Automatic Motivation Detection for Extract Method Refactoring Operations

Mohammad Sadegh Aalizadeh

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Master of Applied Science (Computer Science) at
Concordia University
Montréal, Québec, Canada

August 2021

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Mohammad Sadegh Aalizadeh**

Entitled: **Automatic Motivation Detection for Extract Method Refactoring Operations**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Tse-Hsun Chen

_____ External

_____ Examiner
Dr. Weiyi Shang

_____ Examiner
Dr. Tse-Hsun Chen

_____ Thesis Supervisor
Dr. Nikolaos Tsantalis

Approved by _____
Dr. Leila Kosseim, Graduate Program Director

August 19, 2021 _____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Automatic Motivation Detection for Extract Method Refactoring Operations

Mohammad Sadegh Aalizadeh

Refactoring is a common maintenance practice that enables developers to improve the internal structure of a software system without altering its external behaviour. In this study we propose a novel method to automatically detect 11 motivations driving the application of EXTRACT METHOD refactoring operations. We conduct a large-scale study on 325 open-source Java repositories to automatically detect the motivations of 346K EXTRACT METHOD refactoring instances. Previous studies have been merely based on surveys, manual analysis of pull requests or commit-messages to detect the motivations of developers. In this study we develop motivation detection rules to automatically extract the developer motivations based on the context of a refactoring operation in the commit. We find that the top four motivations for EXTRACT METHOD refactoring is to *introduce reusable methods*, *remove duplication*, *facilitate the implementation of new features and bug fixes*, and *decompose long methods to improve their readability*. There is an association between the removal of duplication and the introduction of reusable methods in the refactoring instances with multiple motivations. The findings of this study provide essential feedback and insight for the research community, refactoring recommendation tool builders, and project managers, to better understand why and how developers perform EXTRACT METHOD refactorings and help them build refactoring tools tailored to the needs and practices of developers.

Acknowledgments

I would like to express my best gratitude and thanks to my supervisor, Prof. Nikolaos Tsantalis. This thesis could not be possible without his compassionate guidance and invaluable motivational insights that opened new horizons of knowledge towards me.

I would also like to express my appreciation to my family, my wife, Faezeh and my daughter, Fatemeh for all the beautiful moments and the serenity and aspiration they provided for me to concentrate and proceed in all the hard times.

Finally, I would like to thank my colleagues, Hassan Mansour, Mehran Jodavi ,Mosabbir Khan Shiblu and Ameya Ketkar that shared their best experiences and were amazing in teamwork and helped me to learn a lot in my journey at Concordia.

Thank you.

Mohammad Sadegh Aalizadeh

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Software systems continuously evolve and adapt to new requirements implemented through maintenance tasks, such as feature additions or bug fixes (Chen et al., 2016). In such environments, refactoring has been adopted by developers as a useful practice to improve the internal structure of software without altering its functional behaviour (Kaya et al., 2018).

Many research studies have been conducted to build refactoring recommendation and prediction tools based on product and process metrics (Pantiuchina et al., 2020; Nyamawe et al., 2018). Next generation tools and techniques, such as user-aware intelligent refactoring bots, automatic advisors and smart search-based refactoring methods, have been developed to optimize the refactoring recommendation process according to the developer decisions and feedback (Ivers et al., 2020; Stefano et al., 2020).

However, there is a lack of large-scale empirical research that signifies the main interest points and reasons behind developer refactoring activities. Therefore, in this study we propose a novel context-aware approach to automatically find the developer motivations behind applied refactoring operation. We focus our study on the EXTRACT METHOD

refactoring that is known as the Swiss army knife of refactoring due to its applicability in many different scenarios (Silva et al., 2016; Hora and Robbes, 2020).

Previous research mainly focused on surveys or interviews and investigations of pull request reviews and commit messages to generate taxonomies and categorize refactoring motivations (Bavota et al., 2015; Silva et al., 2016; Pantiuchina et al., 2020). These research works discovered 11 major motivations behind the application of EXTRACT METHOD refactoring operations. These motivations are detected based on source code changes in commits. Therefore, we formulate detection rules and use them at the core of our automatic motivation detection tool to analyze the context of EXTRACT METHOD refactoring instances and determine the motivation(s) of the developer. We have not analyzed higher-level design artifacts, such as UML models or architecture documents to detect motivations, which can be studied in the scope of architectural refactoring.

1.2 Problem Statement

In this study we will investigate EXTRACT METHOD refactoring motivations from different aspects. We initially evaluate the accuracy of our motivation detection tool in RQ1 according to a dataset of actual developer responses about the reasons they applied EXTRACT METHOD refactorings in their projects. The remaining research questions will shed light to the most prevalent EXTRACT METHOD refactoring motivations (RQ2), investigate what self-affirmed refactoring-related keywords AlOmar et al. (2019a) appear in the messages of commits including EXTRACT METHOD refactorings (RQ3) and investigate the associations between different motivations in the cases of refactoring activities with multiple detected motivations (RQ4).

RQ1: How accurate is our EXTRACT METHOD refactoring motivation detection tool?

The automatic motivation detection tool utilizes an enriched source code representation model, which provides the required contextual information about the EXTRACT METHOD

refactoring operations. In chapter 3, we define detection rules for each motivation by analyzing the context in which an EXTRACT METHOD refactoring instance is applied. The accuracy and efficiency of the refactoring detection process can affect the motivation detection results. Therefore, we relied on the current state-of-the-art refactoring mining tool, RefactoringMiner (Tsantalis et al., 2020; Tsantalis et al., 2018), which has an overall precision of 99.6% and a recall of 94%. Furthermore, it is essential to validate the results of the automatic motivation detection tool by examining the responses from developers about the reason behind the application of specific refactoring instances. Therefore, we reconstructed a training dataset of developer responses about the reasons they applied certain EXTRACT METHOD refactoring operations in specific commits of their projects (Silva et al., 2016). The motivation detection algorithm is optimized on this training dataset to reach an acceptable accuracy level for a large scale study. We further validate the accuracy of the motivation detection algorithm on a testing dataset of manually analyzed and tagged refactoring motivations at pull-request level (Pantiuchina et al., 2020). The high accuracy obtained at this stage will have a direct effect on the analysis of the results to answer the research questions of the study.

RQ2: What are the most prevalent motivations behind the application of EXTRACT METHOD refactoring operations?

Developers commonly apply refactoring techniques to maintain and improve the design quality of software systems during the development period. Automated tools have been developed to detect and recommend refactoring opportunities to assist developers in maintenance tasks, but they are often underused. To tailor these tools to the actual needs and practices of the developers, it is essential to find the most common reasons that motivate developers to refactor code. In this research question we will answer what specific reasons have the highest prevalence when a developer decides to perform EXTRACT METHOD refactoring operations.

There have been a few studies (Silva et al., 2016; Pantiuchina et al., 2020) that categorized possible motivations and reasons for refactoring through manual investigation and surveys to build motivation taxonomies. However, these studies due to their manual analysis nature focused on a small number of project commits and refactoring instances. We automated the detection process of 11 motivations that were recognized in these taxonomies for the EXTRACT METHOD refactoring operations found in a large number of analyzed commits (346K commits from 325 open-source Java repositories). The results for this research question will give us the big picture about the reasons that motivate developers to extract methods.

RQ3: What are the characteristics of the EXTRACT METHOD refactoring instances having *Facilitate Extension* as motivation?

Refactoring is not always done for the sole purpose of design quality improvements and at times can be interleaved with other common development tasks, such as bug fixing or adding new features.

In cases where an EXTRACT METHOD refactoring is tangled with such tasks, we can categorize motivations based on the self-affirmed commit messages (AlOmar et al., 2019a; AlOmar et al., 2021). For instance, bug-removal related keywords, such as *fix*, *bug*, *issue*, *remove*, *cleanup* can be used to detect the code extension motivation at a more fine-grained level (i.e., the refactoring operation facilitates a bug fix). On the other hand, collateral refactoring tasks are done without any clear discussions or messages, and therefore it is more complicated to find the exact intent of the developer without manual analysis.

We have formulated a thorough detection rule for the *Facilitate Extension* motivation (Section 3.3.4), which captures EXTRACT METHOD refactoring instances with additional statements contributing to a bug fix or a feature addition. The rule is optimized to find statements that are directly contributing to functionality extension. At the same time, it filters out the statements inside the extracted method that have a minimal effect on adding

a new feature or fixing a bug.

We will discuss the results of the *Facilitate Extension* motivation detection and the features that are considered in the detection rule. Furthermore, we will discuss what self-affirmed refactoring keywords are commonly used in the messages of commits that include EXTRACT METHOD refactorings with a *Facilitate Extension* motivation.

RQ4: Are there concurrent motivations for extracting methods?

The motivation detection process is designed to analyze the EXTRACT METHOD refactoring context to find the evidence related to each defined motivation. Therefore at times it is possible that multiple motivations are reported for a single EXTRACT METHOD refactoring instance. In such circumstances, various post-processing rules are applied to eliminate the conflicting motivations that have less priority.

We will study the combinations of refactoring motivations that appear concurrently and analyze their co-occurrence to understand the possible relation between different refactoring motivations.

1.3 Objectives and Contributions

In this thesis, we formulate the logical rules to detect the motivations of developers that perform EXTRACT METHOD refactoring operations. The proposed approach and developed tool for the automatic refactoring motivation detection can be utilized in refactoring-related studies that require more detailed information about the intention of the developers when performing refactoring activities. It can be also used for building smart refactoring advisors and recommendation tools to address more effectively the developer needs and practices.

Our most significant and important contributions are listed as follows:

- We build a general motivation detection approach for refactoring operations.
- We formulate EXTRACT METHOD motivation detection rules based on the context of

a refactoring instance for 11 major motivations based on two prominent studies that provided refactoring motivation taxonomies through developer surveys and manual analysis.

- We develop an open-source automated motivation detection tool that can detect EXTRACT METHOD motivations, named Motivation Extractor, which is based on RefactoringMiner 2.0 (Aalizadeh, 2021; Tsantalis et al., 2020).
- We manually analyze the Pull Requests (PRs) studied by Pantiuchina et al. (2020), which contain EXTRACT METHOD refactoring instances in order to validate the high precision and recall of our automated tool that was achieved on a training dataset from real developer responses Silva et al. (2016).
- We present and discuss empirical results found by analyzing 346K commits from 325 open-source Java repositories.

1.4 Outline

The rest of the thesis is organized as follows. In chapter 2 we discuss the various related works that have investigated the developer motivations to perform refactoring operations during maintenance tasks. In chapter 3 we present the details of the motivation detection model for EXTRACT METHOD refactoring and the detection rules for 11 different motivations. These rules are formulated based on the context of a refactoring instance. We will demonstrate the rules using decision trees that clarify how different detection rule conditions are matched to determine the developer motivations. In chapter 4 we present the results of our large scale study and the answers to our research questions. Threats to the validity are discussed further in chapter 5. Finally, we conclude the thesis in chapter 6.

Chapter 2

Literature Review

There has been a tremendous body of research in the field of software refactoring in the past 30 years. The term *refactoring* was first coined by Opdyke (1990) to describe source code transformations that can be automated and used to improve understandability and reusability. Since then there have been numerous papers about the different aspects of refactoring activity, such as the refactoring process, goals and incentives, automation techniques, and recommendation systems.

Refactoring and its applications have been studied on various software artifacts from source code to models and design patterns (Derezińska, 2017). Tanhaei (2020) propose a method to automate architectural refactoring based on stakeholder quality attributes.

Researchers have used different methods to study refactoring activities. For instance, search-based techniques are used to find an optimized refactoring sequence for automated design quality optimization (Mohan and Greer, 2017). Data mining techniques are utilized to predict the effect of refactoring on code quality during software evolution and also analyze the impact of refactoring on design quality metrics (AlOmar et al., 2019b). Some studies use fuzzy logic for automatic code smell detection and refactoring, as well as formal verification methods to ensure refactoring has not caused behavioural changes (Nasagh et al., 2021; LUO et al., 2011). Refactoring actions are performed for multiple

objectives. Refactoring can improve the internal quality attributes of the software like cohesion, coupling, complexity, inheritance depth, and size (Chávez et al., 2017). It can also improve external software quality attributes that are indirectly measured through code metrics like reusability, flexibility, understandability, functionality, extensibility, and effectiveness (Vashisht et al., 2018). Performance-related measures can also be improved by refactoring at various scales. Arcelli et al. (2018) use performance anti-patterns to perform non-functional driven refactoring on UML models. Refactoring is further used for security purposes. Abid et al. (2020) study the impact of refactoring on security metrics related to data access, when improving quality attributes from the QMOOD model.

2.1 Refactoring mining tools

Refactoring actions can be performed manually by editing the code or through the automation support provided in IDEs. Various refactoring detection tools have been developed and researchers used methods to detect the application of a wide range of refactoring types in software systems.

Dig et al. (2006) developed Refactoring Crawler that uses a combination of syntactic and semantic analysis for detecting and refining of refactorings with an accuracy above 85%. Prete et al. (2010) used a template-based refactoring reconstruction approach that uses logic rules to describe code elements and their dependencies (adopted from their previous work on LSdiff (Kim and Notkin, 2009)) and a logic programming engine to infer concrete refactorings. Their tool, Ref-Finder, can identify complex refactoring instances that consist of multiple atomic refactoring types between program versions, and achieves a precision of 0.79 and recall of 0.95.

Silva and Valente (2017) propose RefDiff, an automated tool that employs heuristics based on static analysis and code similarity to detect refactoring operations between two successive revisions of a Git repository. This tool finds 13 well-known refactoring types

and has a precision of 100% and a recall of 88%. It is extended in RefDiff 2.0 to support multiple programming languages using the Code Structure Tree (CST) to abstract the specifications of programming languages. In addition to Java, RefDiff 2.0 supports JavaScript and C programming languages (Silva et al., 2020).

Tsantalis et al. (2018) designed, implemented and evaluated RefactoringMiner, which detects refactoring operations with a precision of 98% and a recall of 87%. RefactoringMiner uses in its core an AST statement matching algorithm to detect refactorings in project revisions without requiring user-defined thresholds. In a more recent work Tsantalis et al. (2020), RefactoringMiner has been extended to support low-level refactorings that are performed within the body of methods. The accuracy of the tool was further improved reaching a precision of 99.6% and a recall of 94%.

2.2 Refactoring Motivation

In the human behaviour domain of refactoring, the developer's motivation is recognized as a key factor that has theoretical and practical implications on software development and maintenance.

Liu and Liu (2016) conduct an interview with 25 developers to find the motivations of developers when they perform EXTRACT METHOD refactorings. Among all motivations reuse, decomposition of long methods, clone resolution are known to be the major motivations. They analyze 7 open source repositories to validate the results. They find that EXTRACT METHOD opportunities should not merely be affected by the inner structure of the methods (e.g., complexity or size). Due to the high prevalence of immediate reuse, they suggest refactoring recommendation tools can utilize recognition or prediction of reuse to improve chances of developers applying the recommended EXTRACT METHOD refactorings.

Wang (2009) conducts a semi-controlled interview and presents an empirical model of refactoring that describes the relationship between motivators, contextual factors, and actions. This research only considers psychological and personal values along with cultural, economic and other aspects of refactoring in organizations to determine motivations.

Silva et al. (2016) applied thematic analysis on the responses received from 222 developers contributing in 124 open source projects about why they perform specific refactoring operations. They used RefactoringMiner to detect refactorings and then manually inspected true positives. A catalogue of 44 distinct motivations for 12 types of refactoring operations was compiled. The results show refactorings are mainly performed due to requirement changes rather than code smells. For instance, among all 11 motivations for EXTRACT METHOD refactoring only two of them (*decompose method to improve readability* and *remove duplication*) are targeting code smells. Their study also shows that 55% of the studied refactoring instances are performed manually, mainly because of the developers' lack of trust in automated tools. This research has been the main motivation of our research and it also explicitly proposes that future research should refocus from code-smell-oriented to maintenance-task-oriented refactoring solutions. For this purpose, we automate the process of detecting motivations to facilitate analyzing the motivation categories of refactoring. We believe our empirical findings will help to build smarter recommendation tools that can target the actual developer needs.

Kim et al. (2014) present a field study at Microsoft to quantitatively assess the benefits and challenges of refactoring from the developer perception. They utilize three complementary research methods: survey, semi-structured interviews with professional software engineers and quantitative analysis of version history data. In the survey, they find that developers perceive substantial costs and risks when they refactor, and that refactoring definition is not confined to a rigorous semantic-preserving code transformation. Furthermore,

in the quantitative analysis of Windows 7 version history, they find that the top 5% of preferentially refactored modules experience a higher reduction of inter-module dependencies and complexity measures. This study investigates system-wide metrics such as defects, inter-module dependencies, size and locality of code changes, complexity, test coverage, people and organization metrics. However, it has some limitations, as it analyzes only the commits using the "refactor" keyword in their message, and thus misses a substantial undocumented refactoring activity. Moreover, the quantitative analysis focuses on the version history of a single large system, namely Windows 7, which is a threat to the external validity of the findings.

Bavota et al. (2015) analyze refactoring operations over 63 releases of three Java open source projects. They utilize Ref-Finder to detect refactoring operations and build logistic regression models to find code smells and quality metrics that are significantly correlated with refactoring types. They highlight WMC metric as the only exception for quality metrics that has a clear relationship with refactoring. They also find that only some of the analyzed code smells such as *Blob*, *Long Method*, *Spaghetti Code* and *Feature Envy* increase the chance of affected classes being refactored. Among all refactoring operations detected in the release history of the three examined Java projects, 42% are performed on code smells and 7% actually removed the code smells. Manual analysis is performed to improve the detection of code smells. They conclude that the developers' point-of-view of classes in need of refactoring does not always match with quality indicators.

Pantiuchina et al. (2020) present a large-scale study to quantitatively and qualitatively investigate why developers refactor in order to generalize and complement previous studies that are based on surveys. They mine 287,813 refactoring operations and build a mixed explanatory logistic regression model to find the correlation between 42 product and process related metrics and refactoring operations in 150 open-source projects. Among product-related factors code readability is mostly correlated with refactoring. In process-related

metrics, source code change, fault-proneness, and the experience of developers changing a code component, play a significant role in triggering refactoring. They also manually tag a randomly stratified sample of 551 pull requests with 8,108 refactorings to build a taxonomy of 67 motivations in 6 main categories. This study demonstrates the factors and reasons that make a refactoring meaningful from the developers' perspective and can be utilized to make better refactoring recommendation tools to address specific developer needs.

