Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Implementation of an Automatic Extract Method Refactoring

Johannes Hubert

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	M.Sc. Marvin Wyrich
Commenced:	October 17, 2018
Completed:	April 17, 2019

Abstract

Software quality is an important aspect to guarantee maintainability and comprehensibility of developed source code. Modern software projects use static code analysis tools to continuously monitor the software quality. Based on findings from these tools, developers refactor their code with the aim to remove detected code smells. Refactorings can use up a lot of resources when done manually and a lot of authors suggest semi-automated solutions to improve the refactoring experience for developers. One of the most applied refactorings is the extract method refactoring which is often used to improve long and complex methods. Recent studies showed, that existing semi-automated tools for this refactoring are not preferred by developers. We propose an approach to fully automate the extract method refactorings based on findings from static code analysis tools. Our approach finds refactoring opportunities in a selected method and ranks these candidates according to a scoring function. The highest ranked candidate will be automatically refactored using the extract method refactoring. We implement our approach using the existing Refactoring Bot framework, a software development bot which seamlessly integrates into the build pipeline of existing projects.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Abbreviations

AST abstract syntax tree. 21

CDFG control and data flow graph. 27

CFG control flow graph. 21

DSD data and structure dependency graph. 28

IDE integrated development environment. 17

LCOM lack of cohesion in methods. 31

LoC lines of code. 33

NPE null pointer exception. 21

PDG program dependence graph. 29

SEMI SRP-based extract method identification. 30

SRP single responsibility principle. 30

1 Introduction

This chapter gives an introduction to refactoring in general followed by the research objective of this work. Furthermore it describes the approach and base the implementation is constructed on. The chapter concludes with a description of the structure of this master thesis and a short description of each chapter.

1.1 Motivation

The complexity and size of software products is growing continuously and maintaining these products is already the most cost consuming part in software development [MT04]. Software quality and therefore maintainability decays even more over the course of development if it is not highly prioritized [EGK+01; Wag]. In reality, however, software development projects are often more focused on delivering working features than preserving a high level of software quality due to strict deadlines [FBB+99].

This makes it important that developers have tools to control and improve the quality of their code with little effort. A great number of tools to measure code quality and find code smells exists[DJH+08; WKK07]. These tools can be easily integrated into existing projects and usually use static code analysis to find quality problems in existing code. Refactoring, the process of improving the quality of a software product without changing the functionality [FBB+99], is a widely used method to approach quality issues in software projects. Refactoring aims to make code more understandable, easier to maintain, and easier to update. There are some tools which assist developers in refactoring [MLCP10] but most of the work still has to be done manually. Most of the existing tools to help with Refactoring are integrated into appropriate integrated development environments (IDEs) and can do refactorings for a specific subset of refactoring rules after the developer has detailed how and where a selected refactoring rule should be applied. Performing refactorings manually or semi-automated is complex, time consuming, error-prone, and tedious [Opd92].

Martin Fowler introduced a catalog of refactorings naming and describing most of the commonly applied refactorings [FBB+99]. “Extract method”, for example, extracts code into a new method to reduce the length of methods or improve the comprehensibility [FBB+99]. A research study which analyzed over one million commit messages in open source projects showed that the top three refactorings performed by developers are *rename*, *move method/class* and *extract/inline method* [Bra17]. Extract method is also the refactoring with the highest amount of different reasons why developers perform this refactoring [STV16] and therefore a common solution to tackle different problems in the code. Software quality and code smells are not always the reason for applying an extract method refactoring but are one of the main reasons why this refactoring is performed according to Silva et al. [STV16]. Although extract method and other refactorings are used

frequently, roughly 75% are implemented completely manually [STV16] and only 25% use the help of semi-automatic refactoring tools. Due to the lack of fully automated refactoring tools, these refactorings always require time and work from developers. Fully automated refactoring tools which find refactoring candidates and automatically apply the suitable refactoring could save a lot of time and help the developers to focus on other programming tasks.

1.2 Research Objectives and Contributions

To automate refactoring, a tool named Refactoring Bot exists that performs refactorings based on SonarQube analysis and creates merge requests for each refactoring done. The bot is rule-based and currently only supports *missing override annotation*, *unused method parameter*, *unused variable* and *wrong modifier order*. The objective of this work is to implement an extract method refactoring for the bot based on a SonarQube analysis and to evaluate it. The two main objectives can be broken down to following questions:

1. How can a refactoring that automatically performs an extract method refactoring on methods that are too long be implemented?
    a) What is the best algorithm to find candidates for extraction in a method? Are there different algorithms and how do different algorithms compare to each other?
    b) How can extract method candidate be evaluated? Can this evaluation be used to choose a suitable candidate?
    c) Which parameters should be used in an extracted method?
    d) Can extracted methods automatically be named including its parameters?
    e) Can it be guaranteed that the generated code always compiles?
2. Can the implemented refactoring successfully improve code with methods that are too long?
    a) How should an automatic extract method bot be evaluated?
    b) Are merge requests for extract method refactorings by the bot generally accepted in open source projects?
    c) Is there a significant difference between what a human would do to refactor long methods compared to the implemented algorithm?
    d) What is the generally accepted length of methods before they should be refactored?

1.3 Methodological Approach

To implement an automatic extract method refactoring we performed a literature research. We found the best solution for our project and improved it so that it met all of the requirements. This solution was implemented using the existing Refactoring Bot framework to automatically perform an extract method refactoring based on quality findings provided through the framework. We conducted a study through GitHub to evaluate our approach and to explore the acceptance of automatic refactorings in open-source projects.

1.4 Structure of the Work

Chapter 2: Fundamentals This chapter introduces important terminologies and concepts which are fundamental for understanding this work. This includes an overview of static code analysis and specifically SonarQube. The refactoring bot, which was used to implement the extract method refactoring, is presented in detail. At the end extract method refactoring is described in detail.

Chapter 3: Related Work In this chapter different approaches to implement fully automated code refactorings are discussed and evaluated. It is mainly focused around extract method refactoring and finding the best candidate for a method. To evaluate existing work, requirements are introduced which are used to assess different approaches including this work. We compare existing approaches and elaborate what differentiates them from our approach.

Chapter 4: Implementation This chapter describes the approach which we chose to implement the extract method refactoring in this work and also includes the practical implementation in Java based on the refactoring bot developed in a previous work.

Chapter 5: Evaluation The implemented refactoring is evaluated in this chapter. It describes the procedure of the evaluation as well as the reasons behind it. The results are presented and discussed thoroughly.

Chapter 6: Conclusion and Outlook The last chapter gives a short conclusion about the implemented extract method refactoring and the evaluation results. Possible problems with this implementation and topics for further research are discussed as the tail of this work.

2 Fundamentals

This section explains fundamental terms and concepts which are necessary to understand this work. We provide information about extract method refactoring and static code analysis followed by introducing some tools used in this work. At the end of this chapter the Refactoring Bot and its development state to this date are explained.

2.1 Static Code Analysis

Software and developed code often have quality issues or defects which are not caught by a compiler [Lou06]. The first tool developed to find additional defects in code was Lint, a tool to review C code [Joh78]. After that, many tools were created for different languages[1] and have quickly become an important part for software development and quality control for every developer. Such tools usually check the source code for predefined patterns and rules indicating a potential defect. Because they analyze the source code without executing it compared to test suites like JUnit [Lou05], this process is called static code analysis. Static code analysis findings can be divided into three categories: runtime errors, security issues and logical inconsistencies [AHM+08]. Modern static code analysis tools also point out common software quality issues and design flaws to prevent defects in advance.

Static code analysis tools can analyze the plain source code or compiled bytecode. There exist tools for both, bytecode analysis is much faster and sometimes needed for some checks while plain source code analysis can better pinpoint problematic statements. A static code analysis tool builds various graphs representing the source code. Most common source code representations are abstract syntax trees (ASTs) and control flow graphs (CFGs).

2.1.1 Abstract Syntax Trees

An AST is an abstract representation of the structure of uncompiled source code in form of a tree. A node in the AST represents a single word from the source code. Usually it doesn't contain format information and other unnecessary information to compile the software such as comments in the code. ASTs are a widely used data structure in compilers and are also commonly used for static code analysis.

[1] for example CheckStyle (http://checkstyle.sourceforge.net/), FindBugs (http://findbugs.sourceforge.net/) or SonarQube (https://www.sonarqube.org/)

Listing 2.1 Example method before an extract method refactoring was applied.

```
1  void printRequest() {
2      printHeader();
3
4      // print details
5      System.out.println("url: " + url);
6      System.out.println("body: " + getBody());
7  }
```

Listing 2.2 Example method after an extract method refactoring was applied.

```
1  void printRequest() {
2      printHeader();
3      printDetails(getBody());
4  }
5
6  void printDetails(string body) {
7      System.out.println("url: " + url);
8      System.out.println("body: " + body);
9  }
```

2.1.2 Control Flow Graphs

CFGs are a graphical representation of results from control flow analysis [All70]. A control flow analysis is an important part of every compiler to optimize code and is used in static code analysis to find defects like potential null pointer exceptions (NPEs). Creating an AST is a commonly used intermediate step when a CFG should be created for some source code. Nodes in a CFG correspond to one or more statements of source code that are always executed together, in order. The control flow between statements is represented by directed edges between nodes. Statements splitting the control flow to multiple successors are control flow altering statements like `if`-statements or `for`-loops.

2.2 Extract Method Refactoring

Extract method refactoring is an often used refactoring method which can solve many design and quality issues in software. To apply an extract method refactoring, a subset of statements of a method is taken out of the original method and used to create a new method. The new method has to be called in the same place, where the sub statements of the original method were extracted. An example for an extract method refactoring is given in Listing 2.1 and Listing 2.2.

A selected subset of statements in a method can only be extracted when it meets certain conditions. Java, as many other languages, can only have zero or one return parameter, therefore the part to be extracted must not assign more than one local variable or parameter which is read in the code after the subset of statements. In addition, the control flow must exit the subset of statements only in one way. This means that the extractable part must not have a return statement when there is the possibility that code folowing the extracted part might still be executed in some cases in the original method. It also includes that only complete block statements can be extracted. Block Statements are

statements which have child statements, for example `if`-statements or `for`-loops. If a block statement is extracted without all of its child statements, control flow would exit the extracted part at multiple points.

An extracted method might have a return type if a local variable, which is assigned in the extracted method but used in the original method after the extracted statements, exists. The return type of the extracted function then equals that variable's type. Additionally the return value of the extracted method must be assigned to the local variable in the original method. An extracted method can also have one ore more parameters. Parameters are determined by the local variables and parameters used in the extracted method.

