass viele der Pull-Requests nicht wegen ihres Inhalts abgelehnt wurden, sondern weil sie von einem Bot erstellt wurden. Leider blieben die meisten abgelehnten Pull-Requests unkommentiert, weswegen man diesbezüglich nur spekulieren kann. Wie bereits in 5.2.1 erläutert, besteht bei Menschen generell ein größeres Misstrauen gegenüber Bots. Bei manchen Projekten kann es sogar sein, dass gar keine Beiträge von außenstehenden Personen erwünscht sind. Bei einigen der untersuchten Projekte gab es wenige oder gar keine Beiträge von Personen, die nicht zum Entwicklerteam gehören. Hier ist auch anzumerken, dass Repositories bei GitHub grundsätzlich öffentlich sind, bei einem privaten Repository muss man ein kostenpflichtiges Abonnement abschließen oder ist auf drei mitwirkende Entwickler beschränkt. Auch ist es manchmal so, dass ein öffentliches Repository hauptsächlich dazu da ist, anderen die Nutzung der Software zu ermöglichen, aber Beiträge zum Quellcode nicht unbedingt willkommen sind. Daher ist die Existenz eines öffentlichen Repositories nicht unbedingt ein Zeichen dafür, dass Beiträge von fremden Leuten tatsächlich erwünscht sind.

Für eine zukünftige Evaluation würde es sich anbieten, Entwickler von Projekten vorher direkt zu kontaktieren und deren Kooperation zu erbitten. So kann sichergestellt werden, dass die Refactorings tatsächlich angesehen und nach ihrer Qualität beurteilt werden.

6 Fazit

Auskommentierter Code ist ein häufiges Problem in Softwareprojekten und erschwert oftmals die Lesbarkeit des Codes. Es wurde aufgezeigt, dass heutzutage genügend bessere Alternativen existieren, sodass man als Entwickler nicht auf auskommentierten Code zurückgreifen sollte. Daher sollten solche Codestellen immer entfernt werden. Mit der Implementierung im Refactoring-Bot wurde gezeigt, dass auskommentierter Code zuverlässig und restlos entfernt werden kann. Mit Hilfe von SonarQube werden entsprechende Codestellen nahezu fehlerfrei ermittelt und können anschließend bereinigt werden.

Nach der Implementierung wurde eine Evaluation durchgeführt, bei der der Refactoring-Bot auf realen Projekten ausgeführt wurde und auskommentierten Code entfernt hat. Diese Änderungen wurden dann den jeweiligen Entwicklern als Pull-Request vorgeschlagen. Dabei zeigte sich, dass das Entfernen von auskommentiertem Code zwar fehlerfrei funktioniert hat, Entwickler jedoch trotzdem zögerlich waren, diese Refactorings anzunehmen. Um den Bot in der Praxis einsetzbar zu machen, bedarf es deswegen noch einiger Verbesserungen. Insbesondere die Bereitstellung von jedem Refactoring als einzelner Pull-Request erhöht den Arbeitsaufwand enorm, weswegen man in Erwägung ziehen sollte, diese stattdessen zu bündeln.

Ausblick

Der Refactoring-Bot ist nun in der Lage, auskommentierten Code automatisch zu entfernen. Damit wurde er um ein wichtiges und häufig vorkommendes Refactoring erweitert. Um den Refactoring-Bot weiter zu verbessern, sollte dieser um neue Refactorings erweitert werden, sodass er die meisten SonarQube *Issues* abdeckt und somit die Codequalität deutlich erhöhen kann. Außerdem sollte die Grundfunktionalität in einigen Punkten verbessert werden.

Bei der Implementierung neuer Refactorings sollte man sich zunächst auf einfache Refactorings konzentrieren, die gefahrlos automatisch durchgeführt werden können. Außerdem könnte man sich die Häufigkeit von Issues auf SonarCloud ansehen, um zu prüfen, welche Refactorings den größten Nutzen hätten.

Eines der größten Probleme ist die Tatsache, dass der Refactoring-Bot für jedes Refactoring einen neuen Pull-Request erstellt. Bei größeren Projekten mit mehreren tausend Issues ist dies nicht mehr praktikabel, da ein Entwickler jeden Pull-Request einzeln bearbeiten müsste. Besser wäre es, mehrere Refactorings in einem Pull-Request zusammenzufassen. Beispielsweise könnten man alle Refactorings einer bestimmten Art in einen Pull-Request bündeln, wie zum Beispiel sämtlicher auskommentierter Code, der entfernt wurde. Außerdem sollte der Bot mehrere Änderungen in einem Commit zusammenfassen können, da zu viele einzelne Commits die Versionshistorie eines Projekts schwerer lesbar machen. Sollte diese Änderung tatsächlich implementiert werden, sollte vorher sichergestellt werden, dass alle Refactorings fehlerfrei durchgeführt werden. In [DCM+18] wird