AlOmar et al. (2021) propose a two-step approach to automatically classify self-affirmed refactoring (SAR) operations that contain developer-related events documented in commit messages. They build a model with a high F-measure that combines N-Gram TF-IDF feature selection and binary and multi-class classifiers that outperforms pattern-based and random classifier approaches in classifying SAR commits into internal/external quality attributes and code smells categories. Their method is solely based on the commit messages to find the intent or rationale behind the refactoring.

Paixão et al. (2020) inspect and classify developer intents behind refactoring during code reviews in 7 distinct categories. They manually analyze code changes in 1,780 code reviews that employ refactoring operations. They study the sequence, composition and evolution of refactoring operations in code reviews. They find that developers most often apply refactoring with other code changes to support feature additions or bug fixes.

Our work is the first to automate the detection of refactoring motivations by analyzing the refactoring-related edits and the context in which the refactoring operations are applied within a commit. Such fine-grained analysis allows to extract the refactoring motivations with a high accuracy. Moreover, it enables the collection of refactoring motivations in a large-scale, as it has been implemented as a fully automated process.

Chapter 3

Research Methodology

In this chapter, we first present a general method for the automatic refactoring motivation detection. Using this method, we came up with logical rules and decision trees customized to detect 11 major motivations for EXTRACT METHOD refactorings. The motivation detection rules are implemented in an open-source tool named *Motivation Extractor*, which is built on top of RefactoringMiner Tsantalis et al. (2020) and is available on GitHub (Aalizadeh, 2021).

3.1 Automatic Refactoring Motivation Detection

We have developed a general process for the automatic detection of refactoring motivations. This process can be further extended and applied to detect the motivations of different refactoring types. Figure 1 shows the two main steps of our method. In the first step, we build motivation detection rules in an iterative process and optimize them based on a training set that is constructed from the motivations given by the actual developers who performed specific refactoring operations in open source projects Silva et al. (2016). In the second step, we use the optimized detection rules and perform a large scale motivation detection in the commit history of open-source repositories.

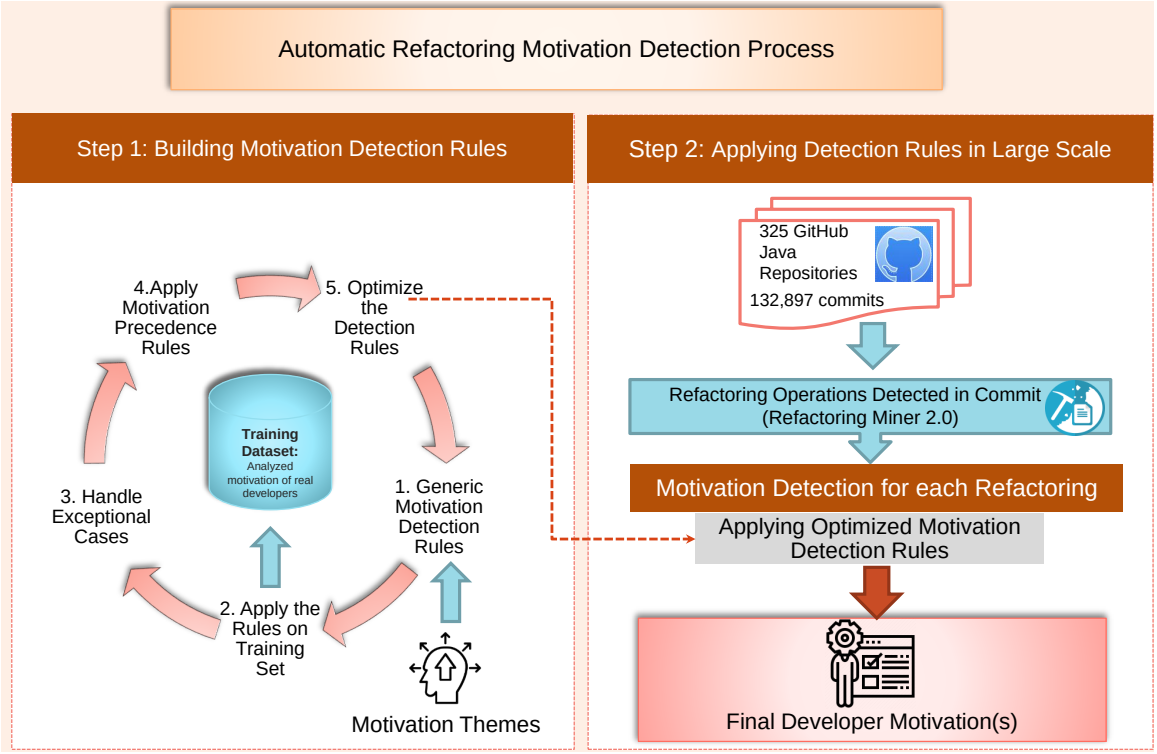

Figure 1: Refactoring Motivation Detection Process in Large Scale

3.2 Step 1: Building Motivation Detection Rules

The first step of the automatic motivation detection process is to build the motivation detection rules from the generic descriptions of refactoring motivation themes, which are obtained from the Silva et al. (2016) study. In this study, the authors monitored 185 open-source projects for commits including refactoring operations. When such commit was detected, the commit author was contacted via email to explain the reasons behind the application of the detected refactoring operations. By applying thematic analysis on the responses received from 222 developers, Silva et al. created a catalogue of 44 distinct motivations for 12 well-known refactoring types. Each motivation theme is accompanied with a number of refactoring instances, which were applied for the reason described in the corresponding motivation description.

We initially build a generic detection rule based on the motivation description and then optimize it by handling exceptional cases and conflicts with other co-detected motivations

in an iterative process as shown in Figure 1.

3.2.1 Generic Motivation Detection Rules

The generic motivation detection rules are initially formed based on the description of the motivations themes in natural language. Table 1 shows the descriptions of the motivation themes for the EXTRACT METHOD refactoring type. For example, the description for *Reusable Method* is "Extract a piece of reusable code from a single place and call the extracted method in multiple places". As a result, the generic rule should check if the extracted method is called in at least one more method other than the method from which it was extracted.

These generic rules are very simple at first and are not tuned to address exceptional cases or conflicts with other co-detected motivations. Therefore, we need to use our training dataset to assess the current precision and recall of the generic rules, and guide the optimization of the rules to detect the motivations in various refactoring contexts more accurately.

3.2.2 Apply the Rules on the Training Dataset

It is necessary to ensure the compatibility of the automatically detected motivations with the actual intentions of the developers. Therefore, we apply the rules derived from the previous step on our training dataset, which includes instances of refactoring operations labeled with a motivation theme based on the explanations of the developers who actually performed these refactorings Silva et al. (2016). Since the motivation labels come directly from the developers who performed the refactorings, we consider our training dataset a reliable ground truth (i.e., oracle), based on which we can optimize the detection rules. We can compute the precision and recall of the detection rules based on this oracle, and we can find all the exceptional cases in which the generic rules missed or mislabeled the

actual developer motivation(s). We use these insights to add the missing logic to the generic detection rules and handle specific scenarios.

3.2.3 Handle Exceptional Cases

Guided by the missed and mislabeled refactoring instances from the oracle, we analyze the commits in which these refactoring were detected to find exceptional cases that should be taken into account when enhancing the detection rules. For example, a missed case for the *Reusable Method* motivation, involved an extracted method that was reused in the same method from which it was extracted from. This case helped us to improve the detection rule to handle additional calls to the extracted method that were added in the origin method after the EXTRACT METHOD refactoring.

Moreover, by analyzing individually each refactoring in a commit we might get motivations that are in disagreement with the motivations obtained when analyzing refactorings in groups. For example, when a duplicated piece of code is extracted from multiple methods, a separate EXTRACT METHOD refactoring instance is reported for each source method. Analyzing each EXTRACT METHOD instance in isolation from the rest triggers the detection of the *Reusable Method* motivation, as the corresponding rule finds that the extracted method is also called in methods other than the method from which it was extracted. However, by combining the information from multiple EXTRACT METHOD instances, we can group the instances that have exactly the same extracted method, detect the correct motivation, namely *Remove Duplication*, and discard the individually reported *Reusable Method* motivations.

We will show these exceptional cases while explaining the motivation detection rules in Section 3.3 and demonstrate how different contexts in which a refactoring is applied affect the detected motivation.

3.2.4 Filtering Motivations by Applying Precedence Rules

In some cases there are multiple detected motivations for a single refactoring instance. For example, if we have a method that is extracted in production code, and in the same commit a unit test is added calling the extracted method, two motivations will be detected, namely *Reusable Method* as the extracted method is called in a method other than the method from which it was extracted, and *Improve Testability* as a new unit test is added to test the extracted method. In that case, *Improve Testability* is a more dominant motivation, as it captures more accurately the refactoring intention of the developer, and thus has precedence over *Reusable Method*.

We present the precedence rules for each motivation in Section 3.3, and use them to filter out some co-detected motivations with lower priority for the same refactoring instance.

3.2.5 Optimize the Detection Rules

Following this iterative process for each motivation rule, we end up with a set of optimized detection rules, which maximizes the overall precision and recall on our training dataset. The achieved accuracy results on the training dataset are shown in Section 4.1.1. To make sure our detection rules are not over-fitting the training dataset, we applied them also on a testing dataset. The accuracy results on the testing dataset are shown in Section 4.1.2. The high accuracy levels achieved in both training and testing dataset gives us confidence that the automatically detected motivations in our large-scale experiment (Section 3.4) will accurately reflect the intentions of the developers, and thus our experimental results will have a strong validity.

3.3 Extract Operation Motivation Detection

In this section, we describe the motivation detection rules for the EXTRACT METHOD refactoring. We are focusing on the motivations of EXTRACT METHOD refactoring for two reasons. First, according to Negara et al. (2013) and Tsantalis et al. (2020), EXTRACT METHOD is the most commonly applied refactoring on methods. Second, according to Silva et al. (2016), EXTRACT METHOD has the most observed motivations (11 in total) compared to other refactoring types. Therefore, EXTRACT METHOD is a significant refactoring that is worth investigating the reasons driving its application at a large-scale.

Table 1 shows 11 motivation themes along with their descriptions, as documented by Silva et al. (2016). We used the descriptions of the motivation themes, along with the accompanied refactoring instances for each motivation theme, as inputs for the process explained in Section 3.2 in order to derive our motivation detection rules.

Table 1: Extract Method Motivation Themes

Extract Method Motivation	**Description**
Reusable Method	Extract a piece of reusable code from a single place and call the extracted method in multiple places.
Introduce Alternative Method Signature	Introduce an alternative signature for an existing method (e.g., with additional or different parameters) and make the original method delegate to the extracted one.
Decompose Method to Improve Readability	Extract a piece of code having a distinct functionality into a separate method to make the original method easier to understand.
Facilitate Extension	Extract a piece of code in a new method to facilitate the implementation of a feature or bug fix, by adding extra code either in the extracted method, or in the original method.
Remove Duplication	Extract a piece of duplicated code from multiple places and replace the duplicated code instances with calls to the extracted method.
Replace Method Preserving Backward Compatibility	Introduce a new method that replaces an existing one to improve its name or remove unused parameters. The original method is preserved for backward compatibility, it is marked as deprecated, and delegates to the extracted one.
Improve Testability	Extract a piece of code in a separate method to enable its unit testing in isolation from the rest of the original method.
Enable Overriding	Extract a piece of code in a separate method to enable subclasses override the extracted behavior with more specialized behavior.
Enable Recursion	Extract a piece of code to make it a recursive method.
Introduce Factory Method	Extract a constructor call (class instance creation) into a separate method.
Enable Async Operation	Extract a piece of code in a separate method to make it execute in a thread.

The motivation rules are described in two ways. First, in a formal way using logical conditions combined with AND or OR logical operators, and second in a visual way using decision trees. Each of the sub-section that follows includes the definition of one of the 11

motivation detection rules. For the formal descriptions, we use the some common general notations shown in Table 2, while specific notations applicable only for a single motivation detection are shown in the corresponding sub-sections.

Table 2: General Notations used in EXTRACT METHOD Motivation Detection Rules

Notation	Description
EM	Extract Method Refactoring instance
$m_a \rightarrow m_b$	A single call from method m_a to method m_b
$calls(C, m_b)$	A set of calls within class C to the method m_b
$calls(M, m_b)$	A set of calls from the set of methods M to method m_b
$calls(S, m_b)$	A set of calls from the set of statements to method m_b
m_a	Source Operation Before Extraction
$m_{a'}$	Source Operation After Extraction
$m_b.n$	Signature of the Extracted Method
$m_b.C$	Name of the Class containing the Extracted Method
$m_a.P$	Ordered Parameter list of m_a
$m_a.b$	Set of All statements in the m_a
$m_a.b.C$	Set of Composite statements in method m_a
$m_a.b.L$	Set of leaf statements in method m_a
$m_b.T_2$	Set of added statements(T_2) in the extracted method m_b
$m_{a'}.T_2$	Set of added statements(T_2) in the source operation after extraction $m_{a'}$
$m_b.T_2.L$	Set of leaf statements added in method m_b
$codeElementType(m_a.leaf)$	Code element type of the leaf statement in method m_a
$calls(m_a.leaf)$	Set of all invocations in the leaf statement of method m_a
M_{s_i}	Set of Methods in the parent commit s_i
C^+	Set of Added Classes in child commit s_{i+1}
$C^\sim$	Set of Modified Classes in child commit s_{i+1}
M^+	Set of Added Methods in child commit s_{i+1}
$M^\sim$	Set of Modified Methods in child commit s_{i+1}
$EM.Mapping$	Mapped code in the extracted method
$EM.Mapping.Fragment1$	Set of mapped code statements before extraction
$EM.Mapping.Fragment2$	Set of mapped code statements after extraction
$EM.Mapping.L$	Set of Leaf statements in the Extract Method Refactoring mapped code
$EM.Mapping.C$	Set of Composite statements in the Extract Method Refactoring mapped code
$EM.NotMapped.T_1$	Set of not-mapped statements that are deleted from m_a
$EM.NotMapped.T_2$	Set of all not-mapped statements that are added to $m_{a'}$, m_b
$EM.NotMapped.T_2.Child$	Set of not-mapped statements that are added to m_b
$EM.NotMapped.T_2.Parent$	Set of not-mapped statements that are added to $m_{a'}$
$T_{s_{i+1}}$	All test methods present in child commit s_{i+1}
$m_b.b.L.Return$	Set of return statements in the extracted method m_b

3.3.1 Reusable Method

Reusability is a significant external quality attribute that can be improved by refactoring (Bogart et al., 2020). EXTRACT METHOD refactoring is useful to extract smaller pieces of

functionality from longer methods for reusability purposes (Yang et al., 2009).

Generic Motivation Detection Rules

The rule to detect the *Reusable Method* motivation is shown in Table 3.

- All the changed classes and newly added classes will be examined to find invocations to the Extracted Operation.
- Additional invocations should exist outside of the Extracted Operation and Source Operation After Extraction.
- OR multiple invocations should exist within the Source Operation After Extraction.

Table 3: Reusable Method Detection Rule

$\exists\, m_r \rightarrow m_b \mid m_r \in \{C^+ \cup C^\sim\} \wedge$
(1) $m_r \rightarrow m_b \notin calls(D_{s_{i+1}}, m_b) \wedge$
(2) $(m_r \rightarrow m_b \notin calls(T_{s_{i+1}}, m_b) \wedge m_{a'} \notin T_{s_{i+1}}) \vee (m_r \rightarrow m_b \in calls(T_{s_{i+1}}, m_b) \wedge m_{a'} \in T_{s_{i+1}}) \wedge$
(3) $\neg\, m_r \rightarrow m_a$ replaced with $m_r \rightarrow m_b \wedge$
(4) $\nexists\, m_r \rightarrow m_{b'} \mid m_{b'}.n = m_b.n \wedge m_{b'}.C \neq m_b.C \wedge$
(5) $m_r \rightarrow m_b \notin calls(N_{s_{i+1}}, m_b)$
$\vee\, |calls(\{m_{a'}\}, m_b)| > 1$

$m_{b'}$: Another method with the same signature as the extracted method m_b
$D_{s_{i+1}}$: All methods from which the body of m_b was extracted in child commit s_{i+1}
$N_{s_{i+1}}$: All nested extracted methods in child commit s_{i+1}

Exceptional cases

1. Duplicate code fragments extracted from a single method or multiple methods. The call site of an extracted method which is involved in duplication removal should not be considered as reuse. (e.g., BuildCraft [1])
2. Test methods call the extracted method, but it belongs to production code. If the extracted method belongs in production code, all calls from test code are ignored

[1] https://github.com/BuildCraft/BuildCraft/commit/a5cdd8c

when assessing reusability. However, if the extracted method belongs in test code, then calls from test code are considered when assessing reusability.

3. The additional call takes place in a method which was originally calling the source method, but calls the extracted method after the refactoring. This typically happens when the entire body of a method is extracted in a new method for the purpose of deprecating the source method, or introducing an alternative signature for the source method. In such case, the additional call is ignored when assessing the reusability of the extracted method, because it simply changes the delegation from the source to the extracted method. (e.g., Google truth[2])

4. The additional call to the extracted method takes place in another class, in which a new method is added having the same signature as the extracted method. In such case, the additional call is ignored when assessing the reusability of the extracted method, since a local method is called and not the extracted one. (e.g., Intellij [3])

5. When we have nested EXTRACT METHOD refactorings (i.e., a method is extracted from the body of another extracted method), the calls to the extracted methods are not placed in the source method originally containing the extracted code, but inside the bodies of the subsequently extracted methods. As a result, although the refactoring instance reports that the nested extracted method came from the source method originally containing the extracted code, the call site to the nested extracted method is not placed in the source method. In such case, the call inside the body of a nested extracted method is ignored when assessing the reusability of the extracted method. (e.g., Checkstyle[4] , JGroup[5])

[2]https://github.com/google/truth/commit/200f157
[3]https://github.com/JetBrains/intellij-community/commit/10f769a
[4]https://github.com/checkstyle/checkstyle/commit/5a9b724
[5]https://github.com/belaban/JGroups/commit/f153375

Furthermore, The motivation detection logic for *Reusable Method* is demonstrated as the decision tree shown in Figure 2.