D. Silva describes multiple reasons why this refactoring is used [STV16]. According to him the extract method refactoring is used for:

- extracting code so it can be reused
- introducing alternative method signatures
- increasing the testability by creating smaller test units
- increasing the readability of source code by decreasing the length of methods
- removing duplicate code in two or more methods
- reducing the cognitive complexity

among other things. Most of the causes can be detected by static code analysis. We focus in this work solely on methods that are too long. Long methods are easily detected by static code analysis.

Some basic terms we will use are original method, refactoring candidate[2] and remainder or remaining method. The original method is always the method before any refactoring was applied as seen in Listing 2.1. A refactoring candidate is a specific list of substatements in the original method, which should be extracted in an extract method refactoring. The refactoring candidate for the extraction made in Listing 2.2 would be all statements between the lines 4 and 6 in the original method. The subset of statements in the original method, containing each statements which is not in the refactoring candidate, is called remainder or remaining method.

2.3 Important Tools

Development grew to be a very tool heavy process. The Refactoring Bot thoroughly integrates into existing tools. SonarQube and GitHub are both tools often used in open source and are therefore integrated into the Refactoring Bot.

[2] We use refactoring suggestions or refactoring recommendations substitutional.

2.3.1 SonarQube

SonarQube[3] is a static analysis tool [CP13]. It works on the compiled bytecode to identify many kinds of flaws in the code and displays the results on a website.

Projects can be analyzed using predefined sets of rules, but developers can also create their own rules or adjust parameters of the predefined rule sets. In addition to custom rules and rule sets, SonarQube is build modular, is integrated into many build tools and allows the development of plug-ins which are directly integrated into the system. The three main components in SonarQube are responsible for analyzing the source code, saving the results to a database, and providing management and evaluation of the results on a website. Through its detailed RESTful API SonarQube can be easily integrated into continuous delivery systems or other tools.

2.3.2 GitHub

GitHub[4] is a platform to share code, collaborate in open-source projects and manage source code development. GitHub is based on the Git version control system and is mainly a code-hosting repository. Free hosting of public projects and advanced social and project management features made it a commonly used platform for open source projects. Currently there are over 24 million open-source projects hosted on GitHub[5] where more than 2.5 million are Java projects.

Similar to other distributed version control systems, own contributions to open source code can be made through pull requests. Pull requests are requests to make specific changes to the code which have to be reviewed by approved collaborators before they are added to the project. Everybody can create a GitHub account and create pull requests for any open-source project. With the GitHub API even software can automatically create pull request to any public repository or private repositories where the tool has correct access rights.

2.4 Refactoring Bot

The Refactoring Bot is an open source project developed in the university of Stuttgart led by M. Wyrich [WB19]. The goal is to develop software (in the form of a bot), which integrates seamlessly into a development teams work flow and improves the code based on findings regarding code quality from independent static analysis tools. Since tools for static code analysis are often integrated into continuous integration lifecycles, refactoring becomes an even more recurring task than it was before. Although multiple tools for automatic refactoring exist [KZN12; VCN+12] most developers tend to prform refactorings manually [STV16]. The Refactoring Bot improves code quality without requiring input or human interaction and still gives developers the option to review changes made. The most used tool for professional and open-source software development besides an IDE is a version control system like Git [Spi05]. The Refactoring Bot integrates into the version control system a project is using and uses pull requests so make changes to the code. Every finding that

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

[4] https://github.com/

[5] number of public repositories according to https://github.com/search?l=&q=size%3A%3E0&type=Repositories

has a corresponding refactoring method implemented can be fixed. Pull requests give developers the opportunity to review the changes and have the benefit that they are asynchronous and can be reviewed at the time the developer chooses [WB19]. Another benefit of pull request is the ability for anyone in the developer team to approve or reject them and the review is not bound to a single person. The bot also integrates into existing static code analysis tools and uses their reports to get defects regarding code quality in the source code. Most static analysis tools can be configured enabling the bot to make changes only for defects which the team actually sees as quality issues. The bot should appear as human as possible and can also respond to comments in pull requests. He acts as an additional team member, who works independently and consistently. Little to no configuration is needed for the bot to run, meaning that it can even automatically work on open source projects which haven't integrated the bot themselves.

2.4.1 Development State

The Refactoring Bot provides a REST API which can be accessed for testing purposes via a Swagger UI. The currently supported version control system is GitHub and the supported static analysis tool is SonarQube. Different tools for version control and static code analysis can be added with minor code changes but are not integrated at the time of writing. To refactor a project, the bot forks the project into his own GitHub account and makes all changes in this repository. A bot account has to be configured before using the Refactoring Bot for the first time. Pull requests with the applied changes are automatically created in the original repository. To use the bot, a configuration with the corresponding project on GitHub and SonarCloud[6] has to be created manually. At the current state the refactoring process must also be triggered manually through the REST API. Currently the Refactoring Bot supports following refactorings: add override annotation, remove commented-out code, remove unused method parameters and reorder language modifiers. In addition to this work to implement an extract method refactoring for too long methods, remove duplicate code is also in development.

[6] SonarCloud is a publicly hosted version of SonarQube.

3 Related Work

Common IDEs can assist developers in extract method refactorings or suggest candidates for method extraction and there are different approaches to identify these refactoring opportunities. This chapter describes different approaches and compares the work which was done around this research topic.

3.1 Refactoring Using Scoring Algorithms

R. Haas discusses an automatic refactoring based on scoring algorithms in [HH16] which is implemented as a plugin for an IDE. The approach uses quality analysis findings to select methods for refactorings based on their length. The maximum length of a method is assumed to be around 40 lines of code, as suggested in [SE14].

Selected methods are then represented with control and data flow graphs (CDFGs). The CDFGs are used to find compound statements in the method and deriving appropriate and behaviour preserving extraction candidates for extraction. This usually results in a long list of possible candidates which is sorted using a scoring algorithm. The scoring algorithm evaluates each candidate using individual points. In the end, the top three candidates are suggested to the user. The user should then choose one of the suggestions based on their judgment and perform the refactoring.

The scoring algorithms considers the length of the method, the nesting depth as well as the nesting area, the number of input and output parameters and the number of comments and/or blank lines around the candidate. Using this algorithm to rank candidates results in the highest ranked candidate to reduce the length of the original method the most without introducing new issues regarding quality to the code, for example an extracted candidate which is too long or has to many parameters.

Haas' work also includes an evaluation where he primarily analyses the accuracy of the developed scoring algorithm. Using open source projects he questioned independent developers about suggested refactorings generated with the implemented prototype. He states, that for 86% of the refactoring suggestions, the experts also selected the highest ranked recommendation made by the refactoring tool. The evaluation doesn't compare refactoring suggestions to independent selected refactoring candidates, meaning that no values for recall and precision are calculated and therefore cannot be easily compared to other approaches.

Similar to Haas' work, D. Silva et al. propose a technique to automate extract method refactoring using a model representing the code as a block structure containing statements that follow a linear control flow [STV14]. Every combination of blocks in this structure represents a refactoring opportunity if three conditions are met, including syntactical preconditions, behaviour preservation preconditions and quality preconditions which are also used in Haas' work. A big difference is the scoring algorithm. Since in control and data-flow-based approaches, every possible valid and extractable candidate is generated, the ranking system is the core to find fitting refactoring

opportunities. Although scoring functions are also used in other approaches ([TC11], [CAC+17]) and could be used regardless of the initial candidate generation algorithm every time to rank candidates, they are more important if generated candidates are less specific. Unlike R. Haas, D. Silva considers dependency sets to rank candidates. The rank of an opportunity is depended on variable, type and package dependencies between the subset of statements in the refactoring opportunity and the remaining original method.

In an exploratory study D. Silva analyzes the precision of the proposed scoring function and compares the generated refactoring recommendations to an other refactoring tool called JDeodorant [TC11]. When considering only the top ranked refactoring recommendations, the proposed approach achieves a precision of around 50% while the recall is as high as 85%. Considering only methods which also had refactoring suggestions made by experts, these values improved to 85% for precision and recall. Compared to JDeodorant this approach has a superior precision and recall but the studies were conducted in a completely different context and cannot be compared per se. In a second evaluation, the refactoring algorithm is applied to two open source software systems with reverted extract method refactorings. Reverted extract method refactorings can be achieved by applying inline method refactorings. These methods are later used to calculate precision and recall. In comparison to the smaller exploratory study, the resulting precision and recall are much lower with 38% and 38% respectively.

Another scored based approach is proposed by L. Yang [YLN09]. The author implemented his solution as an Eclipse plugin and evaluated the tool. In order to generate candidates, a method is divided into statements representing the structure of the code similar to work discussed previously. Related statements build refactoring candidates in any possible order-preserving combination.

Because this method leads to multiple candidates, they have to be further sorted. In order to rank the generated candidates, L. Yang calculates the ratio between in/output variables of the candidate and the reduction of length/complexity achieved with the candidate.

To evaluate this approach refactoring suggestions for an open source software system were assessed by two experts. The study showed that over 90% of refactoring recommendations made were accepted by the experts. The author states that this automation would result in an reduction of refactoring time needed by up to 40%.

3.2 Refactoring Using Graph Based Approaches

Graph based approaches use different graphs to represent statements from a selected method to find extract method refactorings. Although the previously-presented work also uses graphs to generate candidates, the generation of candidates is very generic and usually finds every possible candidate. In fact all work presented in this chapter uses graphs to represent code in some way in their approach. The focus of following approaches is on the actual generation of the candidates which usually results in a very small group with candidates based on their algorithm. In most cases other approaches result in a list of candidates without a rank (although some approaches have some specific ranking rules) and therefore require human interaction. Since our goals is a fully automated system it is required that some reliable sort of scoring algorithm finds the best possible candidate.