vorgeschlagen, einfache Refactorings in einem Pull-Request zusammenzufassen, für komplexere Refactorings jedoch einen separaten Pull-Request für jede Änderung zu erstellen. So könnte man für simple Refactorings wie das Hinzufügen einer Override-Annotation alle Änderungen im gesamten Projekt in einem Pull-Request bündeln, da es sich um ein sehr einfaches Refactoring handelt, und es sehr unwahrscheinlich ist, dass der Bot dabei Fehler macht. Außerdem gibt es hier keinen sinnvollen Grund, das Refactoring nicht durchzuführen. Für ein komplexeres Refactoring wie Extract Method, welches dem Bot in Zukunft hinzugefügt werden soll, wäre es hingegen besser, jede Änderung in einem einzelnen Pull-Request zur Verfügung zu stellen. Die Entwickler können dann entscheiden, ob sie das Refactoring an der Stelle tatsächlich für sinnvoll halten, oder ob sie es anders oder gar nicht durchführen wollen.

Literaturverzeichnis

[DCM+18] R. Dantas, A. Carvalho, D. Marcílio, L. Fantin, U. Silva, W. Lucas, R. Bonifácio. "Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate Java programs." In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Mar. 2018, pp. 497–501. DOI: 10.1109/SANER.2018.8330247 (cit. on pp. 38, 41).

[Dod17] K. C. Dodds. *Please, don't commit commented out code*. 2017. URL: http://archive.is/pZLp6 (visited on 11/06/2018) (cit. on p. 16).

[FMPZ15] F. A. Fontana, M. Mangiacavalli, D. Pochiero, M. Zanoni. "On Experimenting Refactoring Tools to Remove Code Smells." In: *Scientific Workshop Proceedings of the XP2015*. XP '15 workshops. Helsinki, Finland: ACM, 2015, 7:1–7:8. ISBN: 978-1-4503-3409-9. DOI: 10.1145/2764979.2764986. URL: http://doi.acm.org/10.1145/2764979.2764986 (cit. on p. 20).

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999. ISBN: 0-201-48567-2 (cit. on pp. 13, 18).

[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, pp. 477–481. DOI: 10.1109/SANER.2017.7884657 (cit. on p. 21).

[MJDV16] A. Murgia, D. Janssens, S. Demeyer, B. Vasilescu. "Among the Machines: Human-Bot Interaction on Social Q&A Websites." In: *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. CHI EA '16. San Jose, California, USA: ACM, 2016, pp. 1272–1279. ISBN: 978-1-4503-4082-3. DOI: 10.1145/2851581.2892311. URL: http://doi.acm.org/10.1145/2851581.2892311 (cit. on p. 37).

[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*. WRT '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 41–44. ISBN: 978-1-4503-0579-2. DOI: 10.1145/1984732.1984742. URL: http://doi.acm.org/10.1145/1984732.1984742 (cit. on p. 21).

[MT04] T. Mens, T. Tourwe. "A survey of software refactoring." In: *IEEE Transactions on Software Engineering* 30.2 (Feb. 2004), pp. 126–139. ISSN: 0098-5589. DOI: 10.1109/TSE.2004.1265817 (cit. on p. 18).

[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)*. Sept. 2015, pp. 253–258. DOI: 10.1109/SCAM.2015.7335422 (cit. on p. 21).

[Spi12] D. Spinellis. “Git.” In: *IEEE Software* 29.3 (May 2012), pp. 100–101. ISSN: 0740-7459. DOI: 10.1109/MS.2012.61 (cit. on p. 19).

[Sta11] Stackexchange. *Is commented out code really always bad?* 2011. URL: http://archive.today/wnpw1 (visited on 11/06/2018) (cit. on p. 17).

[TCC18] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou. “Ten years of JDeodorant: Lessons learned from the hunt for smells.” In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Mar. 2018, pp. 4–14. DOI: 10.1109/SANER.2018.8330192 (cit. on p. 20).

[YM12] Y. S. Yoon, B. A. Myers. “An exploratory study of backtracking strategies used by developers.” In: *2012 5th International Workshop on Co-operative and Human Aspects of Software Engineering (CHASE)*. June 2012, pp. 138–144. DOI: 10.1109/CHASE.2012.6223012 (cit. on p. 15).

Alle URLs wurden zuletzt am 10. 04. 2019 geprüft.

Erklärung

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

Ort, Datum, Unterschrift