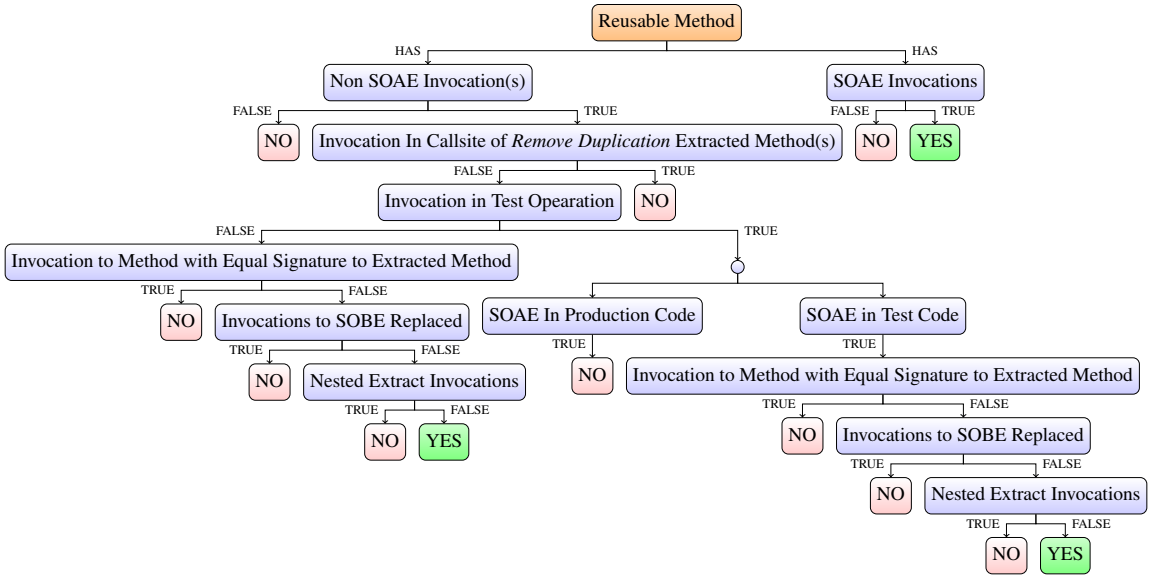

Figure 2: Reusable Method Decision Tree

Filtering the set of refactoring motivations

Precedence of Backward Compatibility and Alternative Method Signature: An extracted method is not considered as reusable, if the *Backward Compatibility* or *Alternative Method Signature* motivations are detected for the same instance.

3.3.2 Introduce Alternative Method Signature

EXTRACT METHOD refactorings are sometimes used to introduce an alternative method signature. The original method will be used as a delegate to the extracted method that may have added parameters or changed parameter types. Understanding the signature-level changes can be utilized to improve refactoring-based tools for API migration (Dig and Johnson, 2006).

Table 4: Introduce Alternative Method Signature Detection Rule

$m_{a'}.P \neq m_b.P \wedge$ $\lvert calls(\{m_{a'}\}, m_b)\rvert = 1 \wedge$ $\nexists\, leaf \in m_{a'}.b.L \mid \{calls(m_{a'}.leaf) \cap calls(\{m_{a'}\}, m_b)\} = 0 \wedge$ $\neg codeElementType(m_{a'}.leaf) = VARIABLE\_DECLARATION\_STATEMENT$

Generic Motivation Detection Rules

The rule to detect the *Alternative Method Signature* motivation is shown in Table 4.

- The extracted operation parameter types are not equal with the Source Operation After Extraction (SOAE) parameter types. Therefore, either the number of the parameters or their types changes in the extracted method.

- And Source Operation After Extraction (SOAE) should delegate and hand over the responsibility to the extracted method. Hence, we look for the invocations to the extracted method in the source operation after extraction. The number of the invocations to the extracted method should be one to consider that as delegation.

- And SOAE can include temporary variables. The source operation after extraction may include statements which declare temporary variables. Therefore, all the leaf nodes that do not subsume EXTRACT METHOD invocations in the SOAE will be examined to see if they are variable declaration statements (VDS). If all the statements in the source operation after extraction are solely used to declare temporary variables and the two previous conditions also hold true, we report the *Introduce Alternative Method Signature* motivation.

There are no exceptional cases to examine for the detection of the *Alternative Method Signature* motivation.

Figure 3 shows the decision tree for the EXTRACT METHOD refactoring operations that are performed to Introduce Alternative Method Signature.

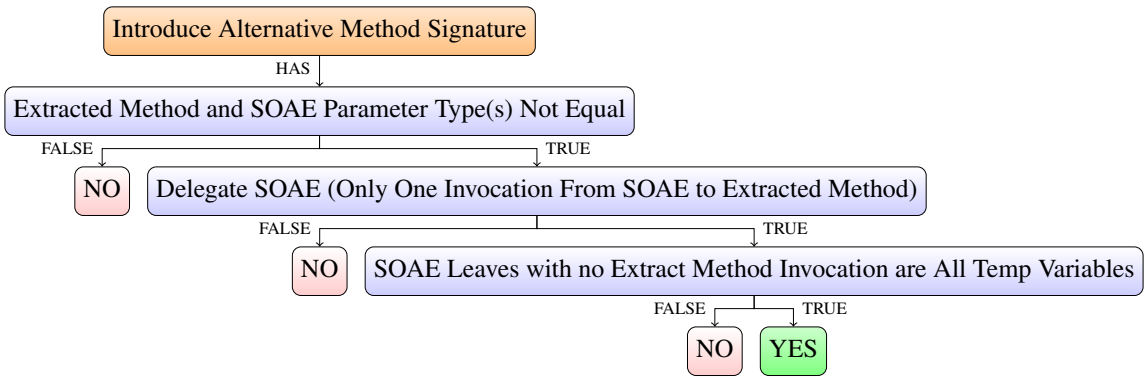

Figure 3: Introduce Alternative Method Signature Decision Tree

Filtering the set of refactoring motivations

Precedence of *Remove Duplication*: In cases where a *Remove Duplication* motivation is also detected, we do not consider *Alternative Method Signature*. Furthermore, if *Facilitate Extension* was previously detected as a motivation it is discarded.

3.3.3 Decompose Method to Improve Readability

Developers consider readability and understandability as an important quality attribute and mainly adopt refactoring for the purpose of source code understandability (Vassallo et al., 2019). EXTRACT METHOD refactorings can also be used to improve the understandability and readability of source code. A piece of code with a distinct functionality can be extracted to a separate method and this makes the original method more readable and easier to understand.

Decomposition of the original method can be performed to break it into multiple smaller extracted methods. When decomposition is performed to extract multiple methods from the source operation, it is required to analyze the list of all EXTRACT METHOD and EXTRACT AND MOVE METHOD refactoring operations together. Decomposition can also be detected with a single extracted method, if the extracted piece of code makes the original method more readable after extraction.

Table 5: Decompose to Improve Readability Detection Rule

Decompose Multiple Method to Improve Readability:
$\exists\, EM_1, .., EM_n \ni n > 1 \ \wedge\ m_{b_1} \neq .. \neq m_{b_n} \wedge m_{a_1} = .. = m_{a_n} = m_a \wedge$
$\textcircled{1}\ EM_1 \neq .. \neq EM_n \wedge$
$\textcircled{2}\ editDistance(EM_i.Mapping.Fragment1, subsume(m_{a'} \rightarrow m_b)) > 0.55$
$\ni |EM.Mapping.L| = 1, 1 < i < n$
Decompose Single Method to Improve Readability:
$\exists\, EM \ni$
$\textcircled{1}\ |m_a.b| - |EM.Mapping.Fragment2| > 0 \wedge$
$m_a.b - EM.Mapping.fragment2 \notin EM.NotMapped.T_1 \wedge$
$\textcircled{2}\ \neg\ (getter(m_b) \vee setter(m_b) \wedge$
$\textcircled{3}\ editDistance(EM.Mapping.Fragment1, subsume(m_{a'} \rightarrow m_b)) > 0.55$
$\ni |EM.Mapping.L| = 1 \wedge$
$|EM.Mapping.composite| > 0\ \vee |calls(m_{a'}.b.C.Expression, m_b)| > 0 \vee$
$|callVars(m_{a'}, m_b) \cap m_{a'}.b.C.Expression.Vars|\ >\ 0 \vee$
$(|m_{a'}.b.L| > 1\ \wedge |calls(m_{a'}.b.L.Return, m_b)| > 0)$

$subsume(m_{a'} \rightarrow m_b)$: Statement subsuming a call from method $m_{a'}$ to method m_b

$editDistance(s1, s2)$: Normalized Levenshtein Edit Distance of statements s1,s2.

$getter(m_b)$: m_b is a getter method , $setter(m_b)$: m_b is a setter method.

$calls(m_{a'}.b.C.Expression, m_b)$: A set of calls from the set of composite statement expressions in $m_{a'}$ to method m_b

$m_{a'}.b.C.Expression.Vars$: Set of all the variables in the composite statement expressions of $m_{a'}$

$callVars(m_{a'}, m_b)$: Variables in method $m_{a'}$ that are initialized with an expression that has a call to m_b

Generic Motivation Detection Rules

The rule to detect the *Decompose Method to Improve Readability* motivation is shown in Table 5.

- Multiple EXTRACT METHOD Decompositions: We primarily check the decomposition from one source method to multiple extracted methods. The detection of the extraction of multiple different pieces of code from a source method depends on analyzing together the groups of EXTRACT METHOD refactorings having the same source method in the same commit (e.g., Closure[6]).

- Single EXTRACT METHOD decompositions: The invocation to the extracted method is

    1. in return statements,
    2. in composite statement expressions of the source operation after extraction
    3. in the initializer of a variable declaration, which is referenced in composite statement expressions of the source operation after extraction
    4. the extracted piece of code has nested composite statements, and therefore its extraction can improve the readability and understandability of the Source Operation Before Extraction.

Exceptional cases

1. If there is only one statement in the extracted method, we compute the normalized Levenshtein (1966) distance between the statement calling the extracted method in the SOAE and the mapped statement in the source operation before extraction (with a threshold greater than 0.55) to determine whether the decomposition is performed

[6]https://github.com/google/closure-compiler/commit/ea96643

for the purpose of readability. If the distance shows that the text similarity between the statements before and after extraction is not much different (less than threshold) we do not consider these cases as *Decompose to Improve Readability* (e.g., jackson-databind[7]).

2. For every EXTRACT METHOD in the commit we check if the extracted method is either a getter or a setter method, and if so, the EXTRACT METHOD refactoring is not considered to have the *Decompose to Improve Readability Readability* motivation.

3. For single method decompositions we should not have all statements in the source operation mapped to the extracted method statements.

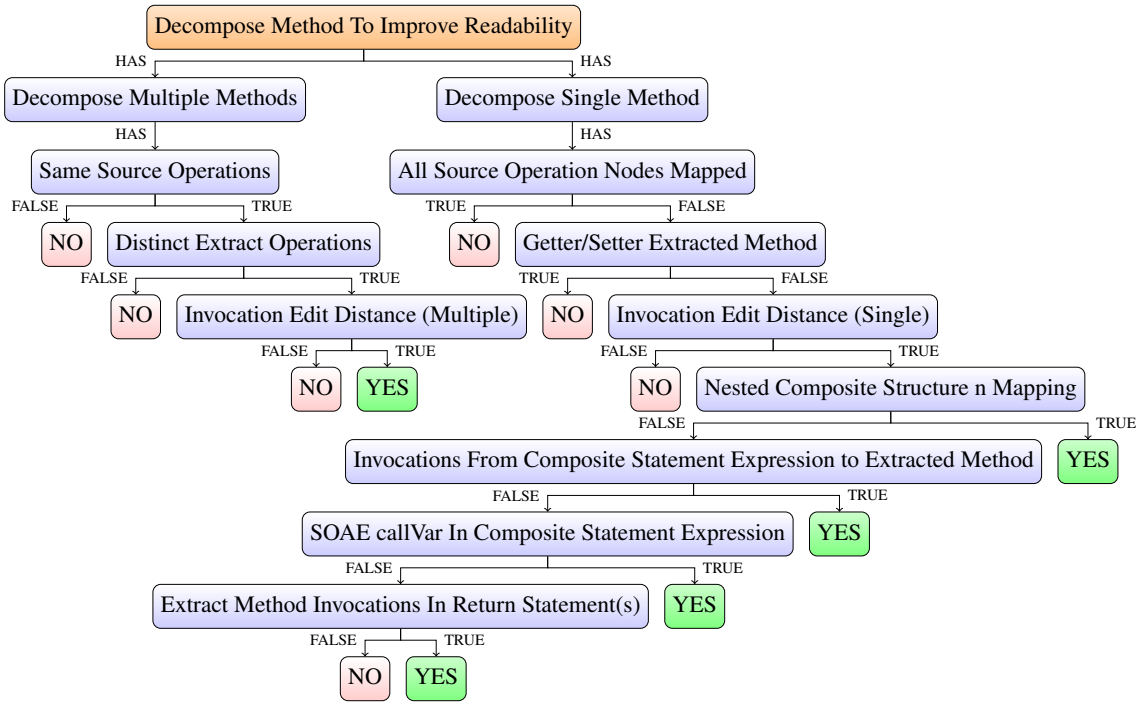

Figure 4: Decompose Method To Improve Readability Decision Tree

The decision tree to detect *Decompose Method to Improve Readability* motivation is demonstrated in Figure 4. Tree nodes show all the circumstances and criteria to detect multiple or single decompositions for the *Improve Readability* motivation.

[7]https://github.com/FasterXML/jackson-databind/commit/cfe88fe3

Filtering the set of refactoring motivations

No filtering is applied for this motivation.

3.3.4 Facilitate Extension

Refactoring is a complex process that is performed for various purposes like facilitating feature additions or supporting bug fixes (Ferreira, 2018). EXTRACT METHOD refactorings can facilitate the implementation of features or bug fixes by adding extra code in the extracted operation or in the original method. The added statements will be verified and filtered to ensure they are used to facilitate extension.

Table 6: Facilitate Extension Detection Rule

$\exists\ EM\ \ni\ \lvert EM.NotMapped.T_2\rvert > 0 \wedge$
① $\lvert calls(m_b.T_2, m_{b_i})\rvert = 0, i >= 1 \wedge$
$\{m_b.T_2 \cap EM.NotMapped.T_1\} = 0\ \wedge\ \{m_b.T_2.L \cap EM.Mapping.Fragment2\} = 0 \wedge$
$\{invocationExpression(EM.NotMapped.T2.Child) \cap m_b.P\} = 0 \wedge$
$\lvert m_b.T_2.Neutral\rvert = 0 \wedge$
② $\lvert calls(m_{a'}.T_2, m_{b_i})\rvert = 0, i >= 1 \wedge$
$\{m_{a'}.T_2 \cap EM.NotMapped.T_1\} = 0 \wedge$
$\{invocationExpression(EM.NotMapped.T2.Parent) \cap m_b.P\} = 0 \wedge$
$\lvert m_{a'}.T_2.Neutral\rvert = 0\ \wedge\ \{declaredVariables(m_{a'}) \cap m_b.P\} = 0 \wedge$
$\lvert\{m_{a'}.T_2 \cap callScope(m_b)\}\rvert = \lvert m_{a'}.T_2\rvert$
$\vee\ \lvert ternary(EM.Mapping.Fragment2)\rvert > 0$

$invocationExpression(EM.NotMapped.T_2.Child)$:Set of invocation expression in the added statements of method m_b
$invocationExpression(EM.NotMapped.T_2.Parent)$:Set of invocation expression in the added statements of method $m_{a'}$
$m_b.T_2.Neutral$:Set of added statements in method m_b that are neutral (i.e. BLOCK,RETURN_STATEMENT or other code element types like IF_STATEMENT , EXPRESSION_STATEMENT, VARIABLE_DECLARATION_STATEMENT if there is no invocation to m_b and other methods in them)
$declaredVariables(m_{a'})$: Set of all the declared variables in the method $m_{a'}$
$callScope(m_b)$: Set of all the statements that are located in the body of the innermost composite statement that encompasses the invocation to m_b
$ternary(EM.Mapping.Fragment2)$: Set of added ternary statements in the mapped code statements after extraction

Generic Motivation Detection Rules

The rule to detect the *Facilitate Extension* motivation is shown in Table 6.

- Added statements (leaf or composite) exist in the extracted operation
- Or Added statements (leaf or composite) exist in the Source Operation After Extraction (SOAE)

Exceptional cases

To handle specific scenarios we examine the following exceptional cases to find *valid* added nodes (i.e., leaf or composite statements) in the extracted method or SOAE:

1. There should be no invocation to the Extracted Method in the added nodes. In cases that there is an invocation to the extracted method the added node is excluded.
2. Added nodes should not be *neutral* statements (i.e., block, `return` statement, `if` statement, expression statement, or variable declaration statement, if there is no invocation to the extracted method in them)
3. If all or part of an added node is in the deleted statements of the source operation before extraction we do not consider it as an added node.
4. The added nodes that contain invocation expressions, or declare variables that are used in the parameters of the extract method invocation are excluded.
5. In cases where a new ternary operator is utilized in a statement mapping between the source operation and the extracted method, then this statement will be considered as an added node (e.g., intellij-community [8]).
6. Added nodes (leaf/composite) in Source Operation After Extraction (SOAE) should be located in the EXTRACT METHOD call scope. If added nodes in the SOAE are off the scope of extracted method call site, we do not consider them as added code for facilitating extension. We determine the valid scope for added statements in the

[8] https://github.com/JetBrains/intellij-community/commit/a973419

SOAE to be the inner-most composite statement that includes the extracted method invocation.

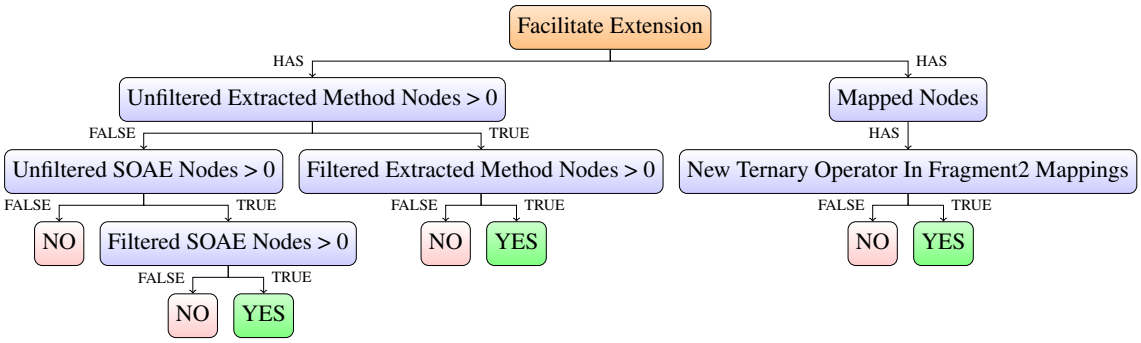

Figure 5: Facilitate Extension Decision Tree

The decision tree to detect EXTRACT METHOD with the *Facilitate Extension* motivation is demonstrated in Figure 5. Validity rules are used to filter the added statements in the Extracted Method (EM) and Source Operation after Extraction (SOAE) and to ensure they are related to facilitating extension.