The following approach uses a combination of a data flow graph and a structure dependency graph to generate a list of candidates. T. Sharma [Sha12] created the data and structure dependency graph (DSD) from the data flow graph and the structure dependency graph to show structural and data dependencies between statements in on graph. He defines the DSD as follows:

Definition 3.2.1 (DSD [Sha12])
A DSD graph G can be defined as a pair (V, E), where V is (V_d, V_s) and E is (E_d, E_s), where V_d is a set of data-vertices (vertices representing source code statements), and V_s is a set of structure-vertices (vertices representing a block of statements), where E_d is a set of data-dependency edges (edges between the vertices V_d; $E_d \in \{(u, v) | u, v \in V_d\}$), and where E_s is a set of structure-dependency edges (edges between a data vertex and a structure vertex; $E_s \in \{(u,v) | u \in V_d, v \in V_s$ or $u \in V_s, v \in V_d\}$).An edge e_d (where $e_d \in E_d$) exists between two vertices u, v when there exists a data dependency between vertex u and v. An edge e_s (where $e_s \in E_s$) exists between two vertices u, v when there exists a structural dependency (one statement is dependent on the structure defined by the other statement) between vertex u and v

T. Sharma uses the generated DSD to find subgraphs using the longest edge removal algorithm. He states that long edges in the DSD indicate low coupling between these nodes and removing these edges results in subgraphs with high connectivity and therefore most likely a logical context. The generated subgraphs represent a refactoring opportunity if they fulfill certain behaviour preservation conditions. The author states that he want to conduct case-studies for his approach in the future but no work on this topic has been published so far.

Compared to previously discussed scoring-based approaches which also use graphs to find refactoring candidates this approach only leads to a very limited and manageable amount of refactoring candidates. That's why the author does not introduce a scoring algorithm to rank found candidates.

3.3 Refactoring Using Method Slicing

Method slicing is a technique to find a minimal subset of statements based on a selected statement which preserves the behaviour even when the slice would stand alone [Wei81]. This technique is mainly used for debugging a program [Tip94] but can also be used to find candidates for an extract method refactoring as the following literature shows. Program slicing is one of the most used techniques to find refactoring opportunities according to many different authors [GL91; LV97; TC11].

N. Tsantalis et al. use program slicing to identify extract method refactoring opportunities [TC11] He uses two different slices, the complete computation slice and the object state slice, to compute refactoring candidates. A complete computation slice is a subset of statements where a specific variable is used [MG01] whereas a object state slice is subset of statements affecting the state of an object [LH96]. The program dependence graph (PDG) [FOW87] is used to create static slices of the original method. In a first step, all complete computation slices are calculated for all local variables and parameters in the method. The slices are further split by block based regions which can be found by analyzing the method using a control flow graph. The second step calculates every object state slice for every object reference in the original method and the resulting list of slices are further distinguished by block based regions. After finding candidates using program slicing, the

list is filtered to remove candidates which would not preserve the program behaviour or could not be removed from the original method for different reasons including return statements in a slice. The list of refactoring opportunities is filtered regarding the usefulness of the extracted code to reduce the amount of proposed candidates. Because refactoring candidates in block based slicing might not be consecutive statements, refactoring opportunities might require the duplication of some statements in the original method and the extracted counterpart.

The proposed algorithm has been implemented in a plugin for the Eclipse IDE called JDeodorant[1]. Refactoring suggestions will be sorted by their effectiveness according to the duplication ratio.

The work includes two studies to evaluate the approach. In the first evaluation an open-source projects is chosen and refactoring suggestions using the implemented tool are independently assessed by an experienced software designer. According to this assessment 90% of suggested refactoring opportunities were approved and 40% of refactorings actively solved design flaws in the code according to the expert. In a second study refactoring opportunities found by experts were compared to results of the algorithm to calculate precision and recall. Results showed a precision of 51% and a recall of 69% which lead to the conclusion that the proposed approach can successfully substitute an human developer.

Compared to other implementations, sliced based extract method refactoring suggestions can contain non consecutive statements but also have the risk to introduce duplicate code to the project. In contrast to previously discussed refactorings using control flow graphs, sliced based implementations cannot find all possible candidates due to the nature of the finding process where only slices of local variables and object references are considered.

An other approach based on block-based slicing to automatically apply the extract method refactoring is presented by K. Maruyama [Mar01]. In contrast to N. Tsantalis approach, he doesn't create a list with possible candidates for each local variable in the method but lets the developer select a variable whose slice should be extracted. The appropriate block-based slice is calculated using control flow graphs and program dependence graphs. The new method build from the program slice will always have the type of the selected variable as return type. Ultimately this means that only a very limited number of refactoring opportunities are found by this approach, namely as many as the number of local variables existing in the selected method.

The author has not evaluated his approach and concludes that the tool is helpful for developers when extracting methods to be able to extract behaviour-preserving substatements from the original method easily.

3.4 Refactoring Using the Single Responsibility Principle

The single responsibility principle (SRP) [Mar02] is a pattern in object-oriented programming which states that every class should only have a single responsibility. S. Charalampidou et al. [CAC+17] apply this to methods and state that long methods usually don't statisfy the SRP. The paper proposes a completely different approach than the two previously-discussed, more common solutions. The SRP-based extract method identification (SEMI) finds code fragments with a strong cohesion and

[1] https://github.com/tsantalis/JDeodorant

uses them to suggest refactorings to improve the SRP rating for the original method. Similar to approaches discussed previously this implementation also uses a two step process; first it finds appropriate candidates and than evaluates and ranks them in a second process.

To find a candidate the work suggests to evaluate the coherence between statements based on three criteria. Statements are considered more coherent if they access same variables, call methods of the same object or call same methods on different objects (which have the same type). The list of extract method candidates is further processed to remove duplicates and invalid candidates which would lead to compilation errors or alter the behaviour after the method extraction. To rank the resulting opportunities the lack of cohesion in methods (LCOM) measurement and the size of the candidate is used. LCOM measures the cohesion of methods; or how good a method implements the SRP [Sha]. Since this refactoring approach focuses only on the SRP of a method, LCOM is a good indication whether the refactored method and candidate pair is better than the original method.

To evaluate this implementation two case studies were conducted. In the first study they examined two different lengthy methods in an industrial project and compared developer opinions to the results provided by their algorithm in terms of a list of consecutive statements which are considered to be coherent and the ranking of refactoring candidates. The results show that the algorithm has a higher recall in identifying coherent code blocks than other studies but a lower precision compared to top results in different approaches. The ranking created by their approach however only matches the suggestions of the developers by roughly 50%. In a second study the SEMI approach is compared to other solutions like JDeodorant [TC11] or JExtract [STV14]. Five different open-source projects were analyzed using SEMI, JDeodorant and JExtract. The top 5 suggestions of each tool were compared to developer opinions. Overall, their approach is not significantly better than compared implementations in terms of recall and precision. They still state that the evaluation proves that automatically extracting methods from too long methods based on SRP is a good alternative to more common approaches and is better at handling the trade off between recall and precision.

Compared to program slicing based approaches or solutions using the abstract syntax tree, SEMI identifies coherent functionality in functions and therefore identifies a better selection of candidates even before rating them compared to other techniques which either try to find all possible candidates or only a very abstract list of candidates.

As already stated earlier, our solution needs to meet one important condition. The extract method refactoring should run without any human interaction. This implies that a developer cannot select a part or variable of the long method before the extraction and cannot choose the best refactoring opportunity after the solution recommends some candidates. Of the previously discussed approaches, only scoring based solutions can fully comply with these requirements. Our approach combines different scoring algorithms discussed in this chapter to improve the candidate ranking, as we must always apply a refactoring without review.

4 Implementation

This chapter gives detailed insight into our approach to automatically refactor too long methods using an extract method refactoring. Our solution builds onto previously discussed score based implementations. Improvements and changes made to existing solutions are discussed thoroughly in this section. In the second part of this chapter specifics to our implementation are revealed.

4.1 Approach

The approach presented in this work extracts methods based on quality analysis findings. We do not need to specify a maximum method length because in the approach we assume that every method we get as input is too long. When implementing the extract method refactoring in the Refactoring Bot, the input would be issues from an independent quality analysis with specific information about which method is too long.

Although we do not need to know the ideal length of a method before doing an extraction it might still be important later on, because the extracted method and the remaining method should also not exceed this limit. It could also be important for future work to better prioritize methods from quality analysis findings before refactoring them. Unlike other quality issues, there is no commonly accepted opinion on how long a method should or can be. M. Lippert proves that methods with more than 30 lines of code (LoC) are more error prone than others and suggests that methods should not be longer than 30 lines [LR06]. An often used rule for the length of method is that a method should be completely visible on a small monitor [Mar09]. Furthermore a method should always only cover one functionality for better readability and so it does not get too complex. This principle is used in the previously discussed work by S. Charalampidou [CAC+17]. As this cannot be recognized easily by a static analysis tool, they fall back to a simple LoC measurement. SonarQube has a very conservative maximum length of 70 LoC set as default for the "too long method code" smell. A healthy value is proposed by D. Steidel with 40 LoC [SE14]. We will use this as reference value when we talk about methods with a acceptable length.

We generate an exhaustive list of refactoring candidates first. To get this list, the source code is transformed into a block structure, similar to [STV14]. We call this block structure *statement graph* because it contains more than just structural information. The statement graph allows us to get a complete list of all valid (contiguous) refactoring candidates in a method. In a second step, the generated list is sorted using a scoring function. The scoring function is an improved and adapted version of the scoring function presented in D. Haas' work [HH16]. It considers method length, cognitive complexity, the number of input and output parameters and code formatting. The highest ranked candidate will be automatically refactored into a new method.

We will use the example code shown in Listing 4.1 throughout this chapter. To generate a list

Listing 4.1 Example method for applying an extract method refactoring.

```
1  public void runProgram() {
2          println("The calculator is ready to calculate!");
3
4          // get numbers
5          Scanner inp = new Scanner(System.in);
6          int num1;
7          int num2;
8          println("Enter first number:");
9          num1 = inp.nextInt();
10         println("Enter second number:");
11         num2 = inp.nextInt();
12
13         // get operation
14         println("Enter your selection: 1 for addition, 2 for subtraction, 3 for multiplication and 4
               for division:");
15         int choose;
16         choose = inp.nextInt();
17
18         // calculate and print result
19         if (choose == 1) {
20                 println("The result of the addition is:");
21                 println(num1 + num2);
22         } else {
23                 println("Illegal Operation");
24         }
25
26         inp.close();
27         println("Thank you for using this awesome calculator.");
28 }
```

of candidate we need to build the statement graph first. The statement graph is a block structure representation of the source code first introduced by D. Silva [STV14] and further elaborated by F. Streitel [Str14]. F. Streitel uses the CFG to generate a CDFG which is later used to easily build a statement graph. The statement graph is a simplified version of the AST but also stores information from the CFG. Each node or block represents one line (statement) in the method and the nesting depth of the code is the same in the statement graph. Figure 4.1 shows the statement graph for the example method. Additionally the statement graph stores data edges between statements containing same variables. Special type information is also stored in each node regarding special cases like `if`/`else` statements or `try`/`catch`/`finally` statements. The root node/block is always the method definition and contains all following statements from the method.