Filtering the set of refactoring motivations

This motivation is removed if *Alternative Method Signature*, *Replace Method for Backward Compatibility*, *Improve Testability*, *Enable Overriding* or *Introduce Factory Method* motivations are detected for the same refactoring instance.

Furthermore, to determine the validity of the *Facilitate Extension* motivation, we perform post-processing to examine if this refactoring instance is included in any of the extracted operations removing duplication. In that case *Facilitate Extension* will be eliminated (e.g., j2objc[9]).

[9]https://github.com/google/j2objc/commit/fa3e6fa

3.3.5 Remove Duplication

Code duplication can occur to reuse some existing functionality and can potentially be destructive for the evolution and maintainability of the software (Kaur and Mittal, 2017). Various refactoring patterns exist for removing clones and EXTRACT METHOD refactorings can be applied on duplicate code (Chen et al., 2018). Duplication can be removed from a single method or from multiple methods. The duplicated piece of code will be extracted into a new method and there will be an invocation to the extracted method replacing each code duplicate.

Table 7: Remove Duplication Detection Rule

Single Method Remove Duplication: $\exists\ EM_1, .., EM_n \ni n > 1\ \wedge\ m_{b_1} = .. = m_{b_n} = m_b \wedge$ $EM_1 = .. = EM_n = EM\ \wedge\ (\|EM.Mapping\| > 1, n = 2)$
Multiple Method Remove Duplication: $\exists\ EM_1, .., EM_n \ni n > 1 \wedge m_{a_1} \neq .. \neq m_{a_n}\ \wedge\ m_{b_1} = .. = m_{b_n} = m_b \wedge$ $EM_1 \neq .. \neq EM_n \wedge (\|EM_1.Mapping\| = \|EM_2.Mapping\| = s\ >\ 1, n = 2)$

Generic Motivation Detection Rules

The rule to detect the *Remove Duplication* motivation is shown in Table 7.

- Single Method Duplication Removal: If a single EXTRACT METHOD is used several times to extract a duplicated piece of code from different parts of the same source operation. In this case we will have repetitive EXTRACT METHOD refactorings in the same commit.

- Multiple Method Duplication Removal: In this case, there are more than one EXTRACT METHOD refactoring instances, which have different source operations, but all of them have a common extracted method.

Exceptional cases

For both, single method and multiple method cases of *Duplication Removal*, we have set a threshold for the number of mapped statements in the extracted method to be more than one, when there are only two EXTRACT METHOD refactoring instances that are used to remove duplication.

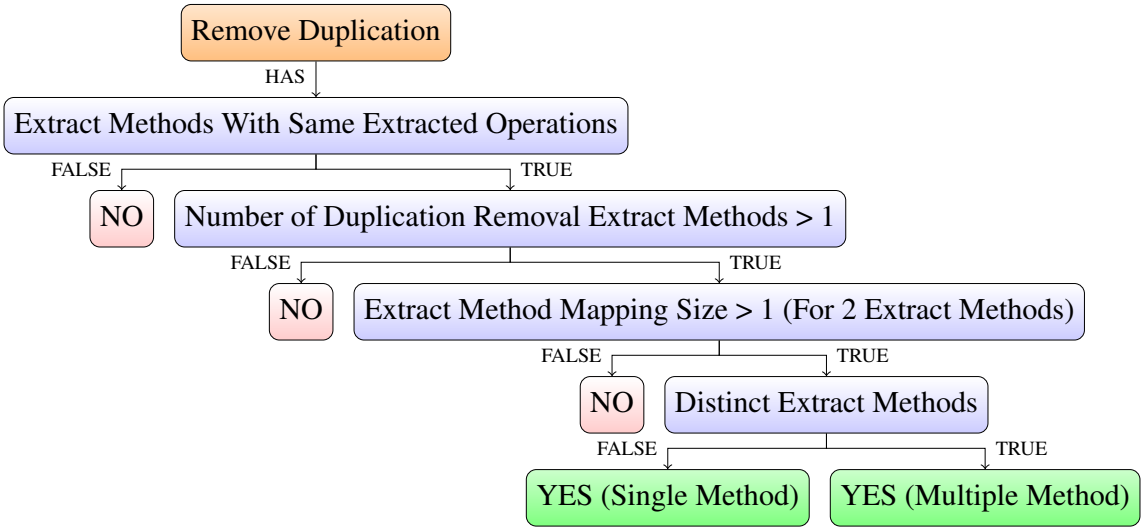

Figure 6: Remove Duplication Decision Tree

The decision tree in Figure 6 shows how single method and multiple method *Duplication Removal* is detected.

Filtering the set of refactoring motivations

This motivation has priority over *Decompose Method* motivation detected for the same EXTRACT METHOD refactoring instance.

3.3.6 Replace method Preserving Backward Compatibility

Software library maintainers use backwards compatibility to create a replacement for deprecated methods, which provides the clients of the library an option between the backwards

compatible version and the new version (Perkins, 2005). EXTRACT METHOD can be a useful refactoring tool for library maintainers to introduce new changes to previous methods. The new method can have a better name or remove some unused parameters. The original method is marked as deprecated and delegates to the extracted method.

Table 8: Replace Method Preserving Backwards Compatibility Detection Rule

$(m_{a'}.P \neq m_b.P \vee m_{a'}.n \neq m_b.n) \wedge$ $\lvert calls(\{m_{a'}\}, m_b)\rvert = 1 \wedge$ $accessModifier(m_{a'}) \neq protected \ \wedge \ accessModifier(m_{a'}) \neq private \wedge$ $\nexists\, leaf \in m_{a'}.b.L \mid \{calls(m_{a'}.leaf) \cap calls(\{m_{a'}\}, m_b)\} = 0 \wedge$ $\neg codeElementType(m_{a'}.leaf) = VARIABLE\_DECLARATION\_STATEMENT \vee$ $annotation(m_{a'}) = @Deprecated$

$accessModifier(m_{a'})$ Access modifier of the $m_{a'}$
$annotation(m_{a'})$ JavaDoc annotation of the method $m_{a'}$

Generic Motivation Detection Rules

The rule to detect the *Replace method Preserving Backward Compatibility* motivation is shown in Table 8.

- The Extracted Operation and Source Operation After Extraction (SOAE) parameters are not equal. Therefore, either the number of the parameters or their types change in the extracted method. OR Extracted Operation and SOAE have different names.
- AND SOAE should be a delegate and hand over the responsibility to the extracted method. Hence, we look for the invocations to the extracted method in the source operation after extraction. The number of the invocations to the extracted method should be equal to one to consider it as delegation.
- AND the access modifier of the source operation after extraction should not be *protected* or *private* to provide access to the backward compatible version of the method.
- AND SOAE can include temporary variables. The source operation after extraction may include statements which declare temporary variables. Therefore, all the leaf

nodes that do not include invocations to the extracted method within the SOAE will be examined to see if they are variable declaration statements (VDS).

- OR SOAE uses the `@deprecated` annotation. When the `@deprecated` annotation is not available the JavaDoc is also checked to find usages of the keyword "deprecated".

Exceptional cases

In some cases the developer forgets to use the `@deprecated` annotation for the SOAE. The detection rule enables us to detect backward compatibility even when `@deprecated` annotation is not provided.

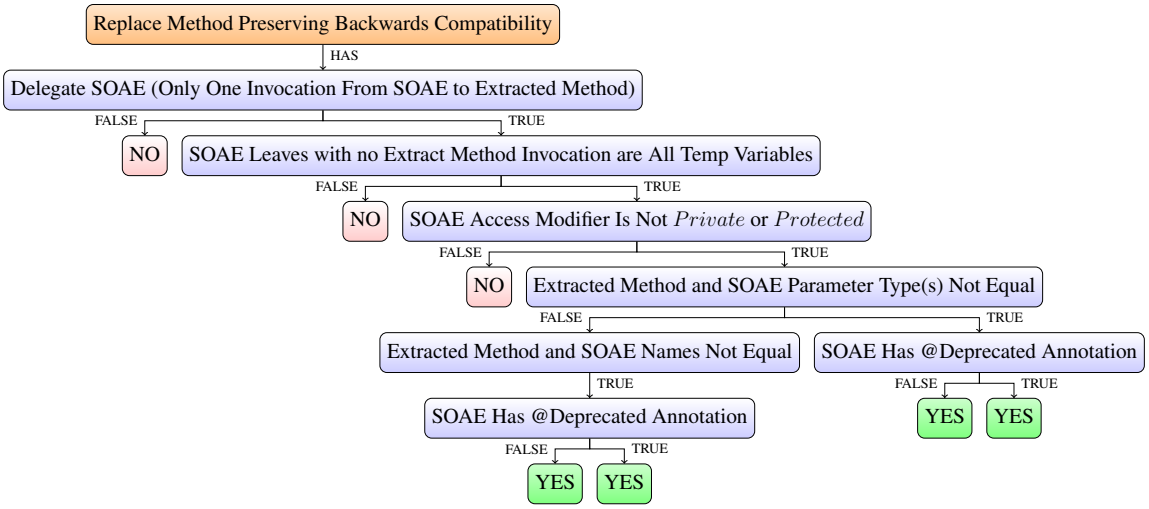

Figure 7: Replace Method Preserving Backwards Compatibility Decision Tree

The decision tree to detect EXTRACT METHOD refactorings that are performed to Replace Method for Backward Compatibility is shown in Figure 7 in which the logic to detect this EXTRACT METHOD motivation is demonstrated.

Filtering the set of refactoring motivations

- Precedence of Remove Duplication.

- Removing *Introduce alternative Method*, *Improve Testability* and *Facilitate Extension* if they are detected for the same refactoring instance.

3.3.7 Improve Testability

To achieve the highest confidence in software quality, it is essential to utilize techniques like refactoring to improve the design for testability (Tarlinder, 2016). EXTRACT METHOD refactorings can facilitate the unit testing of a method by isolating a piece of code for testability purposes in a separate method.

Table 9: Improve Testability Detection Rule

$\exists\, m_t \rightarrow m_b \mid m_t \in \{C^+ \cup C^{\sim}\} \wedge$
① $m_t \in T_{s_{i+1}} \wedge$
② $\{m_t \rightarrow m_b \cap calls(m_b.C, m_b)\} = 0 \wedge$
③ $\{calls(\{m_t\}, m_b) \cap \{calls(M^+, m_b) \cup calls(m_t.T2, m_b)\}\} > 0$

Generic Motivation Detection Rules

The rule to detect the *Improve Testability* motivation is shown in Table 9.

- All changed classes and newly added classes are examined to find test methods. We check the annotations of test methods and also their signature to find if they are test methods in a test class (i.e., include "test" in their name).
- And there is an invocation from a test method to the extracted method.

Exceptional cases

1. The invocation should be located in a different class than the extracted method class. If the test method is inside the extracted method class, it has already been part of the test code and therefore the extract method is not used for testability purposes.

2. And the invocation should be in the added or edited nodes. Otherwise we do not consider that invocation valid for improving testability (e.g., VoltDB[10]).

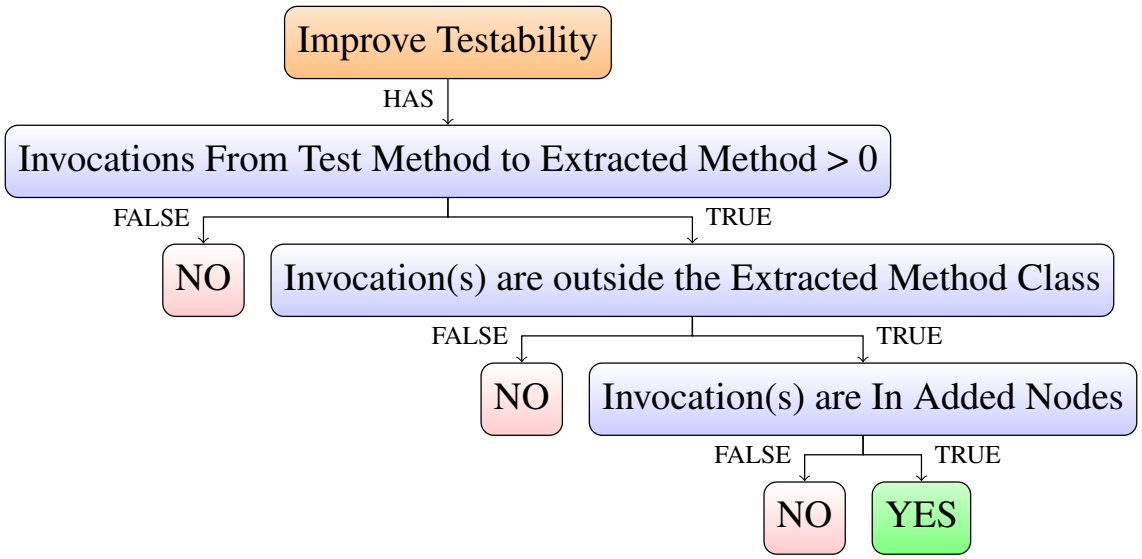

Figure 8: Improve Testability Decision Tree

Figure 8 shows the decision tree that is used to detect the EXTRACT METHOD refactorings that are used to improve testability.

Filtering the set of refactoring motivations

- Precedence of Remove Duplication
- Removing *Decompose to Improve Readability* if they are detected for the same refactoring instance.
- Removing *Facilitate Extension* if they are detected for the same refactoring instance.

3.3.8 Enable Overriding

EXTRACT METHOD refactorings can be utilized to edit a stable piece of code for the purpose of preparing it to be overridden in a subclass. The extracted method will be overridden

[10]https://github.com/VoltDB/voltdb/commit/e58c9c3

in a separate method with a more specialized behaviour when it is intended to enable overriding. Sometimes developers do not override the extracted method, but simply extract the method for future changes related to overriding the method.

Table 10: Enable Overriding Rule

$\exists\, m_v \rightarrow m_b \mid m_v \in \{C^+ \cup C^\sim\} \ \wedge subType(m_V.C, m_b.C) \ \wedge \ m_v.n = m_b.n$ $\vee \, \lvert\{overridingRelatedKeywords() \cap commentTokens(m_b)\}\rvert \ > \ 0$

$subType(m_V.C, m_b.C)$: returns true if class of the virtual method $m_v.C$ is a subtype of the class of the extracted method $m_b.C$
$overridingRelatedKeywords()$: Returns set of all overriding-related keywords like virtual, override, etc.
$commentTokens(m_b)$: Set of all words in the extracted method comment.

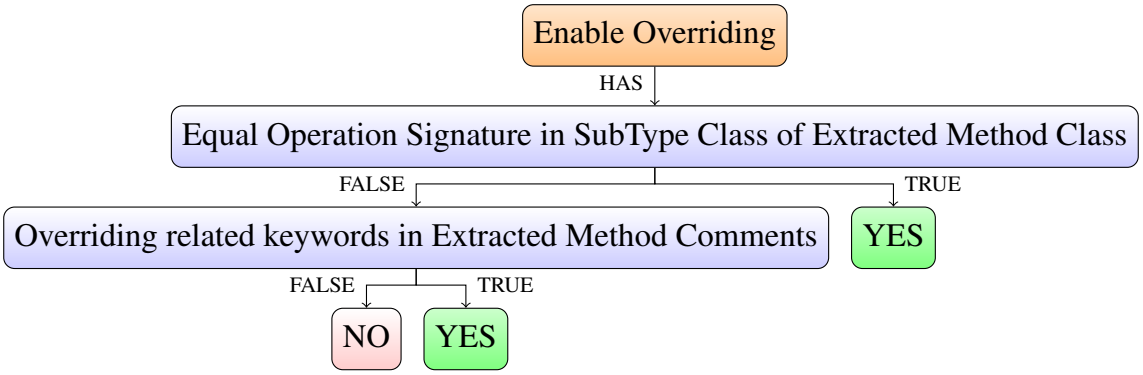

Figure 9: Enable Overriding Decision Tree

Generic Motivation Detection Rules

The rule to detect the *Enable Overriding* motivation is shown in Table 9.

- All the changed classes and newly added classes will be examined.
- A method with equal signature to the extracted method is found in a Subtype class of the extracted method class. This method implements the specialized behaviour for the extracted method.

Exceptional cases

We further check the following scenarios to improve the precision of the detected instances:

1. The extracted method behaviour is sometimes specialized in an anonymous class. Therefore we also check anonymous classes that override the extracted method.

2. And we further check the comments of the extracted method to find certain keywords in the Javadoc (e.g., virtual, override, etc.) that shows the intention of the developer to override the extracted method behaviour in the future.

Figure 9 depicts the decision tree to detect EXTRACT METHOD refactorings that are used to enable overriding. The extracted method might not be overridden in the current commit but it may have keywords to show that it is intended to be utilized for such a purpose in the future.

Filtering the set of refactoring motivations

- Precedence of Remove Duplication
- Removing *Reusable Method*, *Facilitate Extension*, *Introduce Factory Method* and *Remove Decompose to Improve Readability* if they are detected for the same refactoring instance.

3.3.9 Enable Recursion

Recursive structures make it easier for developers to understand, remember and implement algorithms that are more elegantly expressed (Kourie and Watson, 2012). EXTRACT METHOD refactorings can be intended to extract a piece of code to implement a recursive method.

Table 11: Enable Recursion Detection Rule

$\exists\, m_a, m_b \ni \lvert calls(\{m_a\}, m_a)\rvert = 0 \ \wedge \lvert calls(m_b, m_b)\rvert > 0 \wedge$ $\lvert\{invocationExpression(m_b.b) \cap \{this, null\}\}\rvert \ > \ 0$

$invocationExpression(m_b.b)$:Set of invocation expressions in all of the m_b statements.

Generic Motivation Detection Rules

The rule to detect the *Enable Recursion* motivation is shown in Table 11.

- Source Operation After Extraction (SOAE) should be recursive (with at least one matching invocation inside itself)

Exceptional cases

1. The Source Operation Before Extraction should not be recursive.

2. To ensure that the recursive invocation in the extracted method is calling the extracted method, we check the expression of the matched invocations in the extracted method to ensure that it is either *null* or *this*.

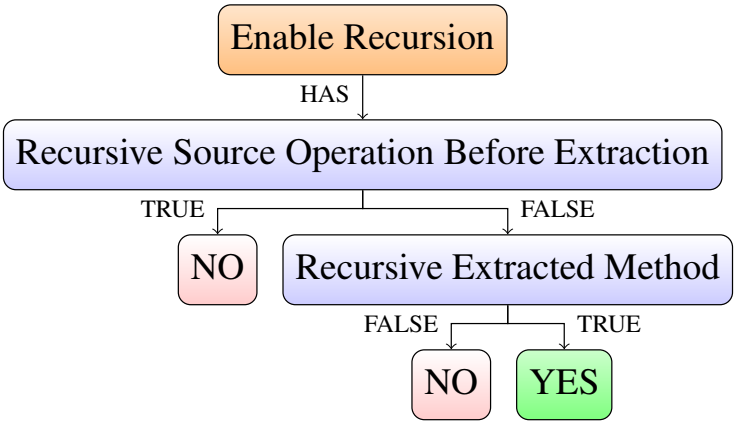

Figure 10: Enable Recursion Decision Tree

The decision tree for detecting recursive extracted methods is shown in Figure 10.

Filtering the set of refactoring motivations

No filtering is applied for the detection of this motivation.

3.3.10 Introduce Factory Method

Factory methods create objects and have a non-`void` return type corresponding ot the type of the object being created (Seng et al., 2006). EXTRACT METHOD refactorings can be utilized to create factory methods by extracting the constructor call to a separate method.

Table 12: Introduce Factory Detection Rule

$\exists\, m_b \ni \lvert newOperator(m_b.b.L.Return)\rvert > 0 \wedge$ $\lvert\{newExpressionType(mb.b) \cap \{m_b.t \cup subType(m_b.t)\}\}\rvert > 0 \vee$ $\{objectCreationVars(m_b.b.L.Return) \cap m_b.t\} > 0 \ \wedge allFactoryRelated(Vars(m_b)) \wedge$ $\neg factoryMethod(m_a)$

$m_b.t$:Return type of the method $m_b.t$
$subType(t)$: Set of subtypes of the type t
$newOperator(S)$: Statements in S that have only one new object creation operator
$newExpressionType(S)$: Set of types of new expressions in the set of statements S
$objectCreationVars(S)$: Set of variables in the set of statements S that are initialized with object-creation expression
$Vars(m_b)$:Set of all variables in method m_b
$allFactoryRelated(V)$:Returns true if all the variables in V are used for object creation or state setting
$factoryMethod(m)$: Reutrns true if method m is a factory method.

Generic Motivation Detection Rules

The rule to detect the *Introduce Factory Method* motivation is shown in Table 12.

- There is an object created in the return statement of the extracted method using the *new* keyword.
- And the object that is created in the return statement has an equal type or is a subtype of the extracted method return type.
- OR, All variables appearing in the return statements are initialized with an object creation that is equal to the return type of the extracted method.

Exceptional cases

1. Source Operation Before Extraction (SOBE) should not be a Factory Method. If SOBE is already a factory method the extracted method will not be accepted as a

valid factory method.

2. To ensure that extra statements in the extracted method are contributing to the factory method implementation, we also further ensure that all the statements in the extracted method are related to object creation. This means they should help create the object for the first time or change its state after it is created. Otherwise, we do not consider the extracted method a valid factory method.

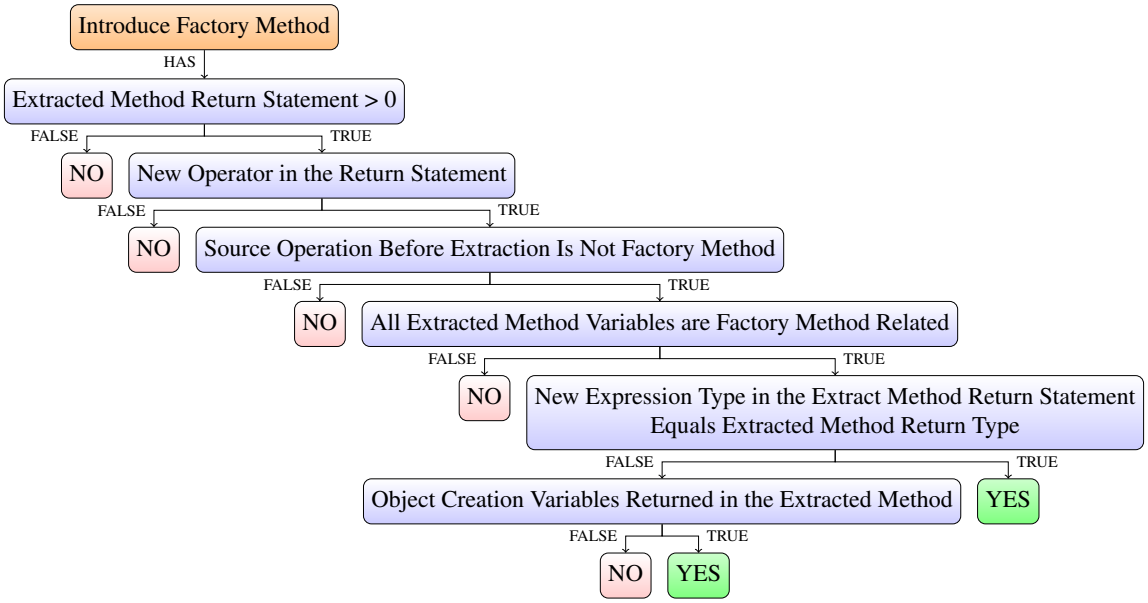

Figure 11: Introduce Factory Method Decision Tree

The decision tree in Figure 11 shows the detection logic to find the motivation of the developer when EXTRACT METHOD refactoring is used to Introduce Factory Method.

Filtering the set of refactoring motivations

- *Facilitate Extension* and *Remove Duplication* motivations are removed if they are detected for the same refactoring instance.

3.3.11 Introduce Async Operation

In concurrent programming, developers usually utilize low-level constructs to implement asynchronous operations. Vendors are interested to know the code transformations done by developers to introduce better constructs for improving performance and reducing execution time (Pinto et al., 2015). For instance, Asynchronizer, is a refactoring tool that extracts a piece of sequential code into a concurrent one (Lin et al., 2014). It is important to know how EXTRACT METHOD refactorings are used to extract a piece of code to introduce an asynchronous operation that is executable in a separate thread.

Table 13: Enable Async Detection Rule

$\exists EM \ni \lvert\{calls(m_{a'}, m_b) \cap calls(anonymousRunnableClass(m_{a'}), m_b)\}\rvert > 0$

$anonymousRunnableClass(m)$: Set of anonymous runnable classes in method m

Generic Motivation Detection Rules

The rule to detect the *Introduce Async Operation* motivation is shown in Table 13.

- There is at least one anonymous class implementing the Runnable interface in the Source Operation After Extraction (SOAE)
- And there is an invocation from within the anonymous class to the extracted method.

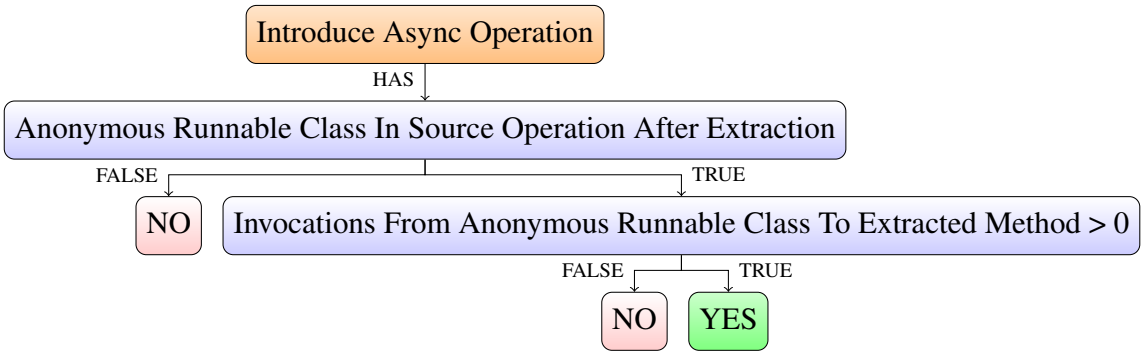

Figure 12: Introduce Async Operation Decision Tree

The detection rules are used to form the decision tree of the EXTRACT METHOD refactorings which are intended to enable an async operation and shown in Figure 12.

Filtering the set of refactoring motivations

No filtering is applied for this motivation.

3.4 Step 2: Applying the Detection Rules in Large Scale

In the second step, we use the optimized detection rules to detect the developer motivations in a large number of open source projects. We built our dataset based on the projects that were selected in the two most prominent studies on refactoring motivations (Silva et al., 2016; Pantiuchina et al., 2020). More specifically, we included all 185 projects used in the Silva et al. (2016) dataset and all 150 projects used in the Pantiuchina et al. (2020) dataset, reaching a total of 325 projects (as 10 projects were commonly used in both studies). For each one of the 325 projects, we mined their entire commit history (master branch), excluding merge commits, as the analysis of merge commits introduces duplicate refactoring instances (Tsantalis et al., 2013). This resulted in 346K EXTRACT METHOD and EXTRACT AND MOVE METHOD refactoring instances, which were detected in 132,897 commits. We used RefactoringMiner 2.0 as it is the current state-of-the-art refactoring mining tool with the highest precision and recall and fastest execution time among the currently available tools.

We have also defined certain flags that show the characteristics of code elements and changes in the commit that are related to each EXTRACT METHOD motivation. These flags can be considered as the motivation detection triggers and are logged by the tool. We can manually validate each motivation by checking the type of changes and code characteristics that confirm the corresponding detection rule. Table 14 shows the motivation flags that are used for all the EXTRACT METHOD motivations.

Table 14: Extract Method Refactoring Motivation Flags

Motivation Flags	Flag Description
1. Reusable Method Flags:	
EM_INVOCATION_IN_REMOVE_DUPLICATION	EM Invocation is in the SOAE of RemoveDuplication Refactoirng
EM_INVOCATION_IN_TEST_OPERATION	EM Invocation is in test Opeataion
SOAE_IS_TEST_OPERATION	SOAE is a Test Operation
EM_INVOCATION_EQUAL_MAPPING	Equal Mapping of invocations in SOBE AND SOAE
EM_EQUAL_SINATURE_INVOCATIONS	Invocations to an operation with an equal signature to Extracted Mewthod
EM_NESTED_INVOCATIONS	Invocation to an EM from within a Nested Extrcted Method
EM_SOAE_INVOCATIONS	Invocations to the EM from SOAE
EM_NONE_SOAE_INVOCATIONS	Invocations to the EM from Non SOAE methods
2. Introduce Alternative Method Signature Flags:	
EM_SOAE_EQUAL_PARAMETER_TYPES	EM has Euqal Paramater Types with SOAE
SOAE_IS_DELEGATE_TO_EM	EM is a delegate to the SOAE
SOAE_IS_ALL_TEMP_VARIABLES	All SOAE statements h temporary variables
EM_HAS_ADDED_PARAMETERS	Extracted Operation has more parameters than the Source Operation
3. Introduce Decompose Method to Improve Readability Flags:	
EM_DECOMPOSE_SINGLE_METHOD	Single Method Decomposition to Improve Readability
EM_DECOMPOSE_MULTIPLE_METHODS	Multiple Method Decomposition to Improve Readability
EM_WITH_SAME_SOURCE_OPERATION	EMs Have same source operation
EM_DISTINCT	EMs are Distinct
EM_INVOCATION_EDIT_DISTANCE_THRESHOLD_SM	EM Invocation Edit Distance in Single Decomposition
EM_INVOCATION_EDIT_DISTANCE_THRESHOLD_MM	EM Invocation Edit Distance in Multiple Decomposition
EM_ALL_SOURCE_OPERATION_NODES_MAPPED	All the nodes in the source operation are mapped to the Extracted Operation
EM_GETTER_SETTER	EM is a Getter/Setter Method
EM_MAPPING_COMPOSITE_NODES	Number of Composite Nodes in the Mapping
EM_COMPOSITE_EXPRESSION_INVOCATIONS	Invocations to the EM from the Composite expression
EM_COMPOSITE_EXPRESSION_CALLVAR	callVar to the EM exists in the Composite expression
EM_RETURN_STATEMENT_INVOCATIONS	Invocations to the EM from the Return Statement
4. Facilitate Extension Flags :	
EM_MAPPING_FRAGMENT2_TERNARY	Ternary operator is used in the fragment 2 of the EM Mapping
SOAE_NOT_MAPPED_T2_UNFILTERED	Unfiltered notMapped T2 nodes in the SOAE
EM_NOTMAPPED_T2_UNFILTERED	Unfiltered notMapped T2 nodes in the Extracted Operation
SOAE_NOTMAPPED_T2_FILTERED	Filtered notMapped T2 nodes in the SOAE
EM_NOTMAPPED_T2_FILTERED	Filtered notMapped T2 nodes in the Extracted Operation
EM_T2_IN_T1	NotMapped T2 Nodes exist in NotMapped T2 Nodes in the EM
SOAE_T2_IN_T1	NotMapped T2 Nodes exist in NotMapped T2 Nodes in the SOAE
EM_T2_IN_MAPPING	T2 NotMapped Nodes are in the EM(child) mapping
SOAE_T2_IN_MAPPING	T2 NotMapped Nodes are in the SOAE mappings
EM_T2_IE_IN_EM_PARAMETERS	EM T2 NotMapped Nodes Invocation Expressions are in the EM Parameters
SOAE_T2_IE_IN_EM_PARAMETERS	SOAE T2 NotMapped Nodes Invocation Expressions are in the EM Parameters
EM_T2_NEUTRAL	EM T2 NotMapped Nodes are Neutral
SOAE_T2_NEUTRAL	SOAE T2 NotMapped Nodes are Neutral
SOAE_T2_DV_IN_EM_PARAMETERS	SOAE Declared Variable are in the EM Parameters
EM_T2_EM_INVOCATIONS	Extracted Operation T2 has invocation to the EM
SOAE_T2_EM_INVOCATIONS	SOAE T2 has invocation to the EM
5. Remove Duplication Flags :	
EM_SAME_EXTRACTED_OPERATIONS	EM operations with the same extracted operations
EM_MAPPING_SIZE	EM Mapping Size
EM_NUM_METHODS_USED_IN_DUPLICATION_REMOVAL	Number of EMs used in Duplication Removal
6. Replace Method Preserving Backwards Compatibility Flags :	
EM_SOAE_EQUAL_PARAMETER_TYPES	EM has Euqal Paramater Types with SOAE
SOAE_IS_DELEGATE_TO_EM	EM is a delegate to the SOAE
EM_SOAE_EQUAL_NAMES	EM has equal names with the SOAE
SOAE_DEPRECATED	SOAE is Deprecated
SOAE_PRIVATE	SOAE is Private
SOAE_PROTECTED	SOAE is Protected
7. Improve Testability Flags :	
EM_INVOCATION_IN_TEST_OPERATION	EM Invocation is in test Opeataion
EM_TEST_INVOCATION_CLASS_EQUAL_TO_EM_CLASS	EM test invocation class is equal to the EM class
EM_TEST_INVOCATION_IN_ADDED_NODE	EM test invocation is in added nodes
8. Enable Overriding Flags :	
EM_EQUAL_OPERATION_SIGNATURE_IN_SUBTYPE	Equal Operation Signature in SubType of the EM exists
EM_OVERRIDING_KEYWORD_IN_COMMENT	EM has overriding keywords in its comment
9. Enable Recursion Flags :	
SOBE_RECURSIVE	Source Operation Before Extraction is recursive
EM_RECURSIVE	EM is recursive
10. Introduce Factory Method Flags :	
EM_HAS_RETURN_STATEMENTS	EM has return statements
EM_RETURN_STATEMENT_NEW_KEYWORDS	new keywords exist in the return statement.
EM_RETURN_EQUAL_NEW_RETURN	EM return type equals the object creation type in the return statement
EM_OBJECT_CREATION_VARIABLE_RETURNED	Variabale initialized with Object Creation is returned in the EM
EM_VARS_FACTORY_METHOD_RELATED	EM Variables are related to object creation for factory method
SOBE_FACTORY_METHOD	SOBE is Factory Method
11. Introduce Async Operation Flags :	
SOAE_STATEMENTS_CONTAIN_RUNNABLE_TYPE	SOAE statements have Runnable Type
SOAE_ANONYMOUS_CLASS_RUNNABLE_EM_INVOCATION	SOAE has an anonymous class and runnable type that has invocation to EM

EM : Extracted Method
$SOAE$: Source Operation After Extraction
$SOBE$: Source Operation Before Extraction

Chapter 4

Experiment Results

In this chapter we present the results of our large scale study on EXTRACT METHOD refactoring motivations. We conducted an experiment on 346K instances of EXTRACT METHOD and EXTRACT AND MOVE METHOD refactoring operations in 132,897 commits of 325 open-source repositories hosted in GitHub.

4.1 RQ1: Accuracy of Automatic Motivation Extractor

To evaluate the accuracy of our motivation detection rules, we used two separate oracles of developer motivations based on previous studies (Silva et al., 2016; Pantiuchina et al., 2020) as our training and testing datasets, respectively.

4.1.1 Accuracy on the Training Dataset

The oracle provided by Silva et al. (2016) is very convenient for the purpose of training and evaluating our motivation detection rules, as the refactoring motivations are provided at commit-level based on the responses of the actual developers who performed the refactorings. We filtered all commits that include at least one EXTRACT METHOD refactoring

instance, and then mapped the motivations related to EXTRACT METHOD (i.e., the 11 motivation themes shown in Table 1) to the actual refactoring instances. This process resulted in a total of 261 EXTRACT METHOD refactoring instances assigned with 320 motivation labels, as some instances are involved in multiple motivations.