The candidate generation algorithm uses the statement graph to extract a complete list of possible refactoring candidates. For that, every block/node from the statement graph is combined, so that continuous candidates are built. Not all of these substatements are valid refactoring candidates, for a candidate to be valid D. Silva et al. point out three preconditions that must be met [STV14]. These three preconditions are syntactical preconditions, behaviour-preservation preconditions and quality preconditions. The candidate generation algorithm in Algorithm 4.1 makes sure that every candidate meets these conditions.

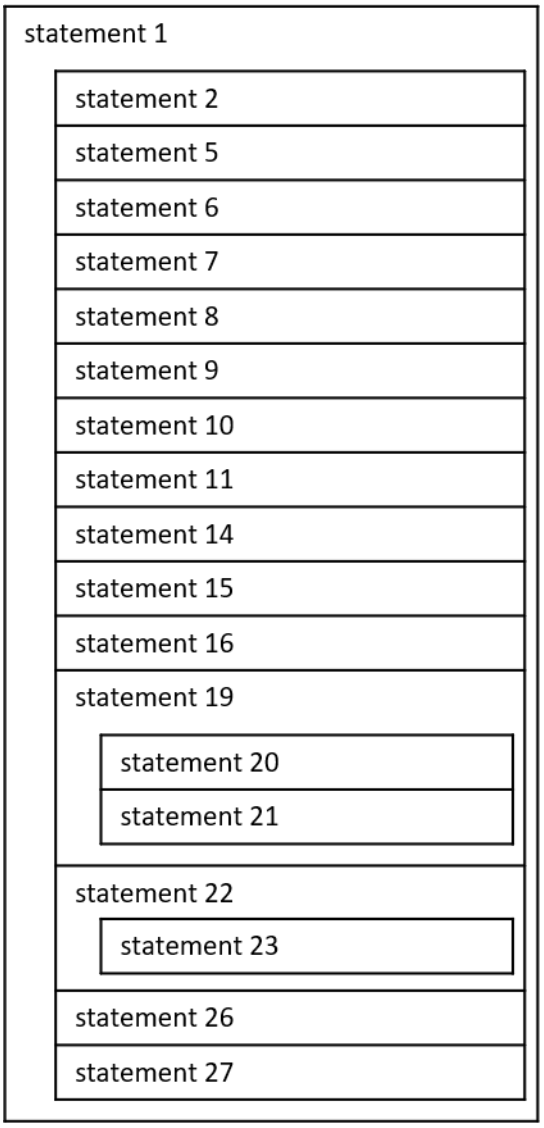

Figure 4.1: Statement graph generated from the example method

Syntactical Preconditions Syntactical Preconditions define how a group of substatements in the statement graph can be built to resemble a refactoring candidate. The syntactical preconditions are important to produce code which can still be compiled. All the syntactical preconditions are already satisfied by the structure of the statement graph and the nature of the generation algorithm. This means that only candidates which satisfy the syntactical preconditions are generated whereas for behaviour-preservation preconditions and quality preconditions candidates might be generated which do not satisfy them. These candidates will then be rejected later on in the generation algorithm. Following conditions have to be met:

- Extracted substatements must always have the same parent statement. For example you can extract statement 10 and 11 but not statement 23 and 26 because they have different parent statements whereas statement 10 and 11 both have statement 1 as their parent. This rule only

Algorithm 4.1 Pseudo code of the candidate generation algorithm.

```
procedure generateCandidates(rootNode)
    candidates ← ∅
    for all i ∈ 0...countChildren(rootNode) − 1 do
        for all j ∈ countChildren(rootNode) − 1...i do
            candidate ← getChildrenBetween(i, j)
            if isLongEnough(candidate) then
                if isValid(candidate) then
                    if isExtractable(candidate) then
                        candidates ← candidates + candidate
                    end if
                end if
            end if
        end for
        child ← getChildAtIndex(rootNode)
        candidates ← candidates + generateCandidates(child, i)
    end for
    return candidates
end procedure
```

applies to the root node of a selected subgraph. For example if statement 19 to 23 are selected only statement 19 and 22 must have the same parent statement because statement 20, 21 and 23 are child statements of 19 and 22 respectively.

- Selected substatements must be continuous. Each selected statement must be connected directly in the statement graph node. For example statements 8 and 9 can be selected but not statements 8 and 10. Statements 8 and 10 do not have a direct connection in the statement graph.
- If a statement is part of a refactoring candidate all its child statements are also included in the candidate. For example if the algorithm selects statement 19, statement 20 and 21 are also automatically included in the candidate.

These preconditions are automatically satisfied for every generated candidate with the two `for`-loops at the beginning of the algorithm, the recursive calling at the end and the `getChildrenBetween`-method which returns all concurrent statements with their children.

Special cases are `if`/`else` blocks and `try`/`catch`/`finally` blocks. These blocks have to be extracted as a whole but are not covered by the three syntactical preconditions. That is because in the statement graph `if`, `else`, `try`, `catch` and `finally` statements each have one corresponding block. This can be seen in Figure 4.1 where statement 19 is an `if`-statement and statement 22 is an `else`-statement. The method `isValid` in the candidate generation algorithm checks this and only accepts candidates where only complete `if`- and `try` statements are included.

Behavior-Preservation Preconditions As a second precondition, the algorithm ensures that the behavior of the original method is preserved. Following four aspects are important to preserve the behavior when extracting methods in Java programs:

- An extracted method can only have one return value. In the Java language only zero or one values can be returned from methods. This means that only one variable or parameter that is assigned in the selected statements can be used after the selected statements in the original method. If more than one variable or parameter would be reused after the selection, the extracted method would need more than one return value.
- `return` statements can only be extracted if all code paths lead to a return statement. R. Haas states that `return`, `continue` and `break` statements cannot be extracted, because they change the control flow of a method in such a way that a behavior-preservation could not be guaranteed [HH16]. This is only partially true; these statements alter the the control flow but if the change of control flow only occurs within the selected statements, they still form a valid candidate. A `return` statement can be extracted if all control flow paths in the candidate lead to that or another `return` statement. We can extract this information from the CFG.
- `continue` and `break` statements can only be extracted if their corresponding loop statement is extracted too. The loop statement is not the immediate parent statement in every case. As already stated above, `continue` and `break` statements can be extracted in some cases. They affect the nearest loop or can be labeled to directly jump to the specified label. They need to be extracted with the corresponding loop or label together. A candidate with a `continue` or `break` statement is valid, if the nearest loop or corresponding label is also part of the candidate.
- A `switch-case` expression is represented as nested `if-else` statements in the statement graph but can only be extracted as a whole. Due to the fact that the statement graph is mostly generated from the CFG, `switch-case` statements are not correctly represented. We need to check for `switch-case` statements and reject all possible candidates where they are only partly included.

The behaviour-preservation preconditions are checked by the `isExtractable` method in our candidate generation algorithm.

Quality Preconditions The last precondition ensures a minimum quality for each candidate. An extractable candidate could also be found without any quality requirements but this works as a prefilter to filter out unwanted candidates. D. Silva proposes to check for a minimal length of the candidate and remainder. This ensures that the candidate has more lines than a set minimum, because a candidate with only one statement is rather useless. The same holds for the remainder, a candidate that covers the whole body does not change anything at all. The minimum applies to the candidate and remainder which means that a method with less than twice the length of the set minimum would never result in any candidates. He states that a minimum of 3 lines leads to the best results and we will use the same value in our approach.

4.1.1 Scoring Algorithm

After a list of valid refactoring candidates is generated, they are given an individual score in a second step. The scoring function we use in our approach is heavily inspired by the scoring function which R. Haas used in his work [HH16]. We made minor improvements based on our opinion, our experience and independent research.

The scoring function evaluates four different categories in a candidate, namely length, cognitive complexity, parameters and semantics. The final score is calculated by summing up each category score multiplied with the category weight. The candidates are ranked according to the highest score. Each category and score calculation for the category is described in detail in following paragraphs.

Length One of the most important scoring categories is the length. We rely on the method length to find defects in the static code analysis and the method length should therefore be the value which is primarily improved with the refactoring. We calculate the length score using the minimum of the length of the candidate L_C and the length of the remainder L_R. The highest length score S_{length} for a candidate can be achieved when the candidate splits the original method perfectly in half. The best score in a method with 100 LoC would be achieved by a candidate who extracts exactly 50 LoC leading to two new methods where both have 50 lines (plus one line in the remainder method for the call to the extracted method). Because these methods would violate the too long method rule again, we set an upper bound of MAXLENGTH. A candidate where the minimum length of the remainder or candidate is longer than this value, no extra points are rewarded. The default value for MAXLENGTH is set to 40 in our approach. The complete length score S_{length} is calculated with:

$$S_{\text{length}} = W_{\text{length}} * 0.1 * (\min(\min(L_C, L_R), \text{MAXLENGTH})) \tag{4.1}$$

where W_{length} is the weight for the length score. All weights are 1 per default and a scoring function might have a normalizing constant (in this case 0.1) to bring the score to the desired range. When the weight and MAXLENGTH are set to the default values, the length score rewards points in a range of 0 to 4. A candidate where the remainder and the candidate are at least 40 lines long would achieve the maximum score of 4.

Cognitive Complexity There are a lot of different approaches to measure the methods complexity besides its length. Even though the complexity for method is generally higher the longer a method is, there are a lot of different approaches to measure method complexity based on nesting [HWY09]. We use a complexity rating as a secondary score for the effectiveness of a refactoring candidate. Contrary to R. Haas who calculates a complexity score based on the nesting depth and nesting area, we calculate a simplified cognitive complexity measurement to determine the complexity score. Cognitive complexity measurements aim to better model the comprehensibility of methods [SW03]. The cognitive complexity also replaced the previously used cyclomatic complexity as complexity measurement in SonarQube in the beginning of 2017 [Cam16]. The cognitive complexity measurement used in SonarQube considers that methods get significantly more complex the deeper they are nested but recognizes that structures with multiple statements (for example `switch - case` statements) do not increase the complexity significantly [Cam17].

We use a simplified version of the cognitive complexity measurement in our approach. The simplified cognitive complexity gives each nesting statement points according to their nesting depth. The Complexity C of a method is calculated like this:

$$C = \sum_{s=1}^{\text{length}_{NS}} d(s) + 1 \tag{4.2}$$

where length_{NS} is the number of nesting statements in the method and $d(s)$ is the depth of the nesting statements at index s.