After following the process explained in Section 3.2 to optimize our motivation detection rules, we reached the precision and recall shown in Table 15. The overall precision and recall of our tool on the training dataset is 97.2% and 95.9%, respectively.

Table 15: Extract Method Motivations Detection Precision and Recall

Motivation Type	TP	FP	FN	Precision	Recall
1. Reusable Method	72	2	4	0.973	0.947
2. Alternative Method Signature	37	3	3	0.925	0.925
3. Improve Readability	54	0	3	1.000	0.947
4. Facilitate Extension	50	0	0	1.000	1.000
5. Remove Duplication	39	3	1	0.929	0.975
6. Backwards Compatibility	10	1	0	0.909	1.000
7. Improve Testability	8	0	1	1.000	0.889
8. Enable Overriding	3	0	1	1.000	0.750
9. Enable Recursion	2	0	0	1.000	1.000
10. Introduce Factory Method	31	0	0	1.000	1.000
11. Introduce Async Method	1	0	0	1.000	1.000
Total	307	9	13	0.972	0.959

The high precision and recall achieved on the training dataset might be the result of overfitting (i.e., the detection rules overfit the characteristics of the refactoring instances in the training dataset). Therefore, to ensure that our tool has no overfitting problems, we compute its accuracy on an another refactoring motivation oracle with refactoring instances from different projects.

4.1.2 Accuracy on the Test Dataset

The motivations of refactoring activities were also manually analyzed and tagged at pull request level in a study by Pantiuchina et al. (2020). We used the oracle available in this

study as our test dataset to further validate the accuracy of our motivation detection tool.

We filtered the pull requests from the test oracle, which contained only EXTRACT METHOD or EXTRACT AND MOVE METHOD refactoring instances. Since the motivation labels are assigned at the pull request level, it would be very risky to map the motivation labels to the actual refactoring instances within commits in pull request containing multiple different refactoring types in addition to the EXTRACT METHOD refactoring type. Therefore, we ended up with 56 pull requests in total and assigned the motivation labels to the individual refactoring instances detected in the commits of each pull request. Moreover, because Pantiuchina et al. (2020) use in some cases different names for the motivation themes, we did a mapping between the motivation themes used in the studies by Pantiuchina et al. (2020) and Silva et al. (2016), as shown in Table 16.

We automatically detected 170 motivations using our tool in these 56 pull requests. 61 motivations were matched according to the mappings shown in Table 16. For the remaining 109 motivations that were not matched (i.e., potential false positives), we manually analyzed the commits to ensure that the automatically detected motivation is present in the commits of each pull request.

Motivation Extractor provides extra information about the detection of a certain motivation. For instance, in the case of *Reusable Method* motivation, we have the fully qualified name of the operation(s) where the extracted method is reused. Furthermore, the detection rule contains numerous flags that are logged during the motivation detection process. These flags that are shown in Table 14 indicate how code elements in the refactoring context are related to the detected motivation. We used these flags along with the information retrieved from the Motivation Extractor to accurately trace each motivation in the pull request commits and validate the 109 detected motivations that were not matched with the assigned motivation labels by Pantiuchina et al. (2020).

After manual analysis we found that only 3 out of 109 unmatched motivations were

Table 16: Pull Request Motivation Mapping to Extract Method Motivations

Extract Method Motivations	**Pull Request Motivations**
Resuable Method	Adhere to DRY principle Foster code reuse
Introduce Alternative Method Signature Replace Method Preserving Backwards Compatibility	To adhere to naming convention To keep consistency in naming To simplify API usage To use more specific names To promote API compatibility
Decompose Method to Improve Readability	Better distribute responsibilities Improve Understandability & Readability Refactoring confusing code
Facilitate Extension	For implementing a new feature Improve error messages and logging Improve exception handling Improve extensibility To improve performance
Remove Duplication	Remove Clones
Improve Testability	Improve organization of test directory Improve quality of test code To ensure a better mapping between test & production code To simplify testing activities
Enable Overriding	Facilitate subclassing
Enable Recursion	None
Introduce Factory Method	None
Introduce Async Operation	None
None	Cleanup code Fix warnings from static analysis tools Other motivations Improve modularization Remove unnecessary code Unclear Improve Maintainability To expand abbreviations

actually false positives. The remaining 106 unmatched motivations were actually true positives. We suspect that these motivations were missed by Pantiuchina et al. (2020), because their analysis was done at pull request level, and thus some refactoring instances might be missed from the analysis or considered less significant than others at the pull request level.

We also further analyzed the motivations that were tagged in the test oracle at pull request level, but were not detected by our tool (i.e., potential false negatives). For the examined 56 pull requests we had 103 manually tagged motivations among which 21 motivations were matched according to the mappings shown in Table 16. We divided the remaining 82 unmatched motivations into five categories as following:

1. Super-Motivation: Motivations in the test oracle that were more general than the automatically detected motivations.

2. Sub-Motivation: Motivations in the test oracle that were more specific than the automatically detected motivations.

3. Non EXTRACT METHOD Motivation: Motivations in the test oracle that were not related to the EXTRACT METHOD refactoring instances found in the pull request commits.

4. Filtered-out Motivation: Motivations in the test oracle that were filtered out in the automatic motivation detection process due to the fact that another motivation with higher priority was detected.

5. False Negative (FN): Motivations in the test oracle that were not detected by our automatic motivation detection tool. This can be attributed to missed refactoring instances by RefactoringMiner. For instance, if RefactoringMiner detects only one of the instances involved in the removal of duplicated code, then our detection rules cannot infer the *Remove Clone* motivation.

Table 17 shows the number of motivations from the Pantiuchina et al. (2020) study that were not detected by our detection rules categorized in the aforementioned categories. To compute the overall precision and recall of the tool we consider the super-motivations and sub-motivations as true positives since it is basically the different approach the two studies followed coming up with more general/specific themes. Filtered motivations are considered as false negatives since our tool does not report them at the end and Non EXTRACT METHOD motivations are removed from the oracle. We consider as false negatives the instances in the FN column.

Table 18 shows the accuracy of our motivation detection rules on the testing dataset. The **Auto** column refer to FNs that resulted by analyzing the motivations detected by our

Table 17: Pull Request Motivations

			Unmached Motivations				
Pull Request Motivations	**All**	**Matched(TP)**	**Super**	**Sub**	**Non-EM Motivation**	**Filtered**	**FN**
1- Adhere to DRY principle	1	1	0	0	0	0	0
2- Foster code reuse	12	4	0	0	4	3	1
3- To adhere to naming convention	1	0	0	0	1	0	0
4- To keep consistency in naming	1	0	0	0	0	0	1
5- To simplify API usage	1	0	0	1	0	0	0
6- To use more specific names	1	0	0	0	1	0	0
7- To promote API compatibility	1	0	0	0	1	0	0
8- Better distribute responsibilities	6	2	0	0	2	0	2
9- Improve Understandability & Readability	15	3	6	0	5	1	0
10- Refactoring confusing code	4	1	0	1	2	0	0
11-To better reflect code responsibility	1	1	0	0	0	0	0
12- For implementing a new feature	6	0	0	3	3	0	0
13- Improve exception handling	3	1	0	1	0	0	1
14- Improve extensibility	3	1	0	0	1	0	1
15- To improve performance	3	0	0	0	3	0	0
16- Remove Clones	7	4	0	0	1	0	2
17- Improve organization of test directory	1	0	0	0	1	0	0
18- Improve quality of test code	1	1	0	0	0	0	0
19- To ensure a better mapping between test & production code	1	0	1	0	0	0	0
20- To simplify testing activities	3	1	1	0	1	0	0
21- Facilitate subclassing	1	1	0	0	0	0	0
22- Cleanup code	9	0	0	2	7	0	0
23- Fix warnings from static analysis tools	1	0	0	0	1	0	0
24- Other motivations	4	0	0	0	4	0	0
25- Improve modularization	2	0	0	2	0	0	0
26- Remove unnecessary code	2	0	0	1	1	0	0
27- Unclear	9	0	0	0	9	0	0
28- Improve Maintainability	2	0	0	0	2	0	0
29- To expand abbreviations	1	0	0	0	1	0	0
Total	103	21	8	12	50	4	8

tool. The **Manual** columns refer to TPs and FNs, respectively, that resulted by analyzing the motivations manually labelled in the test oracle. The **Validated** column refers to TPs that resulted by analyzing the motivations detected by our tool, for which no matching motivation label was found in the test oracle.

Table 18: Precision and Recall of Extract Method Motivations in Pull Requests

	TP			FN						
Motivation Type	**Matched**	**Validated**	**Manual**	**Auto**	**Filtered**	**Manual**	**FP**	**TN**	**Precision**	**Recall**
1. Reusable Method	5	20	0	0	3	1	0	4	1.000	0.862
2. Alternative Method Signature	0	16	2	1	0	1	3	2	0.857	0.900
3. Improve Readability	14	6	7	0	1	1	0	9	1.000	0.931
4. Facilitate Extension	12	24	4	0	0	3	0	7	1.000	0.930
5. Remove Duplication	27	31	0	0	0	2	0	1	1.000	0.967
6. Backwards Compatibility	0	3	0	0	0	0	0	0	1.000	1.000
7. Improve Testability	2	3	2	0	0	0	0	2	1.000	1.000
8. Enable Overriding	1	0	0	0	0	0	0	0	1.000	1.000
9. Enable Recursion	0	1	0	0	0	0	0	0	1.000	1.000
10. Introduce Factory Method	0	1	0	0	0	0	0	0	1.000	1.000
11. Introduce Async Method	0	1	0	0	0	0	0	0	1.000	1.000
12. None	0	0	5	0	0	0	0	25	-	-
Total	61	106	20	1	4	8	3	50	0.984	0.935

There are 21 manually tagged motivations at pull request level that were matched with 61 automatically detected motivations at refactoring instance level. Overall, we identified 187 refactoring-level true positives including 61 matched, 106 Validated, 20 Manual (12 sub-motivations and 8 super-motivations), and 13 false negatives (4 Filtered-out, 1 Auto, 8 Manual). Finally, we eliminated 50 tagged motivations as true negatives. These motivations were not specifically related to EXTRACT METHOD refactoring instances in the pull requests. These tags were either related to the pull request comments and discussions, or code changes not involving Extract Method refactorings. In some cases the motivation was related to EXTRACT METHOD refactorings, but the manual analysis of the changes in the pull request resulted in an incorrect motivation. For instance, instances related to *Remove Duplication* or *Improve Testability* motivations were mistakenly classified as *Reusable Method*, because the analysis of the extracted method call sites was not properly done. We computed an overall precision of 98.4% and recall of 93.5% on the testing dataset, which are very close to the precision and recall achieved on the training dataset (97.2% and 95.9%, respectively).

RQ1 Conclusion: In both training and testing datasets, our motivation detection rules achieved a precision over 97% and a recall over 93%. Therefore, our *Motivation Extractor* tool is reliable to conduct a large-scale study on the reasons driving refactoring with automatically computed motivations based on our rules.

4.2 RQ2: Most Prevalent Motivations for Extract Method Refactoring Operations

In Figure 13, we present the ranking of the motivations we automatically extracted by analyzing 346K instances of EXTRACT METHOD and EXTRACT AND MOVE METHOD refactoring operations in 132,897 commits of 325 open-source repositories hosted in GitHub. Moreover, we show side-by-side the ranking of the motivations obtained from the survey by Silva et al. (2016) based on developer answers. With red arrows we highlight some notable differences or similarities in the two rankings.

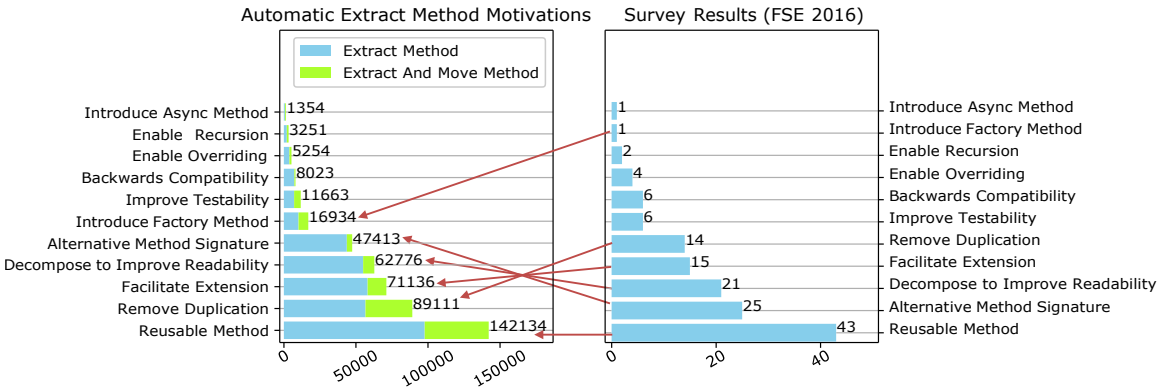

Figure 13: Comparison of our automatically extracted motivation ranking with Silva et al. (2016) survey ranking

Reusable Method is the top-most frequent EXTRACT METHOD refactoring motivation in both studies. About 41% of the examined refactoring instances are intended for reusing the extracted code within the same commit. Unfortunately, there is a huge research gap in recommendation systems for extracting reusable methods and components. Therefore,

researchers that are building refactoring recommendation tools should focus on developing techniques that find opportunities to extract reusable methods. The characteristics of the reusable extracted methods that we found in our dataset using our motivation detection rules can be a useful resource to better understand the factors driving this particular motivation.

In a recent study, AlOmar et al. (2020) found that extracting reusable functionality is increasing the number of methods in a class. Therefore, developers can pull up reusable methods to a superclass to share them with all subclasses, or relocate and move them in classes outside their inheritance hierarchy. At the same time, the visibility of the reusable methods can be changed to allow other objects and methods to access their functionality.

We found that 31% of the reusable extracted methods are a result of EXTRACT AND MOVE METHOD refactorings and 69% of them are a result of EXTRACT METHOD refactorings. We also analyzed the access modifiers of the extracted methods and compared them with the access modifiers of the methods from which they were extracted. We ordered access modifiers from higher to lower visibility levels (*public* > *protected* > *package* > *private*) and recorded whether the visibility of the extracted methods increased, decreased, or remained the same compared to the visibility of the original methods from which they were extracted.

Figure 14 shows the percentages of resuable extracted methods with increased, decreased and the same visibility for EXTRACT METHOD and EXTRACT AND MOVE METHOD refactorings, respectively. About 58% of the extracted methods in EXTRACT METHOD refactorings have a decreased visibility and about 4% have an increased visibility. Moreover, in 74% of the EXTRACT METHOD refactorings with decreased visibility, the extracted methods are *private*, while the methods from which they were extracted are *public*. This shows that locally extracted methods are more intended for *local reuse* within the class in which they are extracted. On the other hand, in the case of EXTRACT AND MOVE METHOD refactorings about 18% of the extracted methods have a decreased visibility and about 27%

Figure 14: Reusable Extracted Methods Visibility Changes

of them have an increased visibility. Therefore, when the extracted methods are moved to other classes the intention is to make reusable in more classes other than the class from which they were extracted.

Liu and Liu (2016) suggest that practitioners should consider the cost and benefits of EXTRACT METHOD refactoring. They find that extracted methods have a small chance to be reused in the future and on the average 17% of them are reused in the future. But at the same time they find that 39% of the extracted methods are reused immediately and are also reused in the future in order to remove clones. Therefore, the development of techniques for recommending the extraction of reusable methods and components has great merit. In our study we found that about 30% of reusable methods are also used to remove duplication in the same commit. We will further discuss the multiplicity of EXTRACT METHOD motivations in Section 4.4.

Remove Duplication motivation is ranked in the second place, while it is ranked fifth place in the Silva et al. (2016) study. About 25% (89111 instances) of all examined EXTRACT METHOD and EXTRACT AND MOVE METHOD refactorings have *Remove Duplication* as their motivation, which shows that the refactoring of clones is an important reason for extracting methods.

Duplication removal can be performed from multiple different source methods, as well as from a single source method. 64% of the *Remove Duplication* motivations are related to EXTRACT METHOD and the remaining 36% are related to EXTRACT AND MOVE METHOD refactorings. Figure 15 shows the number of *Remove Duplication* motivation instances, in which the involved EXTRACT METHOD and EXTRACT AND MOVE METHOD refactorings have been extracted from multiple methods or a single method, respectively. We can see that 37% of all instances are extracted from a single source method, while 63% are extracted from multiple source methods.

Due to the importance of clone detection and refactoring many researchers developed

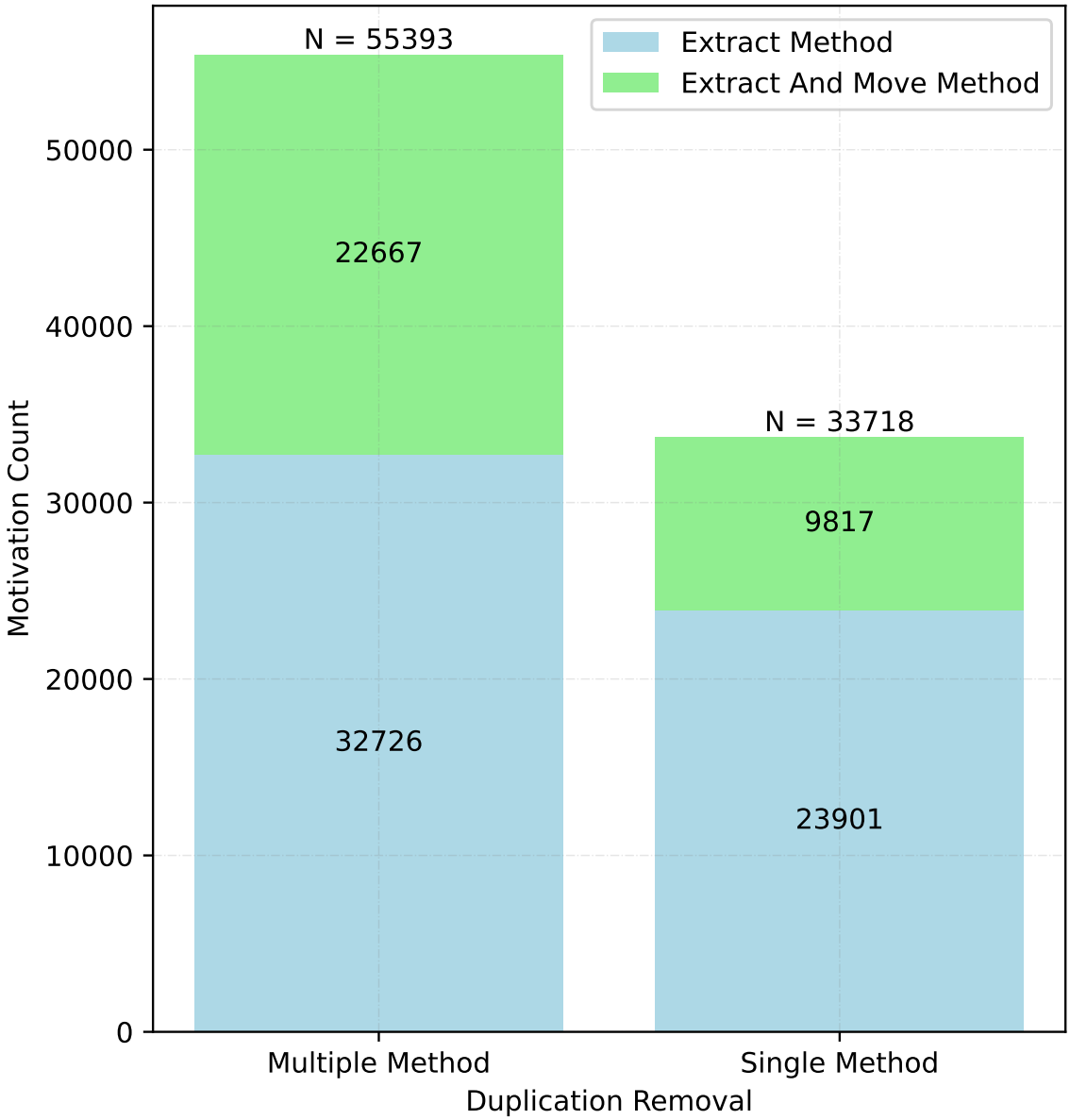

Figure 15: Multiple Method vs. Single Method Duplication Removal

methods and tools to eliminate clones. Tairas and Gray (2012) propose CeDAR that unifies clone detection and refactoring by filtering out clones and reporting only those clones that can be refactored with EXTRACT METHOD refactorings. Mazinanian et al. (2016) extended JDeodorant to import the clone detection results of CCFinder, NICAD, ConQat, CloneDR and Deckard clone detection tools, perform refactorability analysis (Tsantalis et al., 2015), and support multiple clone refactorings, such as EXTRACT METHOD, EXTRACT AND PULL UP METHOD, INTRODUCE TEMPLATE METHOD, and INTRODUCE UTILITY METHOD. In all these refactorings the differences between the refactored clones can be parameterized with Lambda expressions if needed. Chen et al. (2018) propose a pattern-based clone refactoring technique to summarize refactorings of duplicated codes and find clones that are not consistently refactored. Yue et al. (2018) propose a learning-based approach and a tool named CREC that uses 34 features for the characterization of clones to automatically suggest EXTRACT METHOD refactorings for eliminating clones. Our findings and dataset about single method and multiple method duplication removals can be used to better train the clone detection models and find features that are useful to eliminate clones using various clone-removal-related refactorings.

Facilitate Extension motivation is ranked third in our large scale study and is ranked fourth in the survey by Silva et al. (2016). 20.51% of refactoring instances in our study are detected having *Facilitate Extension* as motivation. Among these refactoring instances, 81.70% are EXTRACT METHOD and the remaining 18.30% are EXTRACT AND MOVE METHOD refactorings. We discuss the characteristics of EXTRACT METHOD refactorings that are used to *Facilitate Extension* in more detail in Section 4.3.

Although the have been some recent efforts in the recommendation of refactorings based on feature requests (Nyamawe et al., 2019, 2020), this area is still under-researched. Developers spend most of their time trying to fix bugs and implement new requirements.

Therefore, there is a great need in recommending refactorings based on the particular maintenance task they are currently working on that could help them to complete their task faster and improve the overall quality of the code involved in the maintenance task.

Decompose Method to Improve Readability motivation is ranked fourth in our large scale study and third in the survey by Silva et al. (2016). About 18% of all refactoring instances are detected having *Decompose to Improve Readability* as motivation. Among these instances, about 78% are EXTRACT METHOD and the remaining 12% are EXTRACT AND MOVE METHOD refactorings.

Figure 16 shows the number of *Decompose Method* motivation instances, in which the involved EXTRACT METHOD and EXTRACT AND MOVE METHOD refactorings introduce multiple extracted methods, and a single method, respectively. in 65% of these instances the source method is decomposed into multiple methods, while in the remaining 35% only a single method is extracted. This results shows that in general when the developers' intention is to decompose a method, they typically extract multiple methods from the source method. In the literature, there as many refactoring recommendation systems targeting the decomposition of methods. Some of the most well-known refactoring recommendation tools supporting the *Decompose Method* motivation are JDeodorant (Tsantalis and Chatzigeorgiou, 2011), JExtract (Silva et al., 2014), SEMI (Charalampidou et al., 2017), and GEMS (Xu et al., 2017).

Introduce Alternative Method Signature motivation is ranked fifth in our large scale study and second in the survey by Silva et al. (2016). To the best of knowledge, there are no recommendation systems targeting this particular motivation for EXTRACT METHOD refactoring. Therefore, the development of techniques for recommending the extraction of methods with an alternative signature to support different input parameter types or output types has a lot of merit.

Finally, we can observe in Figure 13 that the *Introduce Factory Method* motivation is

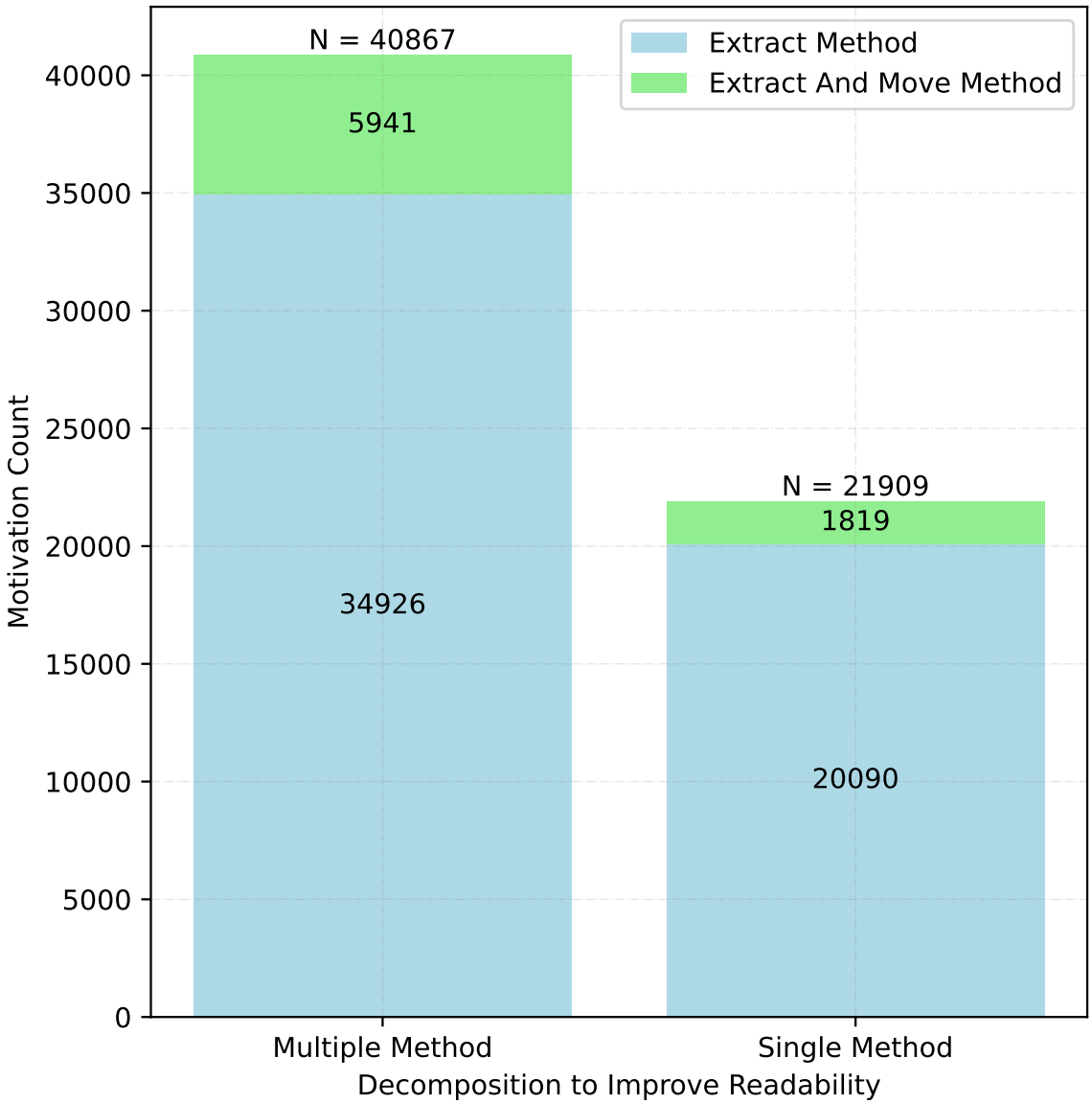

Figure 16: Multiple and Single Method Decomposition to Improve Readability

ranked sixth in our study with a much larger representation compared to the survey by Silva et al. (2016), in which it is ranked last among the 11 motivations. The development of recommendation systems supporting this motivation is quite straightforward, as it involves finding object creations or builder call chains that are complex or long enough to be extracted into a separate method. Factory methods are also great for extendibility, as they allow the consumers to create new objects without having to know the details of how they are created, or what their dependencies are.

RQ2 Conclusion: Despite the prevalence of *Reusable Method*, *Facilitate Extension* and *Introduce Alternative Method Signature* motivations, there is very limited research on refactoring recommendation systems targeting these motivations. On the other hand, the *Remove Duplication* and *Decompose Method* motivations have received great attention from researchers who developed techniques to find relevant refactoring opportunities and automate the required refactoring process.

4.3 RQ3: What are the characteristics of the EXTRACT METHOD refactorings having *Facilitate Extension* as motivation

Among all 346k EXTRACT METHOD refactoring instances, 71136 (20.51%) are related to the *Facilitate Extension* motivation, i.e., refactorings performed to facilitate the addition of new features or fixing of bugs.

To better understand how developers use EXTRACT METHOD refactorings for extension, we analyzed the added AST nodes (i.e., new statements) in the extracted method and in the source method after extraction.

Among the 71136 refactoring instances having *Facilitate Extension* as motivation, 26.66%

have added nodes in the Source Operation After Extraction (SOAE), 61.30% have added nodes in the Extracted Method and 8.41% have added nodes in both SOAE and Extracted Method. Also in 3.74% of the instances the extension is done with the use of a ternary operator in the extracted method. Therefore, the extensions are more often performed in the extracted method than the source operation after extraction.

Table 19: Self-affirmed refactoring patterns

Patterns		
Added more checks for quality factors	Fix quality issue	Reformat
Antipattern bad for performances	Fixing naming convention	Remove
Avoid future confusion	Flaw	Remove dependency
Chang	Formatted	Remove redundant code
Change design	Get rid of	Removed poor coding practice
Change package structure	Getting code out of	Renam
Cleaned up unused classes	Improv	Renamed for consistency
Cleanup	Improve naming consistency	Reorganiz
Code cleansing	Improvement	Reorganize project structures
Code maintenance for refactoring	Inlin	Replac
Code optimization	Inlined unecessary classes	Replace it with
Code quality	Issue	Restructur
Code redundancies	Make maintenance easier	Rework
Code reformat	Minor enhancement	Rewrit
Code reordering	Modify	Simplification
Code revision	Modularize the code	Simplify
Cosmetic	mov	Simplify code redundancies
Creat	Moved and gave clearer names to	Simplify internal design
Decompos	Moved more code out of	Simplify the code
Deleting a lot of old stuff	Naming improvement	Smell
Easily extend	Nicer name	Split
Encapsulat	Nonfunctional	Structural change
Enhanc	Polishing code	Structure
Enhanced code beauty	Pull some code up	Technical debt
Extend	Quality factor	Unnecessary code
Extract	Redesign	Unused code
Fix	Reduc	Use a safer method
Fix a desgin flaw	Refactor	Use better name
Fix module structure	Refactor bad designed code	Use less code
Fix quality flaw	Refactoring towards nicer name analysis	

We further used the self-affirmed refactoring patterns catalogued by AlOmar et al. (2019a) to find commit messages that are related to self-affirmed refactoring activities. Table 19 shows 89 refactoring patterns that we used. These are some keywords and phrases that are commonly used by developers when they document their refactoring activities in

commit messages.

For some patterns we used a more general subset to find matching commit messages. For instance, patterns like *Fix a design flaw* or *Fix a quality flaw* had 0 and 7 matches, respectively, but we found similar variations of the pattern by applying the general *Flaw* keyword with 14 matches. We excluded some patterns like *Merg* and *Add*. *Merg* was used in many automatically generated merge commit messages for pull requests. *Add* is also a general keyword that might not be directly associated with a certain task like adding a new feature besides refactoring. The remaining keywords were utilized to understand what kind of maintenance tasks are supported by the EXTRACT METHOD refactoring instances having *Facilitate Extension* as motivation.

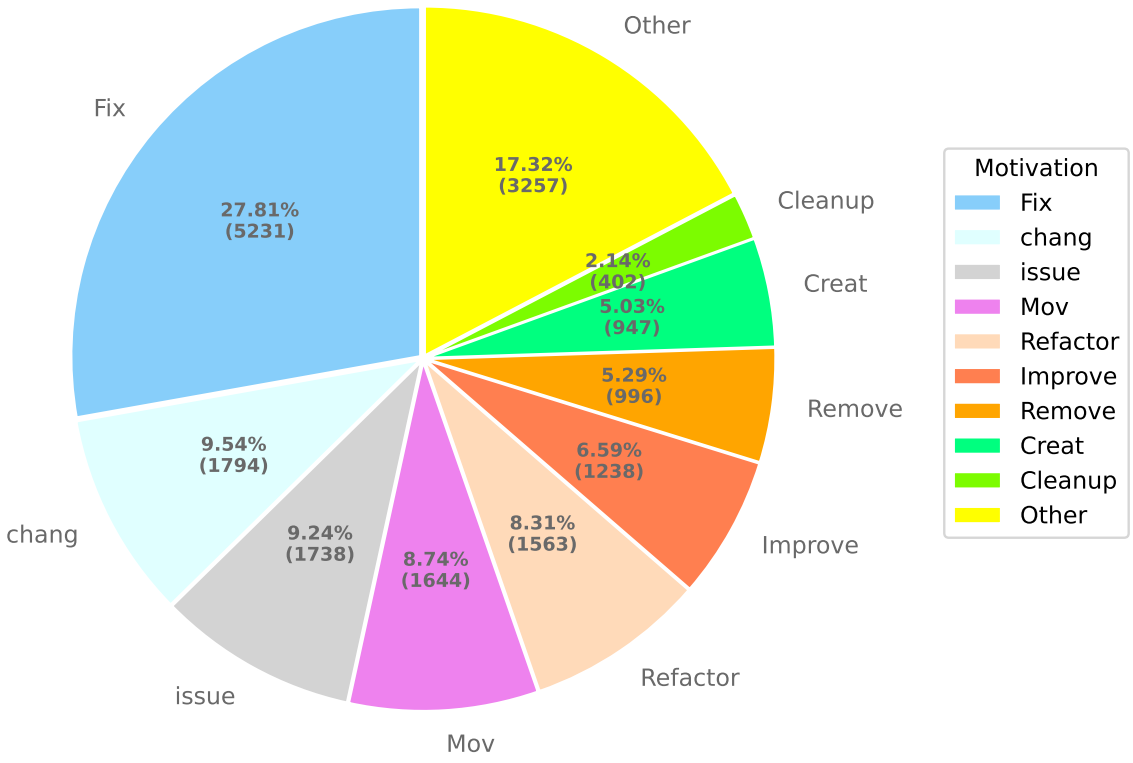

Figure 17: Top SAR patterns in message of commits that include EXTRACT METHOD with *Facilitate Extension* as motivation

Figure 17 shows the top-10 most used keywords in the message of commits including EXTRACT METHOD refactorings having *Facilitate Extension* as motivation. About 28%

of instances with self-affirmed keywords include the *Fix* keyword, which is the most frequent pattern that can be associated with a bug-fixing task. This suggests that EXTRACT METHOD refactoring is a very useful tool for fixing bugs.

The remaining self-affirmed refactoring patterns in the top-10 list cannot be directly associated with bug fixing or the implementation of a new feature, but they are clearly related with some maintenance task affecting the functionality of the project, as indicated from *Chang*, *Issue*, or the organization of the project, as indicated from *Mov*, *Remove* and *Cleanup*. The remaining 80 self-affirmed patterns constitute 17% of the matched keywords.

RQ3 Conclusion: In around 70% of the EXTRACT METHOD refactoring instances having *Facilitate Extension* as motivation, the new code related to the bug fix or the implemented feature is added in the extracted method. Moreover, by analyzing the commit messages, we found that at least 28% of the EXTRACT METHOD refactoring instances having *Facilitate Extension* as motivation, target a bug fix. Therefore, we can conclude that EXTRACT METHOD refactoring is a very useful tool for fixing bugs.

4.4 RQ4: Multiple concurrent EXTRACT METHOD Motivations

In this section, we analyze the co-existence of multiple motivations for the same EXTRACT METHOD refactoring instance. Among all refactoring instances, 56% of them have only a single motivation, while 37% of them have multiple motivations. More specifically, 34% of all refactoring instances have two different motivations, and about 2.5% have 3 different motivations. Figure 18 shows the percentages of refactoring instances falling into four different categories, namely *single motivation*, *two motivations*, *three motivations*, *no motivation*.