The complexity score for the refactoring candidate is the reduction of complexity by the refactoring considering both, the remainder and the candidate. The complexity score is 0 when either the remainder or the candidate have the same complexity as the original method. Otherwise the score is the difference of complexity between the original method and the maximum complexity between candidate and remainder. It is determined as follows:

$$S_{\text{complexity}} = W_{\text{complexity}} * (C_{\text{method}} - \max(C_C, C_R)) \tag{4.3}$$

where $W_{\text{complexity}}$ is the weight of the complexity score, C_{method} the calculated complexity of the original method, C_C the calculated complexity of the candidate and C_R the calculated complexity of the remainder.

Parameters The scoring function in D. Silvas work [STV14] uses dependency sets to calculate the score of a candidate. Dependency sets include dependencies between the remainder and the candidate regarding variables, types and packages. L. Yang also uses the number of parameters to rank refactoring candidates [YLN09]. Parameters from the refactoring candidate are zero or more input parameters and zero or one output value. R. Haas states that more parameters lower the quality of a candidate and therefor assigns a lower score to a candidate if it has more parameters [HH16].

The maximum score is determined by the maximum number of acceptable input and output parameters. We follow R. Haas' suggestions that a method can have at most 4 parameters without significantly reducing the quality of the method in this approach. The parameter score is lowered by one for every input or output parameter. So a candidate with 3 input and 1 output parameter or 4 input and 0 output parameters gets a score of 0 and candidate with more parameters than that get a negative parameter score. R. Haas assigns every candidate which have more than the allowed MAXPARAMETERNUMBER a score of -10000, because methods with more than the set amount of parameters are not sufficient in his opinion, but later states that useful methods exists with more than 4 parameters [HH16]. Therefore we do not automatically give candidates with a lot of parameters a score of -10000. To calculate the parameter score this approaches uses following formula:

$$S_{\text{parameters}} = W_{\text{parameters}} * (\text{MAXPARAMETERNUMBER} - N_i - N_o) \tag{4.4}$$

where $W_{\text{parameters}}$ is the weight of the parameter score, N_i is the number of input parameters and N_o is the number of output values. Contrary to R. Haas parameter score, we do not make an exception for methods where the original method has more parameters than MAXPARAMETERNUMBER because the number of parameters of a refactoring candidate are not dependent on the number of parameters in the original method. We therefore do not see the need to make an exception for methods with a lot of parameters.

Semantics It is hard to interpret code and understand its semantics with a static code analysis tool. Nevertheless semantics of the code are a very important indicator for good extraction candidates. S. Charalampidou et al. use the SRP in their approach to isolate functionality in a method [CAC+17]. R. Haas suggests to count blank lines and comment lines at the beginning and end of a candidate because different or isolated functionality in source code is normally structured by empty lines or

comment lines [HH16]. He states that empty lines or comment lines at the beginning of a candidate are more important and the number of empty lines and comment lines should determine the score additionally to a value indicating if there are any lines at the beginning or end.

In our approach we count the number of empty lines and comment lines at the beginning and end of a candidate and add one more point to each value if the number of lines is greater than 0. Empty lines or comment lines are given more relevance by multiplying their value with the constant factor c_b. The semantics score is calculated as follows:

$$S_{\text{semantics}} = W_{\text{semantics}} * 0.25 * (c_b * (L_b + N_{lb}) + L_e + N_{le}) \tag{4.5}$$

where $W_{\text{semantics}}$ is the weight of the semantics score, c_b is the factor to make comments and blank lines at the beginning more important for the score, $L_b \in \{0, 1\}$ is a value indicating if there are blank lines or comment lines at the beginning, N_{lb} is the number of blank lines or comment lines at the beginning, $L_e \in \{0, 1\}$ is a value indicating if there are blank lines or comment lines at the end and N_{le} is the number of blank lines or comment lines at the end.

R. Haas argues that an excessive amount of empty lines or comment lines should not reward an excessive amount of points and sets an upper bound of 3 to N_{lb} and N_{le}. We do not agree with this, an excessive amount of empty lines or comment lines in methods is rare, but when they occur the probability that they indicate isolated functionality is very high so our scoring function does not set an upper bound for the semantics score.

The lowest score for semantics is 0 but it has no upper bound.

Total Score The total score is calculated by adding all four distinct scores:

$$S_{\text{total}} = S_{\text{length}} + S_{\text{complexity}} + S_{\text{parameters}} + S_{\text{semantics}} \tag{4.6}$$

The total score does not have a lower or upper bound. All generated candidates will be scored using this function. At the end the candidate with the highest score will be selected for extraction.

4.2 Implementation Details

We implemented the presented approach using the existing Refactoring Bot framework. The framework provides an API for an extract method refactoring with information about the location of the file containing the method which was too long and information about the starting line number of the affected method in the file. The refactoring should then be applied to the file at the provided location.

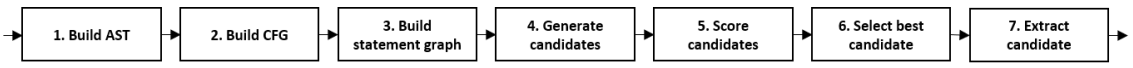

Figure 4.2: Method extraction process in the implementation.

The whole process can be divided into seven steps as shown in Figure 4.2. We used the external framework "checkerframework"[1] to build the CFG. We use the generated CFG to build a simple statement graph that only reflects the code structure without data edges yet. Because the CFG does not reflect `try-catch-finally` structures as we need them, their structure is added to the statement graph in a second step. After that, the data edges are added to the statement graph using control flow information from the CFG.

The candidate generation implementation strictly follows the algorithm presented earlier in this chapter (Algorithm 4.1). The implementation always generates a complete list of candidates no matter how many there are. There can be methods where no valid candidate can be found. In this case, the refactoring will cancel the refactoring process and return an error.

All generated candidates are scored individually. The weight of each score category can be configured manually when the Refactoring Bot is added to the project. Because the score does not have a lower or upper bound, all candidates potentially could have a negative or low score. The implementation does not require a minimum score, always the candidate with the highest score will be extracted.

Our implementation then removes the selected statements from the original method, copies them into a new method and calls the new method from the original location. While extracting the method, we have to differentiate between some cases, to produce code that still compiles and preserves the behaviours of the original method:

- if the extracted method has a return value, the call to the extracted method has to be assigned to the corresponding local variable
- if the extracted method has a return value but no return statement, a return statement with the corresponding local variable has to be added at the end of the extracted method
- if the extracted method has a return statement, the call to the extracted method has to be behind a return statement in the remainder instead of assigning it to a local variable
- if the original method can throw, the extracted method needs the same throw modifiers in its definition
- if the extracted statements are within a `try` block, the extracted method needs throw modifiers for all exceptions caught in the corresponding `catch` blocks

4.2.1 Challenges and Restrictions

During the implementation we faced some challenges, two of them could only be solved partly:

[1] The checkerframework (https://checkerframework.org/) is an open-source library which is used to strengthen the Java type system. It includes a complete control flow analysis which also generates a CFG.

Method Naming The extracted method needs a descriptive name. Generating a suitable name for an extracted method was not scope of this work and we didn't explore possibilities to find a good name further. Some possibilities are described in Chapter 5 because we needed descriptive names in the evaluation. In this implementation, an extracted method gets a generic name combined of "extractedMethod" plus the name of the original method. This allows developers to quickly find automatically named methods and also find the original method based on the method name. The name of the extracted method should be changed before merging the pull request, future versions of the Refactoring Bot could allow developers to change the method name using comments in the pull request.

Local Control Flow Analysis The usage of the checkerframework revealed problems at the end of the implementation. To conduct a complete control flow analysis, the checkerframework needs information about all types used and the control flow of called procedures. Some of these information are only accessible when the whole project is compiled, which can not be done with the Refactoring Bot and is not its intention. This means that our implementation does not work in some cases where a method calls external procedures. We tried several things to solve this problem. A mostly working solution was to change the code of the data flow analysis to handle unknown datatypes and procedures correctly. This worked in most cases but without completely rewriting the code, we could not build a solution which always produced a correct CFG. During this work, we did not have the time to reimplement a custom lightweight control flow analysis, but recommend to implement a custom lightweight solution in future work.

5 Evaluation

Evaluation is an important part of this work and helps us to improve the implementation and find potential topics for future work. This chapter describes the evaluation we performed and describes research questions, study design and study results in details.

5.1 Research Questions

Since the candidate scoring in our implementation is based on the previous discussed work from R. Haas [HH16] with only minor improvements, it is not necessary to reevaluate the algorithm. His evaluation showed that well over 85% of the selected candidates by the scoring function were accepted by developers. We assume our improved algorithm would have similar or better results and try to tackle different topics in this evaluation. Following two research questions should be answered in this evaluation:

RQ1: Is there a strong correlation between the length of a method and the acceptance of an extract method refactoring applied to this method? We aim to improve open-source projects automatically and want to create the best user experience possible for the maintainer of these projects. This means that refactoring suggestions made my the bot should be of the highest possible quality. If there is a pattern for unaccepted refactorings made by the bot, these suggestions should not be considered anymore. As already discussed, there is no agreement between researches on how long a method can be, before it is considered to be too long. The length of the original method is the measurement we use in this work to determine where we can apply an extract method refactoring. Since the process after a method is chosen is already evaluated we want to focus on the first part where methods which are too long are chosen. We evaluate if there is any correlation between the length of a method and the acceptance of an extract method refactoring. If we find a correlation between the length of a method and the acceptance of an extract method refactoring we further evaluate how length of the method affects the acceptance.

RQ2: What is the minimum length of a method for an extract method refactoring to be accepted with high probability? Assuming we find a correlation between the length of a method and the acceptance of an extract method refactoring we want to know how long a method has to be so that we can decide whether or not the refactoring would be accepted with a high probability. Knowing this threshold would enable us to further improve the automatic extract method refactoring.

Name	Repository Link	Size (Java LoC)
Apache Kafka	kafka	400000
MyBatis	mybatis-3	90000
React Native Camera	react-native-camera	12000
Material Theme Jetbrains	material-theme-jetbrains	32000
CAS	cas	264000
Swagger-Core	swagger-core	65000
Graylog	graylog2-server	220000
GoCD	gocd	468000
Alluxio	alluxio	460000
Flyway	flyway	36000

Table 5.1: Study Objects

5.2 Study Design

The goal when designing our study was to conduct the study in an environment similar to the environment the finished Refactoring Bot would work in. Since the Refactoring Bot works with automatic pull requests we decided to use pull requests in the study and to not interact with developers directly.