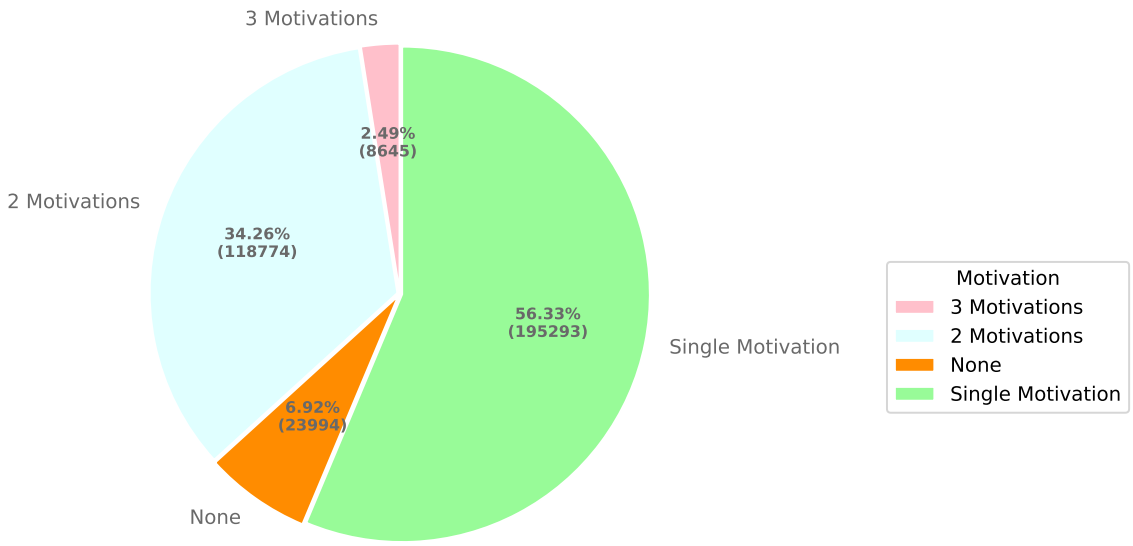

Figure 18: Extract Motivation Motivation Detection Rate

In about 7% of the instances, no specific motivation is automatically detected. We further manually analyzed a random selection of these instances and found that about 50% of them only contain one statement in the extracted method. In about 25% of them, the entire body of the source method was extracted, and the remaining 25% were not related to any of the 11 motivation categories considered in our study. These instances can serve for further improving the recall of our motivation detection rules, as some motivations were missed, but also discover new motivation categories, which were not previously documented in the literature.

We used association rule mining to better understand the relationship between multiple refactoring motivations co-existing in EXTRACT METHOD refactoring instances. The association rules we mined are shown in Table 20. We found the association of motivations among all instances with multiple motivations, and also separately for categories with only two or three motivations. We found consistently in all categories a strong association between *Remove Duplication* and *Reusable Method*. This means that when developers remove duplication, they also tend to find opportunities to reuse the extracted duplicated code.

Table 20: Association Rules for EXTRACT METHOD concurrent motivations

Multiplicity	Rule	Support	Confidence	Lift
All	$RD \rightarrow RM$	0.326	0.888	1.143
	$FE \rightarrow RM$	0.243	0.679	0.874
2 Motivations	$RD \rightarrow RM$	0.328	0.882	1.152
	$FE \rightarrow RM$	0.222	0.647	0.846
3 Motivations	$RD \rightarrow RM$	0.290	0.988	1.057
	$DIR \rightarrow RM$	0.688	0.946	1.012
	$FE \rightarrow RM$	0.528	0.944	1.010
	$FE \rightarrow DIR$	0.523	0.934	1.284

RM : Reusable Method
RD : Remove Duplication
DIR : Decompose Method to Improve Readability
FE : Facilitate Extension

RQ4 Conclusion: In 37% of the studied EXTRACT METHOD refactoring instances, we found two or three concurrent motivations. By applying association rule mining, we found that when developers remove duplication, they also tend to find opportunities to reuse the extracted duplicated code.

Chapter 5

Threats to Validity

5.1 Internal Validity

The internal threats to the validity of our study involve biases or errors. We validated and improved our refactoring motivation detection rules based on a primary oracle that was built from the actual developer responses in 222 commits (Silva et al., 2016). However, to evaluate the accuracy of our detection rules, we compared the motivations detected from our tool with manually validated motivations found in pull requests (Pantiuchina et al., 2020). The number of refactoring instances were limited for some motivation categories, such as *Enable Async Operation* and *Enable Overriding*. Therefore, the detection rules for these motivations may be further improved, if more cases are discovered in the future studies.

One more threat is that we had to reconstruct the two available motivation oracles from commit level (Silva et al., 2016), and pull request level (Pantiuchina et al., 2020), respectively, to refactoring instance level. This means we had to map the motivations given at commit and pull request levels to specific refactoring instances found in the commits. However, as explained in Section 4.1, we followed a very careful and systematic process when performing the mapping of the motivations from a higher level of granularity (i.e., commit

and pull request) to a lower one (i.e., refactoring instance).

5.2 Construct Validity

The construct threats to validity are mainly concerned with the accuracy that RefactoringMiner detects EXTRACT METHOD and EXTRACT AND MOVE METHOD refactoring operations in the commit history of a project, as well as the level of detail RefactoringMiner provides for the context in which the refactoring operations are applied (e.g., the call sites of the extracted method within the project's code base, the added statements in the bodies of the extracted and source methods). The accuracy of RefactoringMiner in the detection of EXTRACT METHOD has been shown to be very high with a precision of 99.8% and recall of 95.8% (Tsantalis et al., 2020), which gives us confidence that the vast majority of the detected refactoring instances in our dataset are correct (almost zero false positives), and that we didn't miss many true instances (i.e., low number of false negatives). Moreover, RefactoringMiner matches the refactored code at statement level, and provides a fine-grained level of details about the call sites of the extracted methods, the statements matched between the source and the extracted methods, and the newly added statements in the bodies of the extracted and source methods. This level of detail allowed us to build very accurate motivation detection rules taking into account several special cases.

5.3 External Validity

In this study, we focused on EXTRACT METHOD and EXTRACT AND MOVE METHOD refactoring types, but there is a large catalogue of refactoring types, which we did not examine. This is due to the fact that EXTRACT METHOD in very common and extensively applied by developers. Moreover, it is the only refactoring type that serves so many different motivations. Our general method can be extended to detect the motivations for other

refactoring types as well.

We selected popular Java systems of various sizes and domains that were previously used in other researches to know why developers refactor source code. In our large scale study we analyzed all the commits in 325 projects to have a better understanding of the motivations driving EXTRACT METHOD refactoring. Our approach is utilizing RefactoringMiner, which only supports Java systems. But our motivation detection rules can be generalized to other programming languages, provided that the refactoring mining tools supporting other languages can provide thorough information about the context of the detected refactoring operations to replicate this study.

Chapter 6

Conclusion and future work

Refactoring is a well-known practice among developers to improve the quality of a software system by altering its internal structure without changing the external behaviour. Among all refactoring activities, EXTRACT METHOD is widely utilized to refactor code for various motivations. In this study we propose a method for the automatic detection of the motivations driving the application of EXTRACT METHOD and EXTRACT AND MOVE METHOD refactoring operations. We conducted a large scale study on 325 open-source java projects, which included 346k EXTRACT METHOD and EXTRACT AND MOVE METHOD instances. We developed detection rules to automatically detect 11 major motivations of these refactorings. The automated motivation detection results are validated against an oracle with manually labelled motivations on refactorings taking place in pull requests with a precision of 98% and recall of 93%.

In out large-scale empirical study, we found that *Reusable Method*, *Remove Duplication*, *Facilitate Extension* and *Decompose Method to Improve Readability* are the the most common reasons for applying EXTRACT METHOD refactorings. We further studied the EXTRACT METHOD instances that have more than one motivation. About 56% of the refactoring instances had a single motivation and about 36% had multiple motivations, while 7% had no detection motivation. We performed association rule mining and found

that the *Remove Duplication* and *Reusable Method* motivations have a strong association. This means that when developers remove duplication, they also tend to find opportunities to reuse the extracted duplicated code.

The results show the feasibility and effectiveness of our approach to detect the motivations related to the EXTRACT METHOD refactoring type. In the future, we aim to extend our tool to automatically detect the motivation of other refactoring types besides EXTRACT METHOD. This will provide to researchers and tool-makers the empirical evidence required to build refactoring recommendation systems tailored to the developer needs and practices.

Bibliography

Aalizadeh, M. S. (2021). Motivation extractor. https://github.com/mosaliza/RefactoringMiner.

Abid, C., Kessentini, M., Alizadeh, V., Dhouadi, M., and Kazman, R. (2020). How does refactoring impact security when improving quality? a security-aware refactoring approach. *IEEE Transactions on Software Engineering*, pages 1–1.

AlOmar, E., Mkaouer, M. W., and Ouni, A. (2019a). Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 51–58.

AlOmar, E. A., Mkaouer, M. W., and Ouni, A. (2021). Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software*, 171:110821.

AlOmar, E. A., Mkaouer, M. W., Ouni, A., and Kessentini, M. (2019b). On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11.

AlOmar, E. A., Rodriguez, P. T., Bowman, J., Wang, T., Adepoju, B., Lopez, K., Newman, C., Ouni, A., and Mkaouer, M. W. (2020). How do developers refactor code to improve code reusability? In *International Conference on Software and Software Reuse*, pages 261–276. Springer.

Arcelli, D., Cortellessa, V., and Di Pompeo, D. (2018). Performance-driven software model refactoring. *Information and Software Technology*, 95:366–397.

Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., and Palomba, F. (2015). An experimental investigation on the innate relationship between quality and refactoring. *J. Syst. Softw.*, 107(C):1–14.

Bogart, A., AlOmar, E. A., Mkaouer, M. W., and Ouni, A. (2020). Increasing the trust in refactoring through visualization. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 334–341.

Charalampidou, S., Ampatzoglou, A., Chatzigeorgiou, A., Gkortzis, A., and Avgeriou, P. (2017). Identifying extract method refactoring opportunities based on functional relevance. *IEEE Transactions on Software Engineering*, 43(10):954–974.

Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D., and Garcia, A. (2017). How does refactoring affect internal quality attributes? a multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, SBES'17, page 74–83, New York, NY, USA. Association for Computing Machinery.

Chen, J., Xiao, J., Wang, Q., Osterweil, L. J., and Li, M. (2016). Perspectives on refactoring planning and practice: An empirical study. *Empirical Softw. Engg.*, 21(3):1397–1436.

Chen, Z., Kwon, Y.-W., and Song, M. (2018). Clone refactoring inspection by summarizing clone refactorings and detecting inconsistent changes during software evolution. *Journal of Software: Evolution and Process*, 30(10):e1951.

Derezińska, A. (2017). A structure-driven process of automated refactoring to design patterns. In *International Conference on Information Systems Architecture and Technology*, pages 39–48. Springer.

Dig, D., Comertoglu, C., Marinov, D., and Johnson, R. (2006). Automated detection of refactorings in evolving components. In Thomas, D., editor, *ECOOP 2006 – Object-Oriented Programming*, pages 404–428, Berlin, Heidelberg. Springer Berlin Heidelberg.

Dig, D. and Johnson, R. (2006). How do apis evolve? a story of refactoring. *Journal of software maintenance and evolution: Research and Practice*, 18(2):83–107.

Ferreira, I. V. (2018). *Assessing the Bug-Proneness of Refactored Code: Longitudinal Multi-Project Studies*. PhD thesis, PUC-Rio.

Hora, A. C. and Robbes, R. (2020). Characteristics of method extractions in java: a large scale empirical study. *Empir. Softw. Eng.*, 25(3):1798–1833.

Ivers, J., Ozkaya, I., Nord, R. L., and Seifried, C. (2020). Next generation automated software evolution refactoring at scale. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1521–1524, New York, NY, USA. Association for Computing Machinery.

Kaur, P. and Mittal, P. (2017). Impact of clones refactoring on external quality attributes of open source softwares. *International Journal of Advanced Research in Computer Science*, 8(5).

Kaya, M., Conley, S., Othman, Z. S., and Varol, A. (2018). Effective software refactoring process. In *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, pages 1–6.

Kim, M. and Notkin, D. (2009). Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, page 309–319, New York, NY, USA. Association for Computing Machinery.

Kim, M., Zimmermann, T., and Nagappan, N. (2014). An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Trans. Softw. Eng.*, 40(7):633–649.

Kourie, D. G. and Watson, B. W. (2012). Procedures and recursion. In *The Correctness-by-Construction Approach to Programming*, pages 161–195. Springer.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

Lin, Y., Radoi, C., and Dig, D. (2014). Retrofitting concurrency for android applications through refactoring. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 341–352.

Liu, W. and Liu, H. (2016). Major motivations for extract method refactorings: analysis based on interviews and change histories. *Frontiers of Computer Science*, 10(4):644–656.

LUO, T., bo GUO, Y., hui HAO, Y., and LI, H. (2011). Method verifying the correctness of code refactoring program. *Journal on Communications*, 32(11A):152.

Mazinanian, D., Tsantalis, N., Stein, R., and Valenta, Z. (2016). Jdeodorant: clone refactoring. In *Proceedings of the 38th international conference on software engineering companion*, pages 613–616.

Mohan, M. and Greer, D. (2017). Multirefactor: automated refactoring to improve software quality. In *International Conference on Product-Focused Software Process Improvement*, pages 556–572. Springer.

Nasagh, R. S., Shahidi, M., and Ashtiani, M. (2021). A fuzzy genetic automatic refactoring approach to improve software maintainability and flexibility. *Soft Computing*, 25(6):4295–4325.

Negara, S., Chen, N., Vakilian, M., Johnson, R. E., and Dig, D. (2013). A comparative study of manual and automated refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 552–576, Berlin, Heidelberg. Springer-Verlag.

Nyamawe, A. S., Liu, H., Niu, N., Umer, Q., and Niu, Z. (2019). Automated recommendation of software refactorings based on feature requests. In *2019 IEEE 27th International Requirements Engineering Conference (RE)*, pages 187–198. IEEE.

Nyamawe, A. S., Liu, H., Niu, N., Umer, Q., and Niu, Z. (2020). Feature requests-based recommendation of software refactorings. *Empirical Software Engineering*, 25(5):4315–4347.

Nyamawe, A. S., Liu, H., Niu, Z., Wang, W., and Niu, N. (2018). Recommending refactoring solutions based on traceability and code metrics. *IEEE Access*, 6:49460–49475.

Opdyke, W. F. (1990). Refactoring : An aid in designing application frameworks and evolving object-oriented systems. *Proc. SOOPPA '90 : Symposium on Object-Oriented Programming Emphasizing Practical Applications*.

Paixão, M., Uchôa, A., Bibiano, A. C., Oliveira, D., Garcia, A., Krinke, J., and Arvonio, E. (2020). *Behind the Intents: An In-Depth Empirical Study on Software Refactoring in Modern Code Review*, page 125–136. Association for Computing Machinery, New York, NY, USA.

Pantiuchina, J., Zampetti, F., Scalabrino, S., Piantadosi, V., Oliveto, R., Bavota, G., and Penta, M. D. (2020). Why developers refactor source code: A mining-based study. *ACM Trans. Softw. Eng. Methodol.*, 29(4).

Perkins, J. H. (2005). Automatically generating refactorings to support api evolution. In *proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 111–114.

Pinto, G., Torres, W., Fernandes, B., Castor, F., and Barros, R. S. (2015). A large-scale study on the usage of java's concurrent programming constructs. *Journal of Systems and Software*, 106:59–81.

Prete, K., Rachatasumrit, N., Sudan, N., and Kim, M. (2010). Template-based reconstruction of complex refactorings. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, page 1–10, USA. IEEE Computer Society.

Seng, O., Stammel, J., and Burkhart, D. (2006). Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1909–1916.

Silva, D., Silva, J., De Souza Santos, G. J., Terra, R., and Valente, M. T. O. (2020). Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, pages 1–1.

Silva, D., Terra, R., and Valente, M. T. (2014). Recommending automated extract method refactorings. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, page 146–156, New York, NY, USA. Association for Computing Machinery.

Silva, D., Tsantalis, N., and Valente, M. T. (2016). Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium*

on Foundations of Software Engineering, FSE 2016, page 858–870, New York, NY, USA. Association for Computing Machinery.

Silva, D. and Valente, M. T. (2017). Refdiff: Detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 269–279.

Silva, I. P., Alves, E. L., and Machado, P. D. (2018). Can automated test case generation cope with extract method validation? In *Proceedings of the XXXII Brazilian Symposium on Software Engineering*, pages 152–161.

Stefano, M. D., Pecorelli, F., Tamburri, D. A., Palomba, F., and Lucia, A. D. (2020). Refactoring recommendations based on the optimization of socio-technical congruence. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 794–796.

Tairas, R. and Gray, J. (2012). Increasing clone maintenance support by unifying clone detection and refactoring activities. *Information and Software Technology*, 54(12):1297–1307.

Tanhaei, M. (2020). A model transformation approach to perform refactoring on software architecture using refactoring patterns based on stakeholder requirements. *AUT Journal of Mathematics and Computing*, 1(2):179–216.

Tarlinder, A. (2016). *Developer testing: Building quality into software*. Addison-Wesley Professional.

Tsantalis, N. and Chatzigeorgiou, A. (2011). Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782.

Tsantalis, N., Guana, V., Stroulia, E., and Hindle, A. (2013). A multidimensional empirical study on refactoring activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, pages 132–146, Riverton, NJ, USA. IBM Corp.

Tsantalis, N., Ketkar, A., and Dig, D. (2020). Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, pages 1–1.

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., and Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 483–494, New York, NY, USA. ACM.

Tsantalis, N., Mazinanian, D., and Krishnan, G. P. (2015). Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering*, 41(11):1055–1090.

Vashisht, H., Bharadwaj, S., and Sharma, S. (2018). Analysing of impact of code refactoring on software quality attributes. *IJ Scientific Research and Engineering Trends*, 4:1127–1131.

Vassallo, C., Grano, G., Palomba, F., Gall, H. C., and Bacchelli, A. (2019). A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming*, 180:1–15.

Wang, Y. (2009). What motivate software engineers to refactor source code? evidences from professional developers. In *2009 IEEE International Conference on Software Maintenance*, pages 413–416.

Xu, S., Sivaraman, A., Khoo, S.-C., and Xu, J. (2017). Gems: An extract method refactoring recommender. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 24–34.

Yang, L., Liu, H., and Niu, Z. (2009). Identifying fragments to be extracted from long methods. In *2009 16th Asia-Pacific Software Engineering Conference*, pages 43–49. IEEE.

Yue, R., Gao, Z., Meng, N., Xiong, Y., Wang, X., and Morgenthaler, J. D. (2018). Automatic clone recommendation for refactoring based on the present and the past. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 115–126. IEEE.