5.2.1 Study Objects

The study objects are shown in Table 5.1. The projects were chosen at random from all projects in GitHub with Java code. To make sure that the chance for responses to pull requests was high, we only chose projects with at least three contributors and recent activity in the last two weeks. Projects were further filtered so that only projects with over 4000 stars were considered. Stars on GitHub can be compared to likes on other social media platforms and indicate that projects meet certain quality requirements and are more often used than projects with less stars. To have quality standards in a project is important, otherwise refactorings which improve the software quality would not be seen as important.

Bigger projects on GitHub tend to enforce restricting contribution rules but chosen projects in this evaluation allow direct contribution over pull requests and do not require to use pull request templates. All ten projects vary greatly in size in terms of LoC and relevance on GitHub. The context and owner for each project are also completely different.

Group	LoC of Method
1	20 - 29
2	30 - 39
3	40 - 49
4	50 - 59
5	60 - 69
6	70 - 79
7	80 - 89
8	90 - 99
9	100+

Table 5.2: Groups for Findings in Evaluation

5.2.2 Study Setup

Each selected project was locally compiled and analyzed using the Sonar Scanner[1]. The analysis was configured to only check for too long methods with a minimum LoC of 20. This value was intentionally lower than previously discussed thresholds for the maximum allowed length of a method enabling us to better answer RQ1. The findings were grouped by their length as seen in Table 5.2.

From every group, one method was randomly chosen and refactored using the Refactoring Bot and the implemented extract method refactoring. With nine groups and ten projects a total of 90 refactorings could be performed in theory. This number was lower in practice because not every project had methods filling all groups and a valid refactoring candidate could not be found every time.

Method naming is an important aspect for code quality [BWYS10]. This is problematic because our approach does not give a meaningful name to extracted methods. Naming extracted methods has do be done manually to this date. Naming a function often requires deep knowledge of the semantics of that method. Manually naming every extracted method in this evaluation could lead to drastically different quality for method names and would lower the validity of the evaluation. To tackle this issue we created a comprehensible naming scheme (Figure 5.1) to guarantee equal naming for every method. The scheme mainly uses input and output parameters to determine a fitting name.

A pull request was created for each extracted method from a completely new GitHub account only used for this evaluation. The name of the account made it clear that it was a bot account. For each project only three pull requests were open simultaneously. New pull requests were only opened once the old ones were processed and the method for the new pull request was chosen randomly.

Every pull request was titled "*Extract Method from [originalMethodName]()*". The pull request description was the same for every pull request and ensured that developers knew that the pull request was made automatically by a bot and was part of an evaluation (see Figure 5.2).

[1]. Sonar Scanner is a tool provided by SonarQube to analyze any project locally (https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner)

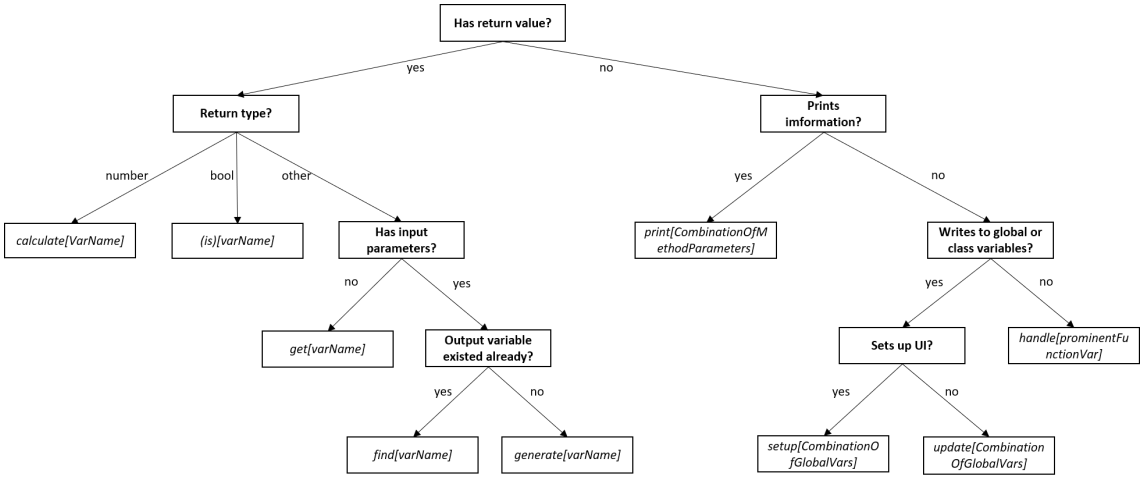

Figure 5.1: Naming scheme for extracted methods

This pull request was made automatically by me, a refactoring bot. Pull request I make are supposed to improve readability and maintainability of your source code. I am currently in development and this refactoring is part of an evaluation. Unfortunately, I cannot respond to comments yet so please consider this while reviewing this pull request. If you reject the pull request, I would appreciate any feedback so that my developers can further improve me.

Figure 5.2: Pull Request text for evaluation

5.3 Results

We collected the data over a period of four weeks. Pull requests were not created simultaneously, so pull requests might have been open for less than four weeks. Overall 23 pull requests were created. Complete results are displayed in Table 5.3.

23 pull requests overall is much lower than the previously mentioned 90. Due to the fact that 85% from our initial pull requests were either not reviewed by the end of the evaluation period or closed without review, no follow-up pull requests were posted for these projects. Furthermore methods often didn't exist for every group (see Table 5.2) in projects. Especially long methods with over 70 LoC were rare in the examined projects. Our quality analysis for *CAS* revealed that only 0,02% of methods had a length of over 70 LoC. An UI-heavy project like *Material Theme Jetbrains* which typically has longer methods, also had only 0,6% of methods with more than 70 LoC.

Unfortunately we cannot answer RQ1 and RQ2 with the limited amount of data of acquired data. The questions arises if our study design was sufficient to answer RQ1 and RQ2 confidently.

5.4 Discussion

In following section we discuss why the study design did not deliver enough data we could analyze. We compare results from different studies and make assumptions based on experiences from comparable experiments as to why our pull requests were ignored or rejected categorically.

Name	Pull Requests	Accepted	Rejected	Closed without Review	Open
Apache Kafka	2	0	0	0	2
MyBatis	3	0	0	0	3
React Native Camera	0	0	0	0	0
Material Theme Jetbrains	3	2	1	0	0
CAS	2	0	2	0	0
Swagger-Core	3	0	0	0	3
Graylog	3	0	0	3	0
GoCD	1	0	0	0	1
Alluxio	3	0	0	0	3
Flyway	3	0	0	3	0
total	23	2	3	6	12

Table 5.3: Data Acquired in the Evaluation

Assumption: Pull request generally have a very low acceptance rate. GitHub is a platform with a lot of users. Every user can make pull requests to any open source project and especially popular projects have to review a lot of pull requests every day. A study examined nearly 80000 pull request on GitHub where nearly 60% were not accepted [RR14]. They found out that projects using Java have an even higher rejection rate of around 80%. Furthermore the authors state that many more factors like developer experience or number of forks might influence the number of accepted pull requests.

Another study investigated the social aspects of GitHub and how developers interact with each other [DSTH12]. They found out that decisions made on pull requests were often influenced by surrounding factors like the thoroughness (usage of pull request templates, inclusiveness of tests corresponding to changes and more) or the quality of the code. Sometimes even the experience and past contributions of the developer making the pull requests were evaluated.

As we used a fresh account with no description and no prior contributions, and very generic pull requests descriptions, it made it harder for reviewers to make a decision, since they couldn't rely on their default processes. We assume that this lowered the already low percentage of accepted pull request in general for our evaluation even further. These studies show that it is not possible to reduce the acceptance rate of pull requests solely on the code changes and a lot of other factors influence the review process.

Our study design also limited the feedback developers could give. The pull requests in our study made it clear that the contributions came from a bot. So it was impossible to determine if a rejected pull requests was rejected because the changes were bad or not useful or because of other various

reasons which were not directly connected to the code changes. We assume, that due to non observed factors and the fact that pull requests are often rejected for other reasons than the committed change, it is hard to get relevant data in an evaluation of this scale.

Assumption: Good code quality of examined projects lowered the quality of our refactorings. We evaluated ten different popular projects with already more than 4000 closed pull requests on average. Having many contributions and being a popular project leads to the conclusion that these projects already have a higher software quality than usual. A study conducted by I. Stamelos can confirm this assumption [SAOB02]. The examined open-source projects showed that their code quality was not significantly lower than industry standards.

We based our refactorings on a static code analysis run locally and randomly selected findings. Especially for longer methods (more than 70 LoC) only few issues were detected. This indicates that the code overall has a very low number of too long methods and remaining methods might be harder to refactor. Looking into some findings from the quality analysis proved that many long methods could not be shortened manually due to their structure. An example from the project *Material Theme Jetbrains* is shown in Listing 5.1 (mutliple lines were removed from the example to make it shorter). The method has 72 LoC but there is no useful refactoring candidate because the method is really simple, does one thing, and is comprehensible even though it is very long. This can be observed for other long methods, especially in test cases and UI components. We found that some of the refactorings done by the bot and created as pull requests were not relevant because they didn't fix any quality issues in the code. For some methods, the method length indicator is not sufficient to measure the comprehensibility and readability of methods and other measurements have to be used.

Another factor for the quality of our refactorings was the scripted method naming. Good method names are very important to make an extract method refactoring useful and although our naming scheme gave every method a name of equal quality, manual naming would still greatly improve the refactoring in many cases.

A last reasons why the quality of some pull requests was exceptionally low was the intentional short maximum length of a method configured in the static code analysis. We configured the maximum allowed length to 20, which is way lower than the previously discussed optimal length. This was the nature of our evaluation but let to the effect that we made pull requests and already knew that the outcome could be a rejection. This is of course not how pull requests should be used and negatively affects reviewers when reviewing future pull requests from the same bot.

Assumption: Developers are more biased towards a bot and will reject more pull requests. More than 25% of projects on GitHub use bots in their build process [WDS+18], showing that developers are pretty familiar with bots working in their projects. Often these bots are only responsible for management and continuous integration task and don't make changes in the code. Is there bias towards bots making pull requests, even if the pull request itself isn't distinguishable from a pull request made by a real developer?

Listing 5.1 Too long method from *Material Theme Jetbrains* with 72 lines

```
1   public void processOptions(@NotNull final SearchableOptionProcessor processor) {
2     final Configurable configurable = new MTConfigurable();
3     final String displayName = configurable.getDisplayName();
4
5     final List<String> strings = Collections.unmodifiableList((
6         Lists.newArrayList(
7             //region Strings
8             MaterialThemeBundle.message("MTForm.accentScrollbarsCheckbox.text"),
9             MaterialThemeBundle.message("MTForm.activeTabHighlightCheckbox.text"),
10            MaterialThemeBundle.message("MTForm.arrowsStyleLabel.text"),
11            MaterialThemeBundle.message("MTForm.codeAdditionsCheckBox.text"),
12            MaterialThemeBundle.message("MTForm.compactDropdownsCheckbox.text"),
13            MaterialThemeBundle.message("MTForm.compactPanel.tab.title"),
14            MaterialThemeBundle.message("MTForm.componentDesc.textWithMnemonic"),
15            MaterialThemeBundle.message("MTForm.componentsPanel.tab.title"),
16            MaterialThemeBundle.message("MTForm.contrastCheckBox.text"),
17            MaterialThemeBundle.message("MTForm.customAccentColorLabel.text"),
18            MaterialThemeBundle.message("MTForm.customTreeIndentCheckbox.text"),
19            MaterialThemeBundle.message("MTForm.decoratedFoldersCheckbox.text"),
20            MaterialThemeBundle.message("MTForm.directoriesColorLink.text"),
21            /* ... */
22            MaterialThemeBundle.message("MTForm.selectedIndicatorLabel.text"),
23            MaterialThemeBundle.message("MTForm.selectedThemeLabel.text"),
24            MaterialThemeBundle.message("MTForm.styledDirectoriesCheckbox.text"),
25            MaterialThemeBundle.message("MTForm.tabFontSizeCheckbox.text"),
26            MaterialThemeBundle.message("MTForm.tabHeight.text"),
27            MaterialThemeBundle.message("MTForm.tabPanel.tab.title"),
28            MaterialThemeBundle.message("MTForm.tabsDesc.textWithMnemonic"),
29            MaterialThemeBundle.message("MTForm.tabShadowCheckbox.text"),
30            MaterialThemeBundle.message("MTForm.themedScrollbarsCheckbox.text"),
31            MaterialThemeBundle.message("MTForm.themedTitleBarCheckbox.text"),
32            MaterialThemeBundle.message("MTForm.themeStatusBar.text"),
33            MaterialThemeBundle.message("MTForm.thicknessLabel.text"),
34            MaterialThemeBundle.message("MTForm.tweaksDesc.textWithMnemonic"),
35            MaterialThemeBundle.message("MTForm.upperCaseButtonsCheckbox.text"),
36            MaterialThemeBundle.message("MTForm.useMaterialFontCheckbox.text")
37            //endregion
38        )));
39
40    for (final String s : strings) {
41      processor.addOptions(s, null, displayName, MTConfigurable.ID, displayName, true);
42    }
43  }
```

A. Murgia et al. asked a resembling question and compared two StackOverflow[2] accounts run by bots [MJDV16]. Both accounts were controlled by the same bot program, but one account was disguised as a real human whereas the other account made it clear that it was a bot. Both accounts automatically answered simple questions in a set period of time. However the reactions were significantly different to the bot emulating a human and the bot acting as a bot and other StackOverflow users heavily criticized answers by the bot until it got banned. The other account stayed undetected and got positive feedback on the answers. They conclude that either humans don't trust suggestions made by bots or have higher expectations for suggestions made by bots. These results were confirmed by a second study experimenting with bots on the popular knowledge base Wikipedia[3] [CG15].

As our bot works pretty similar because he makes suggestions about code, we assume that the bias towards the bot is so big that developers automatically tend to reject pull requests.

5.4.1 Possible Improvements for Further Research

To make future evaluations more successful we recommend a few changes which could greatly effect similar studies. These recommendations face the assumptions we made before to improve results of future studies of this type. A different type of study without using pull requests would be possible, but we won't look into that. Various other authors conducted such a study, refer to Chapter 3 for more information.

Ask multiple projects if they want to be part of a study. We selected random projects on GitHub which showed recent activity. Nevertheless over 50% of pull requests were still open at the end of the evaluation period. Looking into these pull requests we could see that some of the projects didn't show any activity in the evaluation period and some of the reviewed other pull requests but ignored our pull requests.

Instead of selecting random projects for the evaluation, we recommend to randomly select at least three times the amount of projects required for the study. Selected projects can then be asked beforehand if they are fine with refactoring parts of their code. Projects could also be told that this would be part of an evaluation and refactorings will be made by a bot. This would help to filter out projects which don't respond and projects which accept the request would review the pull requests more seriously. We would eliminate the risk of projects ignoring pull requests or rejecting pull requests for other reasons than the committed changes. Sometimes public projects do not want contributions by strangers and only have a public repository to make their code public. Making pull requests to these projects would make no sense, so asking prior could be very helpful.

Manually select findings of high quality instead of using randomly choosing them. We selected random methods from all methods which were too long according to our threshold. We could see that some methods, especially very long methods, were often hard to refactor and therefore resulted in rather useless refactorings made by the bot.

[2] StackOverflow is a popular Q&A platform among developers https://stackoverflow.com/
[3] https://www.wikipedia.org/

To improve the quality of the refactoring suggestions two steps in the process should be done by a group of experts. A group of experts could be a group of at least three developers with experience in Java development. Each expert should independently select findings from the quality analysis for methods which they personally think that the method length could be lowered by applying refactorings. At the end, the group of experts needs to find consensus over the selected methods for each project.

In a second step, the experts name the extracted method independently and agree on the best name together. Establishing such a procedure would eliminate surrounding factors like method naming or the quality of the static code analysis findings. Eliminating the influence of factors we don't want to evaluate is important to limit the threads to validity in such a study.

Make pull request from a real developer account and don't mention the bot. The experiments discussed previously with two bots answering questions on StackOverflow proved that it is possible to emulate a human for simple interactions with developers [MJDV16]. The study showed that a bot acting as a human has vastly more success than a bot acting as a bot. Due to the similarities to our evaluation we think that pretending to be a human would greatly improve our data. In fact we did a lot of things in this study regarding pull requests manually and pretended to be a bot. Turning this around and interacting more with comments in pull requests would help to improve this study.

6 Conclusion and Outlook

In the last chapter we summarize the approach, implementation and evaluation presented in this work. Following this summary, we make suggestions for future work and elaborate how we would improve our existing implementation even more.

6.1 Summary

In this work we presented an approach to identify a good extract method refactoring candidate in a method and extract it using the Refactoring Bot. The Refactoring Bot is a software, developed by the university of Stuttgart, which works independently and automatically on a given project. The bot uses an existing static code analysis to identify code smells in the software and applies various refactorings.

The presented approach generates an exhaustive list of refactoring candidates in a first step followed by second step where the best candidate is selected. To generate the refactoring generates we use a statement graph, a structural source code representation, containing control flow information. All possible combinations of statements in the statement graph are candidates, if they satisfy syntactical preconditions, behaviour-preservation preconditions and quality preconditions. To select the best candidate, a scoring function is used. The scoring function evaluates every candidate, giving them a score depending on length, complexity, parameters and semantics. The candidate with the highest score will be the selected refactoring candidate.

We added an implementation of the approach into the existing Refactoring Bot framework. The implementation extracts methods based on existing findings from quality analysis. Refactoring suggestions are directly implemented in the corresponding source code. The Refactoring Bot then automatically creates pull requests for each extracted method.

The evaluation we conducted had the goal to find a possible correlation between the length of a original method and the acceptance of a pull request which applied an extract method refactoring to that method. We selected ten popular random open-source projects from GitHub and analyzed them using SonarQube. The Refactoring Bot with the implemented extract method refactoring was used to refactor the code based on the findings from the code analysis. We would then manually create pull requests to the original repositories with the changes by the bot, but pretended to be the Refactoring Bot.

Unfortunately due to unforeseen factors we could not collect enough data to make reliable statements regarding our research questions. We assume that the good code quality of the examined projects, the bias of developers towards bots and generally low acceptance rate of pull requests on GitHub hindered the success of the evaluation. We recommend future research to communicate with projects

before making pull requests, to manually choose the code smells which should be refactored and to not pretend to be a bot but instead interact with the project maintainers while reviewing pull requests.

6.2 Future Work

Based on the experiences made during this work we already mentioned areas which are open for improvement and future work. In following paragraph we elaborate on more good opportunities for future work.

Use the implemented refactoring for other quality findings. This work is based on the "method is too long" code smell. The primary goal of refactoring suggestions is the elimination of that code smell and the implementation uses "too long method"-findings to detect methods where a refactoring is needed. As described in Chapter 2, long methods are not the only reason why extract method refactorings are used in software development. Long methods are also not the only reason that can be detected with static code analysis.

The cognitive complexity code smell for example is measurable with static code analysis, already included in tools like SonarQube and most of the time resolved using an extract method refactoring. We already evaluate the cognitive complexity in the scoring function, so technically the cognitive complexity measurement could be used side by side with the method too long measurement. We recommend to adjust the quality preconditions for cognitive complexity findings to set a required minimal complexity reduction for a candidate to be valid when using the cognitive complexity code smell as primary finding.

Improve the scoring algorithm and extend the cognitive complexity implementation. Another starting point for future work could be the scoring algorithm in general. It can be improved in many ways, for example by adding more scoring categories, or improving the existing ones. We used a simplified version of cognitive complexity in our approach. To improve the complexity scoring the complete cognitive complexity algorithm proposed by [Cam17] should be implemented. This is required if the presented approach would be used to improve complexity code smells as suggestes above.

Additionally to the complexity rating, the semantics score could be improved greatly since it currently only considers empty lines and comment lines before and after the candidate. They give a useful indication about the semantics of the method but can be improved with suggestions from other authors [CAC+17; YLN09].

Implement an algorithm for method naming. In our implementation, the extracted method is given a generic name and has to be named manually by a developer. The name is an important part of an useful method extraction and only an automatic naming algorithm would make the method extraction refactoring a fully autonomous process. The created naming scheme presented in Chapter 5 could be helpful for future work.

Implement a lightweight control flow analysis solution. Besides the generic naming, our implementation also lacks in a different area. We use third party libraries to do the control flow analysis resulting in the CFG. Because complete control flow analysis makes deep type checks and follows function calls outside of the current class and file they usually work on compiled code. The Refactoring Bot framework on the other hand is not optimal to compile external projects which can lead to exceptions during the control flow analysis. To tackle this problem, a custom lightweight implementation of the control flow analysis could be implemented which does not leave the current context during the control flow graph building process.

Redo the evaluation with suggested improvements. The evaluation is a great point to start future work. As already described in Chapter 5 we were unable to extract enough data to answer our research questions. To answer the research questions, the evaluation has to be redone with a different study setup. We made assumptions on why our collected data was not sufficient and compared our opinion to results from different independent studies. Future work could keep the general study setup and only tweak minor things following our suggestions made in the same chapter. Results from a successful study could help to improve the Refactoring Bot by adjusting default rules for long methods in the quality analysis or complete customize rules for applying an extract method refactoring.

Furthermore different studies regarding the precision and recall of our improved scoring function could be conducted. R. Haas also suggests in his paper to do more research regarding the scoring function [HH16]. Evaluating the scoring function in detail could lead to further improvements and better refactoring suggestions in the end for example by evaluating what the perfect weight distribution for each scoring parameter is.

Bibliography

[AHM+08] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, W. Pugh. "Using static analysis to find bugs". In: *IEEE software* 25.5 (2008), pp. 22–29 (cit. on p. 21).

[All70] F. E. Allen. "Control flow analysis". In: *ACM Sigplan Notices*. Vol. 5. 7. ACM. 1970, pp. 1–19 (cit. on p. 22).

[Bra17] A. S. E. R. G. D. U. Brazil. *What are the most common refactoring operations performed by GitHub developers?* 2017. URL: https://medium.com/@aserg.ufmg/what-are-the-most-common-refactorings-performed-by-github-developers-896b0db96d9d (cit. on p. 17).

[BWYS10] S. Butler, M. Wermelinger, Y. Yu, H. Sharp. "Exploring the Influence of Identifier Names on Code Quality: An Empirical Study". In: *2010 14th European Conference on Software Maintenance and Reengineering*. Mar. 2010, pp. 156–165. DOI: 10.1109/CSMR.2010.27 (cit. on p. 45).

[CAC+17] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, P. Avgeriou. "Identifying extract method refactoring opportunities based on functional relevance". In: *IEEE Transactions on Software Engineering* 43.10 (2017), pp. 954–974 (cit. on pp. 28, 30, 33, 39, 54).

[Cam16] G. A. Campbell. *Cognitive Complexity, Because Testability != Understandability*. 2016. URL: https://blog.sonarsource.com/cognitive-complexity-because-testability-understandability/ (cit. on p. 38).

[Cam17] G. A. Campbell. *Cognitive Complexity-A new way of measuring understandability*. Tech. rep. Technical Report. SonarSource SA, Switzerland. https://www. sonarsource. com …, 2017 (cit. on pp. 38, 54).

[CG15] M. Clément, M. J. Guitton. "Interacting with bots online: Users' reactions to actions of automated programs in Wikipedia". In: *Computers in Human Behavior* 50 (2015), pp. 66–75 (cit. on p. 50).

[CP13] G. Campbell, P. P. Papapetrou. *SonarQube in action*. Manning Publications Co., 2013 (cit. on p. 24).

[DJH+08] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, M. Pizka. "Tool support for continuous quality control". In: *IEEE software* 25.5 (2008), pp. 60–67 (cit. on p. 17).

[DSTH12] L. Dabbish, C. Stuart, J. Tsay, J. Herbsleb. "Social coding in GitHub: transparency and collaboration in an open software repository". In: *Proceedings of the ACM 2012 conference on computer supported cooperative work*. ACM. 2012, pp. 1277–1286 (cit. on p. 47).

[EGK+01] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, A. Mockus. “Does code decay? assessing the evidence from change management data”. In: *IEEE Transactions on Software Engineering* 27.1 (2001), pp. 1–12 (cit. on p. 17).

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999 (cit. on p. 17).

[FOW87] J. Ferrante, K. J. Ottenstein, J. D. Warren. “The program dependence graph and its use in optimization”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9.3 (1987), pp. 319–349 (cit. on p. 29).

[GL91] K. B. Gallagher, J. R. Lyle. “Using program slicing in software maintenance”. In: *IEEE transactions on software engineering* 17.8 (1991), pp. 751–761 (cit. on p. 29).

[HH16] R. Haas, B. Hummel. “Deriving extract method refactoring suggestions for long methods”. In: *International Conference on Software Quality*. Springer. 2016, pp. 144–155 (cit. on pp. 27, 33, 37, 39, 40, 43, 55).

[HWY09] T. Honglei, S. Wei, Z. Yanan. “The Research on Software Metrics and Software Complexity Metrics”. In: *2009 International Forum on Computer Science-Technology and Applications*. Vol. 1. Dec. 2009, pp. 131–136. DOI: 10.1109/IFCSTA.2009.39 (cit. on p. 38).

[Joh78] S. C. Johnson. “Lint, a C Program Checker”. In: *COMP. SCI. TECH. REP*. 1978, pp. 78–1273 (cit. on p. 21).

[KZN12] M. Kim, T. Zimmermann, N. Nagappan. “A field study of refactoring challenges and benefits”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM. 2012, p. 50 (cit. on p. 24).

[LH96] L. Larsen, M. J. Harrold. “Slicing object-oriented software”. In: *Proceedings of IEEE 18th International Conference on Software Engineering*. IEEE. 1996, pp. 495–505 (cit. on p. 29).

[Lou05] P. Louridas. “JUnit: unit testing and coding in tandem”. In: *IEEE Software* 22.4 (2005), pp. 12–15 (cit. on p. 21).

[Lou06] P. Louridas. “Static code analysis”. In: *IEEE Software* 23.4 (2006), pp. 58–61 (cit. on p. 21).

[LR06] M. Lippert, S. Roock. *Refactoring in large software projects: performing complex restructurings successfully*. John Wiley & Sons, 2006 (cit. on p. 33).

[LV97] F. Lanubile, G. Visaggio. “Extracting reusable functions by flow graph based program slicing”. In: *IEEE Transactions on Software Engineering* 23.4 (1997), pp. 246–259 (cit. on p. 29).

[Mar01] K. Maruyama. “Automated method-extraction refactoring by using block-based slicing”. In: *ACM SIGSOFT Software Engineering Notes*. Vol. 26. 3. ACM. 2001, pp. 31–40 (cit. on p. 30).

[Mar02] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002 (cit. on p. 30).

[Mar09] R. C. Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009 (cit. on p. 33).

[MG01] N. Mittal, V. K. Garg. “Computation slicing: Techniques and theory”. In: *International Symposium on Distributed Computing*. Springer. 2001, pp. 78–92 (cit. on p. 29).

[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*. ACM. 2016, pp. 1272–1279 (cit. on pp. 50, 51).

[MLCP10] R. Marticorena, C. López, Y. Crespo, F. J. Pérez. “Refactoring generics in JAVA: a case study on Extract Method”. In: *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE. 2010, pp. 212–221 (cit. on p. 17).

[MT04] T. Mens, T. Tourwé. “A survey of software refactoring”. In: *IEEE Transactions on software engineering* 30.2 (2004), pp. 126–139 (cit. on p. 17).

[Opd92] W. F. Opdyke. “Refactoring object-oriented frameworks”. In: (1992) (cit. on p. 17).

[RR14] M. M. Rahman, C. K. Roy. “An insight into the pull requests of github”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM. 2014, pp. 364–367 (cit. on p. 47).

[SAOB02] I. Stamelos, L. Angelis, A. Oikonomou, G. L. Bleris. “Code quality analysis in open source software development”. In: *Information Systems Journal* 12.1 (2002), pp. 43–60 (cit. on p. 48).

[SE14] D. Steidl, S. Eder. “Prioritizing maintainability defects based on refactoring recommendations”. In: *Proceedings of the 22nd International Conference on Program Comprehension*. ACM. 2014, pp. 168–176 (cit. on pp. 27, 33).

[Sha] T. Sharma. “Revisiting LCOM”. In: () (cit. on p. 31).

[Sha12] T. Sharma. “Identifying extract-method refactoring candidates automatically”. In: *Proceedings of the Fifth Workshop on Refactoring Tools*. ACM. 2012, pp. 50–53 (cit. on p. 29).

[Spi05] D. Spinellis. “Version control systems”. In: *IEEE Software* 22.5 (2005), pp. 108–109 (cit. on p. 24).

[Str14] F. Streitel. “Incremental language independent static data flow analysis”. PhD thesis. Citeseer, 2014 (cit. on p. 34).

[STV14] D. Silva, R. Terra, M. T. Valente. “Recommending automated extract method refactorings”. In: *Proceedings of the 22nd International Conference on Program Comprehension*. ACM. 2014, pp. 146–156 (cit. on pp. 27, 31, 33, 34, 39).

[STV16] D. Silva, N. Tsantalis, M. T. Valente. “Why we refactor? confessions of github contributors”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2016, pp. 858–870 (cit. on pp. 17, 18, 23, 24).

[SW03] J. Shao, Y. Wang. “A new measure of software complexity based on cognitive weights”. In: *Canadian Journal of Electrical and Computer Engineering* 28.2 (2003), pp. 69–74 (cit. on p. 38).

[TC11] N. Tsantalis, A. Chatzigeorgiou. “Identification of extract method refactoring opportunities for the decomposition of methods”. In: *Journal of Systems and Software* 84.10 (2011), pp. 1757–1782 (cit. on pp. 28, 29, 31).

[Tip94] F. Tip. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica, 1994 (cit. on p. 29).

[VCN+12] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, R. E. Johnson. "Use, disuse, and misuse of automated refactorings". In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press. 2012, pp. 233–243 (cit. on p. 24).

[Wag] S. Wagner. *Software product quality control*. Springer (cit. on p. 17).

[WB19] M. Wyrich, J. Bogner. "Towards an Autonomous Bot for Automatic Source Code Refactoring". In: 2019 (cit. on pp. 24, 25).

[WDS+18] M. WESSEL, B. M. DE SOUZA, I. STEINMACHER, I. S. WIESE, I. POLATO, A. P. CHAVES, M. A. GEROSA. "The Power of Bots: Understanding Bots in OSS Projects". In: *Proceedings of the ACM on Human-Computer Interaction* 2 (2018), pp. 1–19 (cit. on p. 48).

[Wei81] M. Weiser. "Program slicing". In: *Proceedings of the 5th international conference on Software engineering*. IEEE Press. 1981, pp. 439–449 (cit. on p. 29).

[WKK07] D. Wilking, U. F. Kahn, S. Kowalewski. "An Empirical Evaluation of Refactoring." In: *e-Informatica* 1.1 (2007), pp. 27–42 (cit. on p. 17).

[YLN09] L. Yang, H. Liu, Z. Niu. "Identifying fragments to be extracted from long methods". In: *2009 16th Asia-Pacific Software Engineering Conference*. IEEE. 2009, pp. 43–49 (cit. on pp. 28, 39, 54).

All links were last followed on April 15, 2019.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature