Frequently Refactored Code Idioms

Ahmad Tahmid

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 (Software Engineering) at

Concordia University

Montréal, Québec, Canada

June 2020

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Ahmad Tahmid**

Entitled: **Frequently Refactored Code Idioms**

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

Master of Applied Science (Software Engineering)

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. Yann-Gaël Guéhéneuc

_____ External Examiner
Dr. Tse-Hsun Chen

_____ Examiner
Dr. Emad Shihab

_____ Supervisor
Dr. Nikolaos Tsantalis

Approved by _____
Lata Narayanan, Chair
Department of Computer Science and Software Engineering

_____ 2020

_____
Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

Frequently Refactored Code Idioms

Ahmad Tahmid

It is important to refactor software source code from time to time to preserve its maintainability and understandability. Despite this, software developers rarely dedicate time for refactoring due to deadline pressure or resource limitations. To help developers take advantage of refactoring opportunities, researchers have been working towards automated refactoring recommendation systems for a long time. However, these techniques suffer from low precision, as many of their recommendations are irrelevant to the developers. As a result, most developers do not rely on these systems and prefer to perform refactoring based on their own experience and intuition. To shed more light on the practice of refactoring, we investigate the characteristics of the code fragments that get refactored most frequently across software repositories.

Finding the most repeatedly refactored fragments can be very challenging due to the following factors: i) Refactorings are highly influenced by the context of the code. Therefore, it is difficult to remove context-specific information and find similarities. ii) Refactorings are usually more complex than simple code edits such as line additions or deletions. Therefore, basic source code matching techniques, such as token sequence matching do not produce good results. iii) A higher level of abstraction is required to match refactored code fragments within projects of a different domain. At the same time, the structural detail of the code must be preserved to find accurate results. In this study, we tried to overcome the above-mentioned challenges and explore several ways to compare refactored code from different contexts. We transformed the refactored code to different source code representations and attempted to find non-trivial refactored code idioms, which cannot be identified using the standard code comparison techniques.

In this thesis, we are presenting the findings of our study, in which we examine the repetitiveness of refactorings on a large java codebase of 1,025 projects. We discuss how we analyze our dataset of 47,647 refactored code fragments using a combination of state-of-art code matching techniques. Finally, we report the most common refactoring actions and discuss the motivations behind those.

Acknowledgments

I would like to express my special appreciation and thanks to my supervisor Professor Dr. Nikolaos Tsantalis who has been a tremendous mentor for me. The door of his office was always open for me whenever I ran into trouble or had a question about my research. Your advice on both research as well as on my career have been invaluable.

I must express my very profound gratitude to my committee members, Dr. Yann-Gaël Guéhéneuc, Dr. Emad Shihab and Dr. Tse-Hsun Chen for allocating their precious time for reviewing this thesis and giving their invaluable comments and insights.

Finally, I would like to thank my parents for providing me with unfailing support and continuous encouragement throughout my years of study. This accomplishment would not have been possible without them.

Thank you.

Ahmad Tahmid

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction to refactoring

Refactorings are source code transformations that increase the maintainability of software systems without impacting its functionality (Opdyke, 1992). The process of refactoring makes the source code more understandable and eliminates bad design practices and anti-patterns (W. H. Brown, Malveau, McCormick, & Mowbray, 1998; W. J. Brown, McCormick, & Thomas, 2000). As a result, the software becomes easier to debug, test, update and extend, and the overall value of the system increases (Du Bois, Demeyer, & Verelst, 2005; Fowler, 2018). In recent years, refactoring has become an integral part of software development. In software development, the process of refactoring consists of several distinct activities. According to Mens and Tourwé (2004), these activities include locating the code that can benefit from refactoring, deciding which refactoring type to apply, implementing the refactoring transformation, making sure that the behavior of the program is preserved after the refactoring application, assessing the impact of the refactoring on design quality, and maintaining consistency between the refactored code and other software artifacts, such as documentation, tests, and UML diagrams.

1.2 How and Why Developers Refactor

Refactorings can be applied during all the phases of code development or maintenance. If we look at how refactoring tasks are carried out with other programming tasks we can find two distinctly different tactics known as *floss* and *root canal* tactics (Murphy-Hill & Black, 2008). The floss tactic is when refactoring tasks are carried out as part of other development or maintenance tasks, such as bug fixes or feature enhancements, in order to facilitate their completion. This type of refactorings are smaller in nature and are performed more frequently. The root canal tactic is followed when the sole purpose of refactoring is treating unhealthy code and improving code quality. Dedicated resources are allocated for this purpose. The changes are larger in size and are not intermingled with other kinds of program changes. However, it is observed that developers hardly allocate dedicated time for root canal refactoring. Instead, they opt for more frequent floss refactoring (Murphy-Hill, Parnin, & Black, 2011). That means, they refactor their code, but mainly when it helps them to complete other maintenance tasks. They do not refactor, just for the sake of refactoring.

1.3 Why Existing Tools Fail

To aid developers in these tasks, software researchers have been trying to understand the common patterns and nature of refactorings for the past two decades. Many empirical studies have been performed to find out how developers prefer to refactor. Based on the outcome of these studies, several refactoring recommendation techniques and tools have been developed. However, developers often complain that the available tools do not meet their real-life development expectations and needs (Murphy-Hill, Parnin, & Black, 2012). The refactoring suggestions from these tools are not feasible to perform or not worth performing even though they are theoretically accurate. Therefore, developers tend to avoid these tools.

To understand why refactoring tools are underused, we need to look at how developers perform refactoring in real-life. Traditionally, it was believed that developers refactor code to eliminate code smells and increase code quality. However, according to Silva, Tsantalis, and Valente (2016), the actual motivation behind refactoring activities in not removing code smells. In most cases, refactorings are motivated by other maintenance tasks, such as a feature implementation or a bug fix.

For example, this study found that 82% of extract method refactorings are performed to facilitate other development tasks and only 18% are done for the sake of code quality improvement. The problem with existing tools in the market is that they propose refactorings to reduce code smells. But this is not exactly the reason why developers want to refactor. This conflict of interest explains why only 7% of the developers interviewed in the aforementioned study took any help from automated refactoring tools while performing refactorings.

Another reason behind the small adoption of existing tools in practice is that these tools are built on predefined rule sets. They do not consider what task or piece of code the developer is currently working on. They do not take into account the refactoring history of the current developer or the whole development community. Developers are afraid that the tools do not understand the current working context when the refactoring is complex and opt for manual refactoring (Silva et al., 2016). Also, in some cases, the tools produce a large number of refactoring suggestions, which are not relevant to the developers and overwhelm them (Murphy-Hill et al., 2012).

1.4 Problem Definition

To overcome these situations and to improve the accuracy of refactoring suggestions, a handful of empirical studies tried to find out how developers refactor in real life. These studies investigated common refactoring practices in the developer community. Some studies identified the most common refactoring tactics and types in open-source software repositories (Fowler, 2018; Mens & Tourwé, 2004; Murphy, Kersten, & Findlater, 2006), while others focused on how refactorings impact the quality of code (Moser, Sillitti, Abrahamsson, & Succi, 2006; Simon, Steinbruckner, & Lewerentz, 2001). However, none of these studies investigated if there are specific single-purpose code fragments a.k.a. 'code idioms' that tend to be refactored more commonly than others. In this study, we focus on finding code idioms that tend to be repeatedly refactored across projects. We define these code fragments as **frequently refactored code idioms**.

1.5 Scope of the study

Each type of refactoring serves different purposes and motivations. We focus our study on EXTRACT METHOD refactoring, because it is one of the most frequently applied refactoring operations (Negara, Chen, Vakilian, Johnson, & Dig, 2013), and typically developers extract code fragments that have a distinct functionality from the rest of the program into a separate function that could be potentially reused. Therefore, an **extracted code idiom** represents a distinct and reusable functionality having a single semantic purpose. We also limit the scope of our experiments to Java projects only since the tool we are using for refactoring identification (RefactoringMiner) works only on java code.

1.6 Contribution

Although refactoring attracted a lot of interest from the software engineering researcher community in the last couple of decades, no study has tried to find out what code idioms are refactored most. This is the first study focusing on this issue. We hypothesize that some code idioms are refactored across projects for a similar purpose. Identifying those will help in generating more relevant refactoring recommendations for programmers. For this purpose, we worked with a large dataset of refactorings. However, these codes belong to different contexts and contain a lot of context-specific information. Storing these codes as is reduces comparability and increases cost in terms of both computation and storage. Therefore, we have generated a variety of representations of codes with different levels of abstraction. We have used control flow graphs and program dependence graphs (Ferrante, Ottenstein, & Warren, 1987) to abstract the behavior of the refactored code fragments. Based on these graphs we have generated another more abstract representation, called Groum (T. T. Nguyen, Nguyen, Pham, Al-Kofahi, & Nguyen, 2009), which captures control and data flow information among statements performing API calls. We applied further relaxation on the Groum representation to filter out context-specific details. Finally, we performed graph matching on the relaxed Groum representation to identify the most frequently refactored code idioms.

In this study, we have mined, reconstructed and cross-matched a dataset of 47,647 refactorings

mined from 1,025 top Java open-source projects hosted on Github. After cross-matching the refactored code fragments using various techniques, such as a bag of token similarity, program dependence graph matching, and Groum matching, we were able to find 185 idioms that were refactored repeatedly in software repositories. Some of these code idioms were refactored up to 45 times in our dataset.

1.7 Implications

This research can be beneficial to both software engineering research and software developer communities. Understanding which code idioms are refactored most frequently can help us understand developers' common refactoring practices and preferences. Tool builders can improve their tools to prioritize refactoring suggestions and filter out unpopular ones. Since developers do not tend to trust automated suggestions, the data from this study can be used to justify automated suggestions. For example, a refactoring recommendation with a note "You applied a refactoring on similar code 10 days ago" or "7 of your teammates refactored similar code" or "45 developers from other projects refactored similar code" will be more acceptable and trustworthy from the developer's point of view. That's why we believe understanding frequently refactored code idioms can lead us to build more realistic and well-accepted refactoring recommendation tools.

1.8 Thesis contents

Chapter 2 contains a brief discussion of all the works that we found related to our problem. In Chapter 3, we are presenting the techniques we used in this study and the rationale behind them. Chapter 4 contains the details of our experiments.

Chapter 2

Literature Review

In this section, we are presenting numerous studies that tried to understand how refactoring is performed in real-life development. We are also presenting code mining and matching techniques that can be found in the literature. We will also look at existing refactoring recommendations and assistance tools, and analyze their perks and downsides. Then, we will discuss techniques for detecting similar code fragments that can be extended to identify refactored code idioms. After that, we will analyze studies on how to safely apply code transformations. Finally, we will present the techniques we will be using for our research.

2.1 Empirical studies on Refactoring Activity

In the past two decades, many studies tried to see how developers refactor in real life. Murphy et al. (2006) was one of the first researchers who monitored developers' behavior in IDE to understand refactoring activities. Many other studies followed this one, which focused on various aspects of refactoring activities such as, how refactorings are performed, what are the motivations and impacts of refactorings.

In 2011, Murphy-Hill et al. (2011) investigated how developers carry out refactoring tasks in software companies. They showed that programmers some times perform a number of refactoring operations in a batch. This means they perform refactoring as a separate task and not as a part of their regular development activity. Also, programmers perform these refactorings within a short

time period. Most of these refactorings are performed without using any automated tool. According to their study, 90% of refactorings are performed manually.

Researchers have always been curious to know what motivates refactorings. Wang (2009) interviewed 10 professional software developers about factors that motivated them to refactor code. He presented a list of internal and external factors such as self-motivation, peers pressure, etc. as motivations behind refactorings.

Few types of research tried to find the connection between bugs/errors and refactorings. M. Kim, Cai, and Kim (2011) showed that an API-level refactoring is often followed by a number of bug fixes, which probably indicates refactorings often introduce bugs in dependent code segments. A similar phenomenon is reported by Weißgerber and Diehl (2006). Rachatasumrit and Kim (2012) found that some refactoring cases led to situations where up to 50% of the test cases of a system were failing.

2.2 Code idiom mining

A code idiom is a syntactic fragment that recurs across projects and has a single semantic purpose. There has been a lot of research on code idiom mining from open source repositories. Allamanis and Sutton (2013) performed an extremely large scale study on 353 million lines of Java code to find code idioms that can be reused. Allamanis and Sutton (2014) analyzed the most common idioms found in open-source repositories. According to their report, those are related to object creation, exception handling and resource management. In another study, Allamanis et al. (2018) investigated loop idioms in repositories. The study found that loops idioms are highly repetitive and predictable. Only 50 loop idioms resemble 50% of all the loop instances in their dataset of 25 million source code lines.

Gharehyazie, Ray, and Filkov (2017) found that there exists a tremendous amount of code idioms in open source repositories that can be reused. These idioms can be identified using cross-project clone matching. They used the Deckard tool for code clone identification. However, they found most of the clones come from the same project or from projects from the same domain.

Tufano et al. (2018) introduced deep learning for finding similar code across projects. They used different representations of code such as Abstract Syntax Trees, Control Flow Graphs, and Bytecode, etc. in their experiments. This study shows how different representations of code can be used to find code idioms using deep learning and discusses how these representations impact the detection accuracy.

Lopes et al. (2017) performed a cross-project, cross-language and cross-domain code analysis to find non-trivial duplicate codes in open source repositories. They reported that 70% of the new codes are non-trivial copies of existing Github codes. They also found that 31% of the projects contain at least 80% of files that can be found elsewhere.

2.3 Refactoring Recommendation

To design an effective refactoring recommendation system, first, we need to define some specific criteria based on which we can identify potential refactoring opportunities. Some researchers in the literature suggested that recommendation systems based on the existence of code smells yield the best results, while others prefer to do it by learning from examples, or by analyzing software repositories. Most of these code-smell-based and learning-based techniques follow a post-mortem approach, which means they can recommend refactorings only for already committed or released code. However, few studies tried to recommend refactorings during development time, before the code is committed to a repository. We will try to highlight some of the most popular recommendation techniques in this section.

2.3.1 Code Smell Based

It is believed that code smells can be used for identifying refactoring opportunities in source code because they are indicators of poor software design. There are many refactoring tools built on this assumption. For example, JDeodorant is one of the most popular open-source refactoring recommendation systems. It suggests extract method, move method refactorings based on the existence of a long method, duplicate code or feature envy smells (Fokaefs, Tsantalis, & Chatzigeorgiou, 2007; Mazinanian, Tsantalis, Stein, & Valenta, 2016). Other popular clone detection tools such as

Deckard (Jiang, Misherghi, Su, & Glondu, 2007), CCFinder (Kamiya, Kusumoto, & Inoue, 2002) and NiCad (Cordy & Roy, 2011) are also developed based on this assumption. These tools detect code smells accurately yet developers rarely use these tools for finding refactoring opportunities (Yamashita & Moonen, 2013).

Yamashita and Moonen (2013) aimed to investigate the connection between code smells and code maintenance tasks and found that the role of code smells on the overall system maintainability is relatively minor. According to their empirical study, only 30% of the maintenance problems can be predicted by code smells. Furthermore, Silva et al. (2016) found that refactoring is mostly performed in order to make a feature implementation or a bug fix easier. For example, they found that 82% of extract method refactorings are performed to facilitate other development tasks and only 18% are done for the sake of code quality improvement. This explains why only 7% of the developers interviewed in the aforementioned study took the help of automated refactoring tools while performing refactorings. So, it can be said that using code smells for identification of refactoring opportunities may not be the most effective approach.

2.3.2 Learning Based

To overcome the problems that code smell based recommendation systems have, some researchers proposed systems that learn from developers. Raychev, Schäfer, Sridharan, and Vechev (2013) introduced a statistical language model-based system that finds refactoring opportunities by synthesizing examples given by developers. In their follow-up paper, they showed that this approach can work for predicting the use of API methods as well (Raychev, Vechev, & Yahav, 2014). However, the main difficulty, in this case, is training the system. It is simply not realistic for a developer to perform every refactoring just to train his refactoring tool. Moreover, this tool can only work on a single project. It will be interesting to see how this approach can scale to support cross-project refactorings.

To handle this scalability issue, Wasserman (2013) introduced Refaster, a tool that uses compilable before-and-after examples of code to learn a refactoring. After training this tool, it can refactor large-scale systems with 100% accuracy. Google has used this tool to perform a wide variety of

refactorings across Google's massive codebase. However, Refaster is a post-mortem tool and cannot be used while a developer is coding. Training the system manually with examples is also an issue.

Though existing learning-based techniques have their limitations, we believe this approach is more reasonable than smell based techniques and propose to use it for our study. However, post-development refactoring is not what developers do in real-life. So, we envision to apply a learning-based technique in development time.

2.4 Refactoring Assistants

Although useful and widely available, refactoring tools are underused. According to a study, more than 90% of all refactorings are performed by hand (Murphy-Hill et al., 2012). This is because developers often perform code maintenance tasks without having enough knowledge about the refactoring terminology used by the IDE to trigger automated refactoring operations. For instance, they do not always remember the exact name of the refactoring they are want to perform, in order to click on the right refactoring menu option of the IDE. To address this issue, tools like BeneFactor (Ge, DuBose, & Murphy-Hill, 2012) and WitchDoctor (Foster, Griswold, & Lerner, 2012) proposed systems that detect manual refactoring attempts in development time, remind developers that automatic refactoring is available, and complete the refactoring automatically if the developer wants. These systems provide development time refactoring support, which is faster than manual refactoring. The best thing about these tools is that developers do not have to remember anything before starting a refactoring task, such as the name of the refactoring or the process of using any specific tool. However, these systems have some serious limitations. First of all, none of these tools can detect refactoring opportunities by themselves. They only suggest a refactoring when the developer has already started doing it manually. The "in progress" refactoring detection mechanism is also questionable, as it is strictly rule-based. Predefined rules may not be enough to cover all possible ways of refactoring. Though these approaches claim to be refactoring recommendation systems, they are more like refactoring auto-completion systems. From a developer's point of view, these tools might have less accuracy but they certainly have better usability as they blend well with

regular development flow.

2.5 Similar Code Detection Techniques

For our study, we will try to discover refactored code idioms based on knowledge gathered from changes in software repositories. According to a study on 2,841 Java projects, changes in software repositories are highly (70-100%) repetitive (H. A. Nguyen, Nguyen, Nguyen, Nguyen, & Rajan, 2013). Many studies relied on the repetitiveness of code changes in software repositories for finding defective codes and generating automatic patches (Ke, Stolee, Goues, & Brun, 2015; D. Kim, Nam, Song, & Kim, 2013; Long & Rinard, 2016). We aim to build a database of refactored code idioms by extracting refactoring information from software repositories. However, there exist many challenges for creating such a database. For instance, studies have shown that the more distant the locations of two duplicated code fragments, the more differences (and more complex differences) they have (Santos et al., 2017; Tsantalis, Mazinanian, & Krishnan, 2015). Our biggest challenge is to find refactored code idioms not only from different locations of the same project but from different projects. In this section, we will look into some well-known code matching techniques and discuss their applicability for our study.

2.5.1 Token Based

CCFinder is one of the first clone detection tools Kamiya et al. (2002). This tool was built to identify duplicate codes in large software repositories. For this tool, the authors developed a matching algorithm that converts software code into tokens and tries to find clones based on token matching. This approach was developed only to support Type-I clones (i.e., code fragments without any syntactic differences, but with formatting differences). This tool is also limited to operate inside a single project. For this reason, it cannot be used for our study as we are more interested in identifying similar codes rather than exactly identical codes. However, if we apply relaxation techniques in the code tokenization process, it will be possible to support Type-II clones (i.e., code fragments with syntactic differences), and possibly Type-III clones (i.e., code fragments with changed, added, or deleted statements).

2.5.2 Token Based with Filtering

Another more recent token-based approach is SourcererCC (Sajnani, Saini, Svajlenko, Roy, & Lopes, 2016). The motivation of this study was to introduce a technique that can detect both exact (Type-I) and inexact (Type-II, Type-III) clones within a large number of different project repositories. It operates in two phases. In the index creation phase, it parses the code blocks from the source and tokenizes them. Then it builds an inverted index mapping the extracted tokens to the blocks that contain them. It uses a filtering heuristic to construct a partial index of only a subset of the tokens in each block. In the detection phase, it iterates through all of the code blocks and retrieves their candidate clone blocks from the index. As per the filtering heuristic, only the tokens within the sub-block are used to query the index, which reduces the number of candidate blocks. After candidates are retrieved, SourcererCC uses another filtering heuristic, which exploits ordering of the tokens in a code block to measure a live upper-bound and lower-bound of similarity scores between the query and candidate blocks. Candidates whose upper-bound falls below the similarity threshold are eliminated immediately without further processing. Similarly, candidates are accepted as soon as their lower-bound exceeds the similarity threshold. This is repeated until the clones of every code block are located. SourcererCC exploits symmetry to avoid detecting the same clone twice. The authors evaluated the scalability, execution time, recall and precision of this approach, and compared it to CCFinder (Kamiya et al., 2002), Deckard (Jiang et al., 2007), NiCad (Cordy & Roy, 2011) and iClones (Göde & Koschke, 2009). They used BigCloneBench benchmark of 25K projects (250MLOC) for their case study and found that SourcererCC has both high recall and precision, and is able to scale to a large inter-project repository. It is also twice as fast as its nearest competitor CCFinder. In a newer work (Saini, Sajnani, Kim, & Lopes, 2016), they introduced SourcererCC-I, an Eclipse plug-in, that uses SourcererCC to identify clones in a repository in real-time during software development. We might use a similar algorithm for indexing and searching in our refactoring database. However, we will have to increase the precision of Type-III clone detection which is relatively low for this technique.

2.5.3 Suffix Tree Based

Most of the current approaches create an index for available code repositories to find code clones. The index creation, however, is a very costly process and it may not be worth the effort if the analysis is done only once. That is why Koschke (2014) introduced suffix trees to obtain a scalable comparison of source code without the need for index creation. They used this approach to identify license violations in a suspected system by comparing its code to an ultra-large corpus of open-source code. Basically, they generate a suffix tree for the subject system which they are willing to match against the code-corpus. Once the suffix tree is created, they take every file of the corpus and compare it against the suffix tree. This step retrieves all subsequences of the file that are also contained in at least one file represented in the suffix tree. This technique allows them to compare in time linear to the length of the file. Their case study shows that this approach is faster than current index-based techniques. However, if we try to apply a suffix tree-based approach for storing refactorings mined from the commit history of a large number of projects, we will have to store all the raw refactored codes in our database. That means we might end up storing a huge amount of unnecessary data. If our refactoring database is too large, it might not be possible to recommend refactoring opportunities in real-time.

2.5.4 Code Transformation Based

To find similar codes, de Sousa et al. (2016) proposed a system called REFAZER, which learns from code edits that developers perform and saves them using a domain-specific language (DSL). Using this DSL, it identifies the similar phenomenons in code and tries to perform the same edits automatically. In their evaluation conducted on 4 programming tasks performed by 720 students, their technique was able to fix 87% of incorrect codes. The problem with this system is that it tries to identify a single type of edit at a time. If there is a huge database of previous edits, it will not be feasible to run this system for each edit. We could use the same DSL to identify similar refactoring edits in the mined repositories and build the refactoring database, but unfortunately, it supports only small edit operations while refactoring operations typically involve large edits.

2.5.5 Graph Based

Li, Saidi, Sanchez, Schäf, and Schweitzer (2016) came up with a graph-based similarity matching technique to facilitate retrieve, repair, and reuse of code across repositories. They generate the data flow graph and API call graph for a program, compute the Weisfeiler-Leman kernel for each graph and match it with other programs by performing graph isomorphism testing. After performing a case study on 1280 Java programs, their approach was able to detect 78% of similar programs. Though this approach has pretty high accuracy and recall, it operates at the program level. Moreover, it uses graph isomorphism testing which has a very high computational complexity and might increase memory usage and decrease response time dramatically for real-time recommendation (Qu, Jia, & Jiang, 2014). Therefore, we cannot use this approach unless we configure it to work at the method level and use some filtering mechanism to reduce the required graph matching combinations.

2.5.6 Graph Based with Minimization

A. T. Nguyen and Nguyen (2015) introduced a graph-based statistical model named GraLan for analyzing coding patterns and providing code suggestions. GraLan can learn from source code repositories and compute the appearance probabilities of any graphs given the observed graphs. The authors also developed an API suggestion engine and an AST-based language model named ASTLan. ASTLan can analyze the AST of code repositories to detect common syntactic templates and suggest the next valid syntactic template in development time. Their case study shows that their engine is more accurate in API code suggestion than the state-of-the-art approaches, and it can suggest the correct API in 75% of the cases. ASTLan also has considerably high accuracy (between 29.8% to 69.5% according to their experiments). This graph-based system can be applied for finding refactored code idioms in our study. It will help to retrieve multiple refactoring suggestions with high accuracy. We will use a sub-graph filtering algorithm to cut down the number of pairwise matching combinations. It will make the response time faster and enable refactoring recommendations during development time.

2.6 Code Transformation Techniques

Finally, once we can identify the most frequently refactored code idioms, we will focus on applying those on a new code. The challenge here is to find the appropriate recommendation from the refactored code idiom database for the current situation and suggest a behavior-preserving transformation. In this section, we will discuss topics regarding the application of refactoring in code.

2.6.1 Transformation Opportunity Detection

To recommend real-time refactoring opportunities, we should be able to match the code currently developed by a developer with a frequently refactored code idiom. However, most of the matching algorithms take a post-mortem approach and cannot support matching while a developer is still coding. To handle this problem Lee, Roh, Hwang, and Kim (2010) introduced an algorithm that supports instant code search. At first, they convert AST into a multi-resolution numerical vector (also known as characteristic vectors) as proposed in Deckard (Jiang et al., 2007). These characteristic vectors for ASTs are often high-dimensional and not very efficient for finding matching vectors. This is why they apply a dimension reduction algorithm on the characteristics vectors and index those using a multidimensional indexing structure based on R*tree (Beckmann, Kriegel, Schneider, & Seeger, 1990). R*trees are faster for executing queries on. Their approach could yield a sub-seconds response-time during their case study on more than 1.7M code segments. It also managed to keep over 70% precision and recall. However, it is not mentioned in this paper how it performs for identifying different types of clones. If we want to use the R*tree for finding similar codes, we must ensure that it finds near-miss clones with high accuracy.

Su, Bell, Harvey, et al. (2016) introduced an efficient technique named DYCLINK to find similar codes from a large code base. DYCLINK records instruction-level traces from sample executions of a program and creates a dynamic dependence graph from it. Then it applies its specialized sub-graph matching technique LinkSub to find out similar code. LinkSub can reduce a significant amount of matching pairs by analyzing all sub-graphs. Using this technique, they managed to analyze 422 million sub-graphs in 43 minutes. The only problem with this approach is it needs instruction-level traces to find similarities. This means all the projects have to be built first. It will not be able to

work on individual commits without building the whole project. This approach is not suitable for our study because it is simply not possible for us to compile all the existing projects before starting refactoring. For the same reason, we cannot use other dynamic clone detection algorithms such as ScalClone (Farhadi et al., 2015) and Vivo Clone (Su, Bell, Kaiser, & Sethumadhavan, 2016), even though they have a lower response time and high accuracy.

2.6.2 Applying Transformations

Finally, once we can identify frequently refactored code idioms, we will focus on actually applying those on a new code. We need to make sure the proposed code transformation is feasible and safe. We found a few research works that tried to apply code transformations safely. Tsantalis et al. (2015) showed how program dependence graph analysis, statement-level mapping and precondition checking can be used to safely apply code transformation. Meng, Kim, and McKinley (2011) also proposed a tool that can safely apply code transformations learning from examples. This tool identifies context-specific portions of code and replaces those to apply the transformation in new situations. They applied this principle in one of their more recent studies to refactor duplicated code without generating any error (Hua, Kim, & McKinley, 2015). Therefore, we can say that if we can successfully extract program dependence graphs and context-specific portions from frequently refactored code idioms, it is possible to safely apply those in new situations. And we must also develop a set of safe transformation preconditions and come up with a mechanism to check those. Finally, before suggesting any refactoring to a developer, we plan to rank the feasible opportunities. Mondal, Roy, and Schneider (2016) showed that for ranking code transformations the combination of frequency and recency technique is 16% better than only frequency and 57% better than only recency-based techniques. Therefore, we propose to use this hybrid technique to rank refactoring opportunities.

2.7 Use of the Literature in this Study

After reviewing the literature, we found that there is a need for a system that does not rely on code smell detection and does not require manual training for suggesting development-time

refactoring. The recommendation system must learn from existing refactoring history available in various repositories for providing accurate suggestions. To this end, we should create a database of refactorings extracted from the commit history of open-source projects. However, storing the entire code (before and after applying refactoring) for future use is not feasible. This is why we intend to use the graph-based GraLan approach to minimize the code segments into smaller graphs (A. T. Nguyen & Nguyen, 2015). This will save us storage space, as well as computational power when matching codes. One alternative to this approach can be the token-based clone detection tool SourcererCC (Sajnani et al., 2016). This technique can reduce storage and CPU usage by converting code into bags of tokens and the experiments performed in this study show that this technique is quite accurate and fast for code similarity matching. R*tree based 'Instant Search' approach might also be appropriate for our study (Koschke, 2014). However, before using this technique we need to perform some preliminary studies to check how it performs for Type-III clones. Applying the refactoring is our next challenge after detecting frequently refactored code idioms. For applying the refactoring we plan to follow SyEdit and RASE approaches (Hua et al., 2015; Meng et al., 2011). In these two papers, it is shown how to use context extraction techniques to apply similar transformations for different codes. Finally, for ensuring the safety of the transformation, we plan to use the PDG Matching and Precondition Checking techniques as proposed by Tsantalis et al. (2015).

Chapter 3

Research Methodology

This section describes the three major stages of this study. We started our work by creating a database of refactorings. In this stage, we identified the repositories and refactorings we want to investigate and collect code snippets for the selected refactoring instances. The next step was to gather missing details about the code fragments we collected to understand how they function. We name this step code re-construction. Once we had the behavioral information of the refactored codes, we investigated different approaches to compare them and find repetitions (i.e., frequently refactored code idioms). These two steps are done inside our automated tool called RefactoringMatcher. The tool is open-source and available in Github (Tahmid, 2019). Finally, we inspected manually the results to infer the motivation behind each refactored code idiom.

3.1 Refactoring Mining

3.1.1 Repository Selection

First, we selected all Java repositories from Github that have 500 or more stargazers. We found 2,121 projects (as of 18 May 2018). Then we filtered out projects that are not mature, inactive or not maintained anymore. The maturity conditions we set are: 1) At least 2 years old, 2) At least 10 releases, and 3) At least 100 commits. We also removed the projects that were not active for more than a year and projects that do not have any other contributor except its owner. Finally, we ended up with a set of 1,025 repositories. Table 3.1 summarizes our selection criteria.

Table 3.1: Repository selection criteria

Stargazers	Commits	Releases	Contributors	Age	Last Activity
>500	>100	>10	>2	>2 years	<365 days ago

3.1.2 Refactoring Type Selection and Mining

To perform a large scale study on the repetitiveness of refactorings we had to limit our scope to one refactoring type. There are at least 66 different types of refactorings according to Martin (2009). Among those We chose to start with the Extract Method refactoring for the following reasons: 1) It is frequently performed by developers (Murphy et al., 2006; Murphy-Hill et al., 2011), 2) It is driven by a large number of different motivations (Silva et al., 2016), 3) There exist multiple refactoring mining tools that can identify these refactorings with high precision and recall, such as RefFinder, RefDiff, RefactoringMiner, 4) The findings of the study can be used for other method level or code fragment level refactorings such as inline method, move statements, split loop etc. Among the refactoring detection tools RefactoringMiner appears to have the highest precision (0.93) and recall (0.72) (Tan & Bockisch, 2019; Tsantalis, Mansouri, Eshkevari, Mazinanian, & Dig, 2018) and for this reason, we chose to use this tool for our experiments. On our selected 1,025 Github repositories, we executed RefactoringMiner and scanned 3,106,682 commits to find all extracted methods. We included all instances reported by the tool, but excluded some duplicate code refactorings within the same commit. For example, if 5 duplicate code fragments in a project are extracted and unified into a single method, RefactoringMiner reports 5 separate extract method refactoring instances. We treat these cases as a single refactoring operation because our goal is to find frequently refactored code idioms within different commits and projects, not within the same commit and project. In the end, we ended up with a refactoring dataset of 47,647 Extract Method refactorings.

3.2 Code Re-construction

A refactoring instance reported by a refactoring detection tool typically contains information about the nature and the location of the refactoring. It does not always expose details about the code that was refactored. For example, we can see a variable being used in the refactored code

or a method being called. However, we do not know where this variable or method is declared. If multiple variables or methods exist with the same name, we do not have enough information to understand which one is being used by the refactored code. For this reason, it is important to qualify all code elements. A type reference `T` should be qualified as `packageName.T`, where `packageName` is the package in which the type is declared. A field `f` declared in type `T` should be qualified as `packageName.T.f`. Finally, a method `m()` declared in type `T`, should be qualified as `packageName.T.m()`.

When a project is built, the compiler does the qualification which we could use in our case. However, building such a large number of repositories at 47,647 commit versions is not something that is manually feasible. The building process can not be automated either as it requires manual intervention to resolve library dependencies and sometimes compilation errors. Performing these tasks at each commit makes this approach infeasible (Tufano et al., 2017). To overcome this problem, we decided to extract binding information from codes without trying to build them. The following sections describe how we managed to collect qualified names of the variables and types and methods signatures from partial and non-built codes.

3.2.1 API Finder Tool

We developed the tool API Finder to help us re-construct partial code segments that cannot be built. This tool is designed to provide fully qualified method signatures, qualified type names, and qualified field names from its database when queried. Initially, the database only contains information about standard Java API libraries. Whenever it is queried about a type or method it does not recognize, it searches in Maven Central Repository and downloads the relevant jar files. It iterates through the class files inside the jars and stores their public class information, such as qualified names of types and fields, and method signatures. It also processes the import statements declared in Java compilation units and the dependency elements of the POM file of the current project. POM stands for Project Object Model and is an XML representation of a Maven project held in a file named `pom.xml`. Moreover, if the repository address or local project location is provided, the tool can find release jars or local libraries to update its database. However, this tool cannot help if the queried type, method or field is not found using any of the resources mentioned

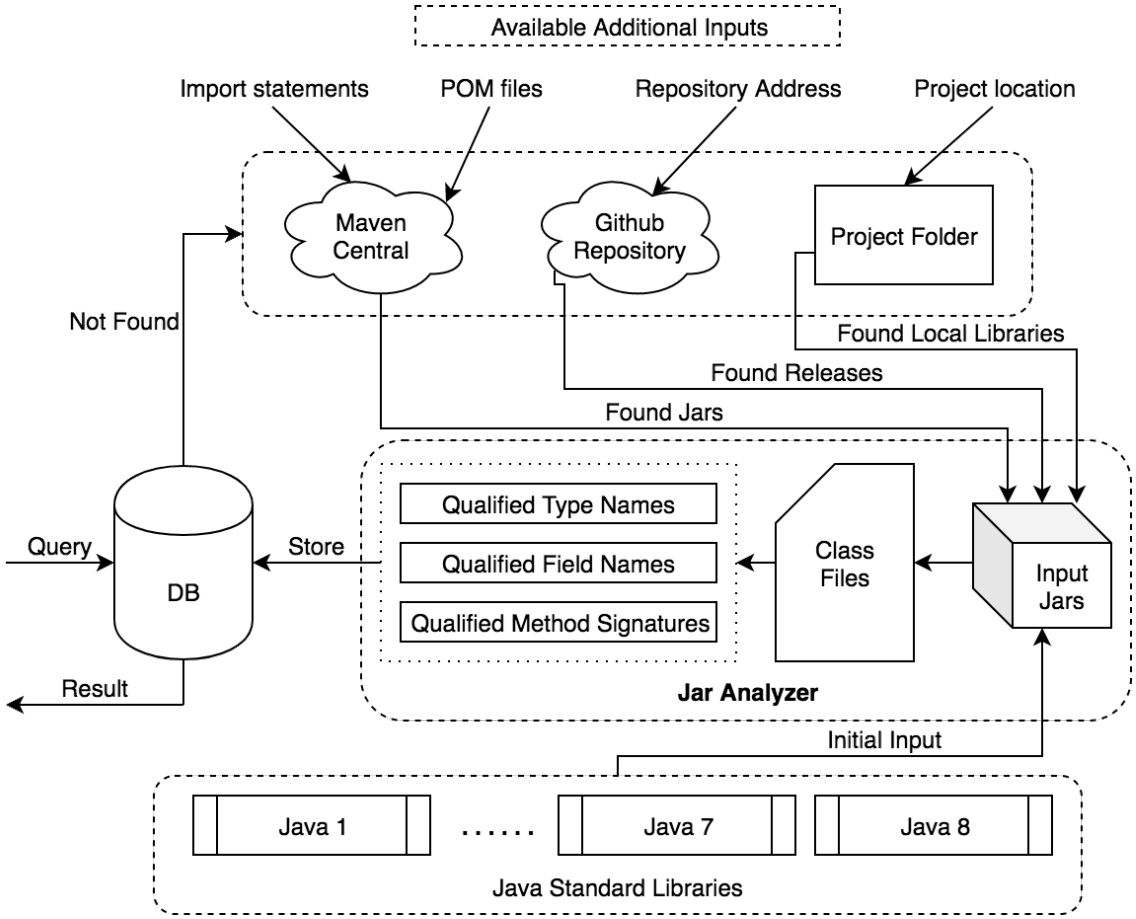

Figure 3.1: API Finder Tool

above. Figure 3.1 shows how API Finder works. The tool is available in GitHub in the repository JarAnalyzer (Tahmid & Tsantalis, 2019).

3.2.2 Resolving Fields, Type and Method Signatures

We need to understand how every variable is declared, initialized and used in the extracted codes. To accurately detect that, we need to find out the qualified types and names of these variables. We start by going to all of the refactoring commits and download the files in which we found refactoring operations. For each variable used in the refactored code fragments, we attempted to identify its declaration statement, and thus find the type of the variable. If the type is not primitive, we try to get its qualified name using the API Finder tool. We pass the unqualified name to the tool along with all the import statements declared in the Java file. API Finder uses the import statements

to locate jars that contain packages with the same names. Then, it examines those jars and returns the qualified name. For imports that are Java API libraries or any popular library that can be found in Maven Central, the tool is able to find the qualified name. For local types or uncommon libraries, we have to use a different approach, which we will discuss in the next subsection.

For resolving the qualified signature of a called method, first, we attempt to find a method with the same signature declared in the current file. If found, we utilize the API Finder to get the qualified return type, and parameter types and using that information we generate the qualified method signature. For a method that is not declared locally, we pass the called method's name along with the number of arguments, and list of import statements to API Finder. API Finder searches all packages matching the import statements and returns all the methods that are declared there with the same name and parameter count. Finally, when found, it returns the qualified signature enabling us to identify any method call uniquely.

3.2.3 Handling Unresolved Fields, Type and Signatures

API Finder is able to resolve most of the qualified names and signatures; however, in some cases, it cannot find anything. For situations like that, we had to help API Finder with additional information to enrich its database. First, we provide the Github address of the repository, so that it can find and download a released jar of the project. The tool downloads the release closest to the refactoring commit date and stores all qualified names and signatures. This way we resolve any local method call and type usage.

For external types and calls, we use two techniques. The first one is passing the POM file to API Finder. A POM file is a file located inside a java repository with a Maven build system that states all the library dependencies of the project and where to find those libraries. Using the POM file, API finder can locate all needed external libraries and enrich its database. The second technique is to provide API Finder all the jars located inside a project. For example, jars located inside the lib folder.

However, if a project does not use a POM file, a released jar in its repository, and locally copied libraries, we have no other option, but to use the unqualified name. Unless we encounter multiple unresolved variables or calls in the same extracted method with the same name, it won't create a

significant complication for us. In our dataset, we were able to uniquely identify all the variables and method calls and did not face this rare situation.

3.3 Generating Appropriate Code Representation for Matching

For finding frequently refactored code idioms, first we have to decide how to compare the codes. We generated the following representations and compared them to each other and finally, picked one to conduct our experiment. In this subsection, we will discuss these representations and how we used them.

3.3.1 Bag of Tokens

As we are looking to find refactored code idioms with similar characteristics, we decided to utilize a proven code clone detection technique. From the history of clone detection, we found that the bag of tokens representation is a good way to find codes that are similar but exactly not the same. In our case, we do not expect codes from different repositories to be identical, that's why we found this technique to be interesting.

For utilizing this technique, we broke down the refactored codes into tokens. We took any method that has more than one statement as a valid code block. Because we wanted to avoid getters and setters in our experiment since they produce a huge number of meaningless matches. Then we generated bags of tokens from each of these methods without keeping any order. While tokenizing, we considered some language-specific factors. For example, we considered `else if` as one token instead of separate `else` and `if`. We used the TXL tool for automating this process. After that, we compared the bags of tokens with each other and obtained all similar bags that have at least a certain percentage of tokens in common. We performed this matching using the SourcererCC clone detector. This tool applies some other normalization techniques while performing code matching to reduce language-specific similarities. After completing this process, we got a list of similar code blocks that exist in the refactored codes that we extracted.

However, we found this technique not to be very reliable. It does not consider the order of the tokens; hence the detected matches may be functionally very different. For example, the following

matched code fragments, have different functionality.

Listing 3.1: Function1()

```
void Function1(){
  obj = new Object();
  if (obj.field == null)
   obj.field = x;
}
```

Listing 3.2: Funtion2()

```
void Function2(){
  obj.field = x;
  if (obj.field == null)
   obj = new Object();
}
```

3.3.2 Program Dependence Graphs

Since the bag of tokens approach does not match the logical structure or functional behavior of the code, we experimented with the Program Dependence Graph (PDG) representation. A program dependence graph is a graph representation of the control structure and data flow in a code fragment. It shows how each statement controls the execution of other statements and how data is passed between statements. Therefore, it is a good representation of the functionality of code.

We construct the PDGs in a couple of steps. In the first step, we extract the abstract syntax tree (AST) for all refactored code fragments. Then, we analyze the branching structures in the code to create the control flow graph (CFG) of the programs. In this graph, every node represents a statement, and every edge directs to a statement that will be executed next. The CFG of the functions in Listing 3.1-3.2 is shown in Figure 3.2.

Next, we add data flow information to the control dependence graph. We have already identified all variables in the code with their qualified names. Now, we locate the statements where each of the variables is declared, initialized and used. For each variable, we add data dependence edges between statements of the CFG based on the execution flow. For example, if a variable is initialized in statement `x` and then used in statement `y`, and there is no other statement modifying the value of the variable in the control flow paths starting from `x` and reaching `y`, we add a data dependence edge from statement `x` to statement `y`. We also add another kind of edge called the anti-dependence edge. An anti-dependence edge for a variable from statement `x` to statement `y` denotes that `x` is using the value of the variable, which is later modified in `y`. Therefore the sequence order of these two statements is important to preserve the functional behavior of the program. Figure 3.2 shows

the data dependencies of the functions in Listing 3.1-3.2.

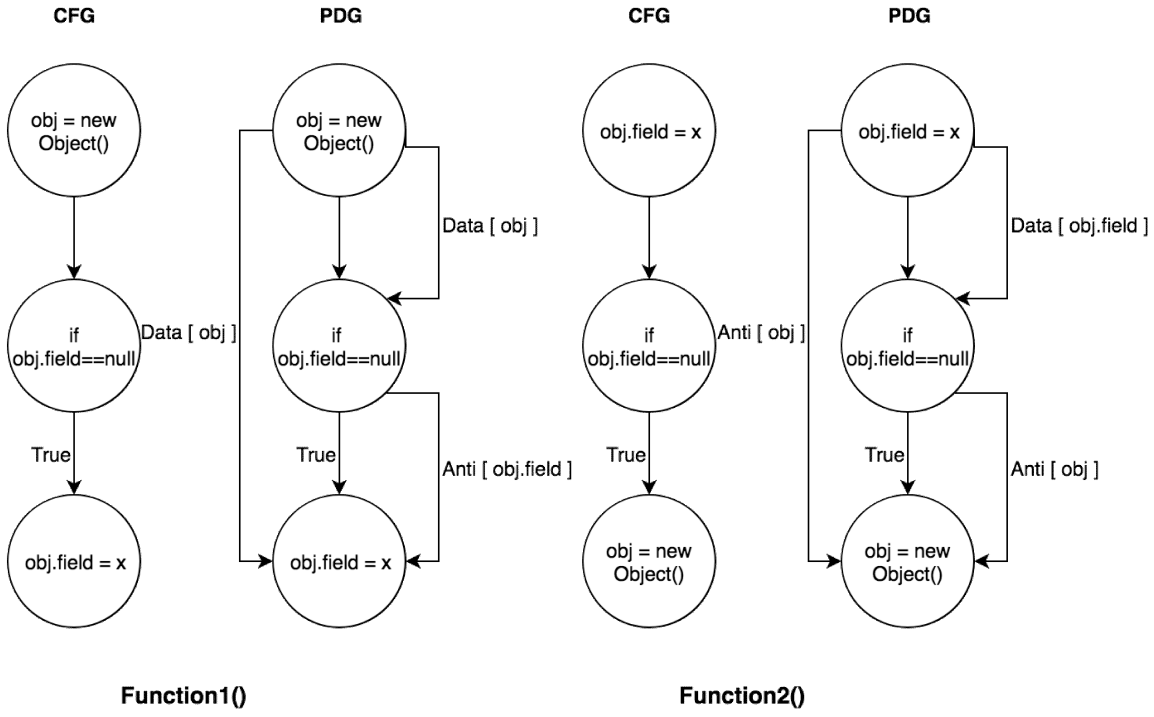

Figure 3.2: CFGs and PDGs for Listing 3.1-3.2

We can use the PDGs of the refactored code fragments and match them against each other using any graph isomorphism algorithm to find codes with the same functionality. However, our intention is not to find exact duplicates, and there is a little possibility to have such duplicates between different repositories. So, we need to relax this graph in such a way that the edges remain the same, but the contents of the nodes become less context-specific. However, we can not remove all the information from the nodes. For example, in listing 3.3-3.4, the PDGs for `Function2()` and `Function3()` will be same as the PDG of `Function2()` in Figure 3.2. The only difference will be in the contents of the last nodes (`obj=new Object()` vs. `obj=new Object(y)`). We need to find a way to preserve this difference in the graph without keeping the whole content of the node.

Listing 3.3: Function2()

```
void Function2(){
  obj.field = x;
  if (obj.field == null)
    obj = new Object();
}
```

Listing 3.4: Funtion3()

```
void Function3(){
  obj.field = x;
  if (obj.field == null)
    obj = new Object( y );
}
```

3.3.3 Groums

There are two main concerns with PDG matching. On one hand, if we keep the contents of the nodes, a PDG becomes too much context-specific. Two PDGs need to have exactly the same statements to be matched. On the other hand, if we completely ignore the contents of the nodes, any two nodes having the same incoming and outgoing edges will be matched. This means, statements `func(x).func(y)` and `func(x,y)` could be matched, since they have the same incoming and outgoing data dependencies. Similarly, if we look at the code in listing 3.3-3.4, the last lines of the two methods are different, however, the data and control dependencies are the same. Hence, if we run a graph matching on the two PDGs and do not consider the contents of the nodes, these two codes will be marked as a match. Therefore, we decided to import more information from the contents of the nodes to the graph. We expanded complex statement nodes to multiple nodes using the following Groum composition rules (T. T. Nguyen et al., 2009).

(1) Cascading Call Node:

`x.function()` to x$\longrightarrow$`function`

(2) Method Call Node:

`function(x,y,z)` to `(x,y,z)`$\longrightarrow$`function`

(3) If Statement Node:

`if(x)` to x$\longrightarrow$`if`

(4) While Statement Node:

`while(x)` to x$\longrightarrow$`while`

(5) For Statement Node:

`for(x,y,z) w` to `x`⟶`y`⟶`for`⟶`w`⟶`z`

Now the edges between the new nodes of the expanded statement represent the structure of the initial statement node. If we erase the contents of all but the control nodes of the graph, we will still have an abstract representation of each statement embedded in the graph as control nodes and edges. This way we can construct a Groum from a PDG that preserves data flow, control flow as well as statement structures without keeping any context-specific information.

If we construct PDGs for the functions in 3.3-3.4, we can see that they have the same graph structure in their PDGs 3.3. However, the last method call in `Function3()` has an extra parameter y. If we convert the PDGs into Groums, the `if` statement expansion rule will be applied to the second node of both of these graphs. However, the method call node expansion rule can be applied to `Function3()` only. Thus, the graphs will look different. Now, even if we remove all node contents, the graphs will still have a different structure and will not be matched using a graph isomorphism test. If two Groums generated from two code segments are isomorphic, we can say that the two code segments have identical control structure, data flow, and functionally similar statements. Therefore, we decided to use the Groum representation in our experiments.

3.3.4 Analyzing Similar Refactorings

The final and most important part of this study is inspecting the reported Groum matches. We started the analysis by grouping all the isomorphic Groums into sets. For each of the set, we checked the full code of each of the idiom. We checked if the only differences between idioms of the same sets are the types and names of the variables and methods, the operators and the values of operands. All the matches from reported by Groum matching clearly exhibited this expected behavior. Therefore, we did not involve multiple reviewers for agreement. In the next step, we removed the matches produced by forked repositories manually. In the remaining sets, we started inspecting the codes that produced these Groums. We checked the refactored codes as well as the type of project, refactoring time, commit message, etc. to understand why these code idioms are refactored most frequently.

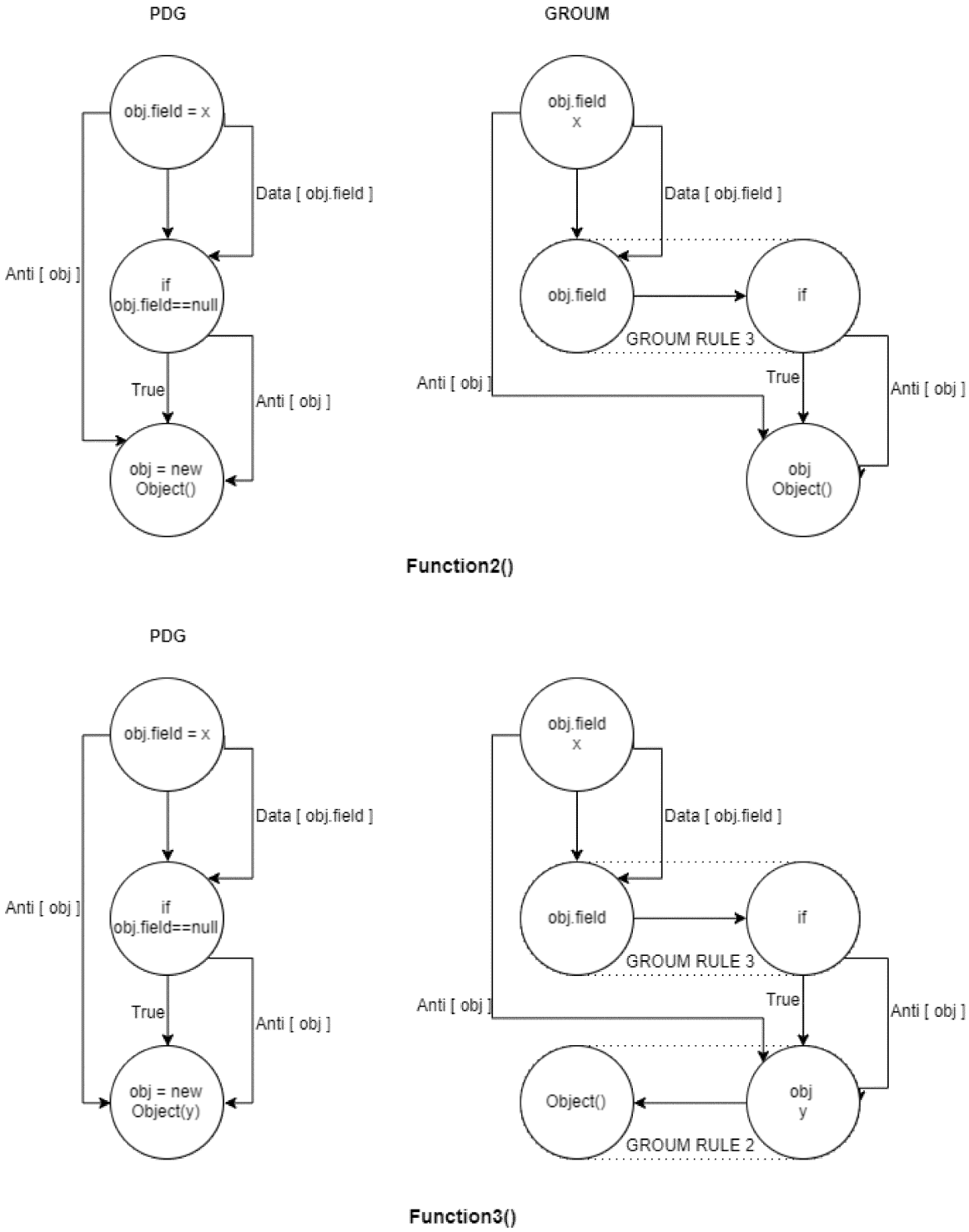

Figure 3.3: PDGs and Groums for Listing 3.3-3.4

Chapter 4

Experiment Results

To mine frequently refactored code idioms, we designed and conducted multiple experiments. First, we experimented using one of the state-of-the-art token-based code clone detection techniques, called SourcererCC. Then, we utilized graph-based source code representations. In this section, we present the results of our experiments. We discuss the most commonly refactored code idioms we have identified and the motivations behind those.

4.1 Clone Matching Experiment

4.1.1 Setup

First, we attempted to find similar refactorings using a token-based clone detection tool called SourcererCC. We took our entire dataset of 47,647 refactorings for this experiment. We broke each of the refactored code fragments into tokens and created bags of tokens for every method. For this task, we used the TXL Tokenizer. In this tokenization process, all the separators, operators and case of the tokens were ignored, and the default tokenization settings were used (`min-block-size` = 2, `min-token-count` = 2, and `token-limit` = unlimited). Then we passed the tokens to SourcererCC clone detector and observed the matches found by the tool. We only considered the refactored code fragments that contain multiple lines of code, because in single line code fragments there are no data or control dependencies. Moreover, if we include single line extracted methods,

SourcererCC reports all getter methods in the dataset as matches, since they all only have keyword `return` in common. Therefore, we configured SourcererCC to match only multi-line code fragments.

4.1.2 Result

SourcererCC matches bags of tokens for finding code clones. It finds the matches based on a threshold value ranging from 1 to 10. The higher this value is set, the more similar matches the tool returns. In our experiment, we started with the highest setting 10. However, for this threshold, the tool could not find any matching refactored code fragments belonging to separate projects. Therefore, we lowered the threshold and experimented with different values from 9 to 5.

Table 4.1 presents the matches found by SourcererCC. From the table we can see that for threshold value 8 and above we could find only one match. For threshold value 7, we found another match. This match is between two refactoring activities between projects `Recyclerview-animators` and `UltimateRecyclerView`. On April 9, 2015 a method named `doAnimateRemove()` was extracted at commit c0122b13b77064b9edec3349d47c54e3dd07eb1b in `Recyclerview-animators`. Another method with the same name was extracted in `UltimateRecyclerView` on May 11, 2015 (commit d0caccbbf9b063a3e65a24bb238365108b63ad53). The two extracted methods have the same data and control flow and look like similar refactoring activities. However, the second project here is a fork of the first one. Therefore, they already had a similar code-base. Hence, from this finding, we cannot suggest that developers repeat refactoring activities.

We performed the same experiments with lower thresholds 6 and 5 and checked the matching code fragments manually to see how similar they are. For threshold 6 we found 10 matches. 8 out of the 10 pairs were actual matches with similar data and control flows. However, 2 out of these 8 matches were between forked repositories. For the threshold value of 5, SourcererCC found 96 matches, but the precision dropped to 45.83%, when we checked the matched codes manually. For example, the two code segments in Listing 4.1 and 4.2 were reported as a match by this tool.

Listing 4.1: SCC False Match

```
@Override
public void finish(){
  if (progressDialog!=null){
    progressDialog.dismiss();
  }
  super.finish();
}
```

Listing 4.2: SCC False Match

```
@Override public void finish(){
  sResult=null;
  sCancel=null;
  sClick=null;
  sLongClick=null;
  super.finish();
}
```

In Listing 4.1 and 4.2, the two methods have 6 out of 11 tokens in common. Therefore, more than half of the tokens match with a threshold value of 5. However, these two codes have completely different functionality and cannot be considered as similar. After this experiment, we decided not to proceed with lower threshold values.

Even though SourcererCC is a widely used clone detection tool and known for its ability to find Type-III clones, in our case, it did not prove to be very useful. The 17 pair of codes we found using this approach is not enough for us to support that developers repeat refactorings. Additionally, we can say that traditional clone detection techniques are not suited for finding frequently refactored code idioms.

Table 4.1: Matched fragments found by SourcererCC for different thresholds (manually reviewed)

SCC Threshold	Match Count	True Match	Match in a Fork	False Match	Precision %
10	0	0	0	0	NA
9	1	1	0	0	100
8	1	1	0	0	100
7	2	1	1	0	100
6	10	6	2	2	80
5	96	17	27	52	45.83

4.2 Graph Matching Experiment

From our initial experiment, we figured out that clone detection techniques are not well suited for finding frequently refactored code idioms from different repositories, because these techniques are not context-independent and cannot ensure functional similarity. This is why we decided to

proceed with a technique where context-specific parts of the code are entirely ignored, and functional similarity is not compromised. Therefore, instead of matching code, we tried to match the Groum (T. T. Nguyen et al., 2009) representations of the refactored code fragments, using standard graph matching techniques. In this section, the details of the experiment along with the results are presented.

4.2.1 Setup

The same dataset from the previous experiment is used in this experiment. We constructed the control flow graph (CFG) and program dependence graph (PDG) for each refactored code fragment in the dataset. Our tool API Finder was used in this reconstruction process to resolve types, qualified names, and method signatures. The PDGs were further expanded using the Groum construction rules (T. T. Nguyen et al., 2009). Finally, the constructed graph representations were cross-matched using a graph isomorphism algorithm (Aho, Garey, & Ullman, 1972; Hecht & Ullman, 1972). In the matching algorithm, we applied exact directed edge matching without any threshold. We ignored the context-specific contents of the nodes such as variable names, variable types, and method names, to remove context dependencies. In this way, we made sure the matched codes are functionally exactly similar, but at the same time not dependent on any name or type. We only considered the refactored code fragments with more than one line of code to avoid finding unnecessary getter or setter matches. We only considered matches between different projects. Also found some matches in the forks of the following projects - SeleniumHQ/selenium, Bigkoo/Android-PickerView and google/guava. We filtered those out manually.

4.2.2 Result

We found 185 sets of matches from our dataset. The actual refactoring instance match count is 1744 which is 3.64% of the dataset. However, we are only interested in the matches that are cross-project. That gives us 723 refactorings distributed in 185 sets. These sets contain between 2 to 45 refactored code fragments each from different repositories. Each of the set gives us one idiom. For example, the code in Listing 4.3 is from project `Retrolabmda`, a back-port of Java 8's lambda expression to previous Java versions. The purpose of this code is to get a value using the parameter

Table 4.2: Frequently Refactored Idioms

Id	Name	Occurrences	Task
S1	Create an Object	45	- Create/get an object - Check a condition - Return an exception or the object
S3	Get a Value after Checking	33	- Get an object - Check a condition - Return an exception or the object
S4	Create Exception String	31	- Create an object - Create another object using the first one - Call a method and use the second object - Return a value using the first object
S5	Safely Call a Method	26	- Call a function - Throw an exception in case of error
S6	Serial Calls	25	- Call a series of methods one by one
S7	Safely Get a Value	22	- Call a function to get a value - Throw an exception in case of error
S9	Copy Values	21	- Copy values from the parameter object
S10	Start Android Activity	19	- Create an object - Pass a couple of values to the object - Call a method inside that object - Call a local method
S13	Complex Creation	13	- Create an object - Pass a couple of values to the object - Call a method - Call another method
S14	Set a Series of Fields	12	- Set a series of fields one by one
S15	Call Multiple Methods Safely	12	- Check one parameter - Call a method - Check another parameter - Call another method

Listing 4.3: Example of a frequently refactored code idiom #1

```
private String getRequiredProperty(String key){
  String value = p.getProperty(key);
  if (value == null) {
    throw new IllegalArgumentException(
      "Missing required property: " + key);
  }
  return value;
}
```

passed to it. If the value is `null`, it throws an exception. Otherwise, it returns the value. This code segment was extracted to method `getRequiredProperty(String)` on 22 July, 2013 in commit

4b6a595ccdea131ee5d286f000fef44d6a76eacd.

This refactored code idiom was found 33 times in different projects. For example, in project `weibocom/motan`, a cross-language remote procedure call framework, a similar refactoring occurred at commit 6e57ccde8fd9466dba8b3dff81f3887e8c745e12.

The extracted method in Listing 4.4 might look different from the method in project `Retrolambda`, but has the same functionality. More interestingly, the two projects are from a completely different application domain, the second refactoring occurred almost four years after the first one (on June 8, 2017), and was performed by two different developers. The commit messages do not mention anything about the developers' intention of performing a refactoring. Another example of this type is shown in Table 4.3 with id **S1**.

Out of the 185 sets of similar refactored code idioms, 11 contain more than ten refactoring instances. This means we found 11 patterns which are repeated at least 10 times in our dataset, with

Listing 4.4: Example of frequently refactored code idiom #2

```
private HeartbeatFactory getHeartbeatFactory(
                        String heartbeatFactoryName){
  HeartbeatFactory heartbeatFactory = ExtensionLoader
    .getExtensionLoader(HeartbeatFactory.class)
    .getExtension(heartbeatFactoryName);
  if (heartbeatFactory == null) {
    throw new MotanFrameworkException(
      "HeartbeatFactory not exist: " +
      heartbeatFactoryName);
  }
  return heartbeatFactory;
}
```

each instance found in a different project. In the next subsection, we will discuss these discovered refactored code idioms, their structures, and the purpose behind their application.

4.2.3 Frequently Refactored Idioms

In Tables 4.3 and 4.4, we are listing the top frequently refactored code idioms we found. We are ignoring the ones that are repeated inside one project, as they cannot be generalized. Table 4.2 shows the names of the frequently refactored code idioms, the number of times they were found, a representative instance, and the task performed by the refactored code.

Create an Object: The most commonly refactored code idiom we found, is when a developer extracts the condition checking for an object to a separate method (Listing 4.5). The extracted code fragment starts with an object creation statement or a method call to get an object followed by a conditional statement. In the conditional statement, the object is checked against a specific value. If the condition fails, an exception is thrown. Otherwise, the object is returned to the calling method.

Listing 4.5: Create an Object

```
Type func(){
  Type obj = func1();
  if (obj 'condition' VALUE1) {
    throw new Exception(VALUE2);
  }
  return obj;
}
```

Listing 4.6: Example for 4.5

```
P getPresenter(){
  P presenter = delegateCallback.getPresenter();
  if (presenter == null) {
    throw new NullPointerException("...");
  }
  return presenter;
}
```

In most cases, the conditional statement checks whether the object is equal to `null` (Listing 4.6). The motivation behind this refactoring appears to be very straight forward. Developers do not want to null-check an object every time they need this functionality. Therefore, they extract the creation/call to get the object to a separate method, so that they can reuse this functionality when needed. We found this refactoring occurring in 45 different projects in our dataset.

Get a Value After Checking: This pattern was repeated across 33 repositories in our dataset (Listing 4.7). In this particular type of refactoring, the developers pass an object to the newly extracted method as a parameter. A second object is obtained by calling a method provided by the parameter object, and it is stored in a local variable. Then a condition is checked on the local variable

(mostly a `null` check). If the condition is met, an exception is thrown inside the conditional statement. Otherwise, the value of the local variable is returned to the calling method.

Listing 4.7: Get a Value After Checking

```
Type func(Type1 param1){

  Type obj = func1(param1);




  if (obj 'condition') {
    throw new Exception(param1);
  }
  return obj;
}
```

Listing 4.8: Example for 4.7

```
String getPreferredReadabilityTextSize
                              (Context context){
  String choice = PreferenceManager
    .getDefaultSharedPreferences(context)
    .getString(context.getString(R.string
    .pref_readability_text_size),null);
  if (TextUtils.isEmpty(choice)) {
    throw new Exception("..." + context);
  }
  return choice;
}
```

This type of refactoring is performed to get a secondary object from an existing object safely and avoid `null` pointer exception. Instead of performing this check every time the secondary object is needed, developers prefer to extract this functionality to a new method and reuse it.

Create Exception String: The structure of this commonly extracted code fragment is a bit more complex than the previous two. This specific refactoring is performed to serve a very specific purpose of creating an exception string. Its code skeleton is shown in Listing 4.9 and an instance is shown in Listing 4.10.

Listing 4.9: Create Exception String

```
Type func(Type1 param1){
  Type2 obj2 = new Type2();
  Type3 obj3 = new Type3(obj2);
  param1.func1(obj3);
  return obj2.func2();
}
```

Listing 4.10: Example for 4.9

```
String stackForException (Throwable exception){
  Writer buffer = new StringWriter();
  PrintWriter writer = new PrintWriter(buffer);
  exception.printStackTrace(writer);
  return buffer.toString();
}
```

A `StringWriter` object is created at the beginning. The stack trace of an exception is passed to the `StringWriter`. Finally, a string is generated from the writer and returned. This refactored code idiom was found in 31 projects. Exception message handling is a very common piece of code that Java developers write. It appears they tend to extract this kind of code to a separate method to

reuse it.

Safely Call a Method: Developers often wrap a method call with a try/catch statement and extract it to a new method (Listing 4.11).

Listing 4.11: Safely Call a Method

```
void func (Type1 param1, Type2 param2){

  try{
    func1(param1, param2);
  }
  catch (ExceptionType1 e){
    throw new ExceptionType2(e);
  }
}
```

Listing 4.12: Example for 4.11

```
void setAclOnAce (AccessControlEntryImpl ace,
                                    AclImpl acl){
  try{
    fieldAcl.set(ace, acl);
  }
  catch (IllegalAccessException e){
    throw new IllegalStateException("...",e);
  }
}
```

The main purpose behind this refactored code idiom is to create layered exceptions to facilitate debugging. We found 26 projects in our dataset, where developers performed this refactoring at least once. The most common type is when two parameters are passed to a newly extracted method to call a function. The function is called within a try/catch block. If the function produces an exception, the exception is caught in the catch block and thrown as another exception type depending on the project.

Serial Calls: Another common refactored code idiom is when developers extract a series of method calls to the new method (Listing 4.13).

Listing 4.13: Serial Calls

```
void func(){
  func1(argument1);
  func2(argument2);

  func3(argument3);
  func4(argument4);
}
```

Listing 4.14: Example for 4.13

```
void initializeActionBar(){
  getSupportActionBar().setDisplayHomeAsUpEnabled(true);
  getSupportActionBar().setCustomView(R.layout
                                .conversation_title_view);
  getSupportActionBar().setDisplayShowCustomEnabled(true);
  getSupportActionBar().setDisplayShowTitleEnabled(false);
}
```

These calls do not have any incoming data dependence or control dependencies, which means that the execution order of these calls is not affected by branching statements, such as `if`, `switch`,

or `for` control structures. They are usually initialization, setting, or configuration method calls and developers prefer to keep them in a separate method.

Safely Get a Value: Developers often extract try-catch blocks containing a method call to a separate method for code reusability (Listing 4.15).

Listing 4.15: Safely Get a Value

```
Type func (Type1 param1){
  try{
    return param1.func1();
  }
  catch ( Exception e) {
    throw new Exception2(e);
  }
}
```

Listing 4.16: Example for 4.15

```
Object newExtensionObject (Class<?> extensionClass){
  try {
    return extensionClass.newInstance();
  }
  catch ( Exception e) {
    throw new RuntimeException(e);
  }
}
```

This kind of refactorings appeared 22 times in our data set. This is similar to the code idiom shown in 4.11, we discussed earlier. The only difference is it takes only one parameter and returns a value directly after the method call.

Copy Values: In object-oriented programming, we often pass an object as a parameter to a constructor and copy its state to instance variables of the object instantiated by the constructor.

Listing 4.17: Copy Values

```
void func(Type param){

  field1 = param.field1;
  field2 = param.field2;
  field3 = param.field3;
  field4 = param.field4;
}
```

Listing 4.18: Example for 4.17

```
void copyRestoredViewStateInstanceIntoNew
            (AbsParcelableLceViewState<D,V> old){
  this.loadedData = old.loadedData;
  this.currentViewState = old.currentViewState;
  this.exception = old.exception;
  this.pullToRefresh = old.pullToRefresh;
}
```

In many cases, more than one state variables are copied. We found that developers often extract all these assignment statements together to a new method (Listing 4.17). To be specific, we found this refactored code idiom to be repeated 21 times in our dataset.

Start Android Activity: We found that the code structure shown in Listing 4.19 was extracted 20 times in various repositories.

Listing 4.19: Start Android Activity

```
void func(){
  Type obj = new Type(VALUE1, VALUE2);

  obj.func(VALUE3, VALUE4);
  func2(obj);
  func3();
}
```

Listing 4.20: Example for 4.19

```
void handleUpgradeDatabase() {
  Intent intent = new Intent(this,
                  DatabaseUpgradeActivity.class);
  intent.putExtra("master_secret", masterSecret);
  startActivity(intent);
  finish();
}
```

However, the structure looks to be a bit different than the refactored code idioms we described so far. When we investigated the instances, we found one thing in common. All the repositories where this refactoring happened were Android projects, and this particular code block is responsible for triggering a page in an Android application. Developers extract this code segment mainly to reuse it. They call the extracted method every time they want to start a new activity.

For example, in the code in Listing 4.20, a new intent of the DatabaseUpgradeActivity class is created. The value of masterSecret is passed from the current intent to the new intent. And finally, the new intent is started and the current one is dismissed.

Complex Creation: In object-oriented programming, a constructor is called to create an object. However, in some scenarios, a couple of actions are performed on a newly created object before starting to use it. Therefore, the constructor call needs to be followed by some other method calls. Developers prefer to keep all these calls together in a separate method to increase reusability. This refactored code idiom was found 13 times in the examined repositories (Listing 4.21).

Listing 4.21: Complex Creation

```
Type func(){
  Type obj = new Type();

  obj.func1(VALUE1, VALUE2);

  obj.func2();
  return obj;
}
```

Listing 4.22: Example for 4.21

```
AnnotationConfigApplicationContext createContext(){
  AnnotationConfigApplicationContext context
            = new AnnotationConfigApplicationContext();
  context.register(PropertyPlaceholderAutoConfiguration
              .class, ZipkinUiAutoConfiguration.class);
  context.refresh();
  return context;
}
```

Set a Series of Fields: When an object is initialized, it is often required to configure a number

of fields. These kinds of configuration code fragments do not usually contain algorithmic logic like most other parts of an object. For the separation of concerns, developers sometimes extract this object configuration code to a different method. We found 12 cases where developers refactored this code idiom (Listing 4.23).

Listing 4.23: Set a Series of Fields

```
void func(){
  field1 = VALUE1;
  field2 = VALUE2;
  field3 = VALUE3;
  field4 = VALUE4;
  field5 = VALUE5;
}
```

Listing 4.24: Example for 4.23

```
void initializeConfiguration(){
  title = DEFAULT_TITLE;
  message = DEFAULT_MESSAGE;
  buttonYes = DEFAULT_YES;
  buttonNo = DEFAULT_NO;
  targetApplications = TARGET_ALL_KNOWN;
}
```

The skeleton of this code idiom (Listing 4.23) may look similar to the Listing 4.17. But these two code idioms have completely different Groum structure and motivation. In the code in Listing 4.17, values from a parameter object is assigned to different fields. The motivation here is to copy values from the parameter object. The code in Listing 4.23 does not have any parameter. Values from Constants or Final variables are assigned to different fields. The purpose of this code is not to copy values, but to do initialize a class with a specific configuration (Listing 4.24).

Call Multiple Methods Safely: In our experiments, we found that code fragments that have the structure shown in Listing 4.25, were extracted at least 12 times in different code repositories. The purpose of this type of code is to make sure that an object is not `null` before using it to call a method. This refactored code idiom is often used to close SQLite database connections safely.

Listing 4.25: Call Multiple Methods Safely

```
void func(Type1 param1,Type2 param2){
  if (param1 'condition' VALUE) {
    param1.func();
  }
  if (param2 'condition' VALUE) {
    param2.func();
  }
}
```

Listing 4.26: Example for 4.25

```
void close(Cursor cursor,SQLiteDatabase database){
  if (cursor != null) {
    cursor.close();
  }
  if (database != null) {
    database.close();
  }
}
```

Table 4.3: Frequently Refactored Idioms (1)

Id	Idiom instance

S1	`P getPresenter(){`
	`  P presenter = delegateCallback.getPresenter();`
	`  if (presenter == null) {`
	`  throw new NullPointerException("...");`
	`  }`
	`  return presenter;`
	`}`

S3	`@NonNull String getPreferredReadabilityTextSize(Context context){`
	`  String choice = PreferenceManager`
	`  .getDefaultSharedPreferences(context)`
	`  .getString(context`
	`  .getString(R.string.pref_readability_text_size),null);`
	`  if (TextUtils.isEmpty(choice)) {`
	`  throw new Exception("..." + context);`
	`  }`
	`  return choice;`
	`}`

S4	`String stackForException(Throwable exception){`
	`  Writer buffer = new StringWriter();`
	`  PrintWriter writer = new PrintWriter(buffer);`
	`  exception.printStackTrace(writer);`
	`  return buffer.toString();`
	`}`

S5	`void setAclOnAce(AccessControlEntryImpl ace,`
	`  AclImpl acl){`
	`  try {`
	`  fieldAcl.set(ace,acl);`
	`  }`
	`  catch ( IllegalAccessException e) {`
	`  throw new IllegalStateException("...",e);`
	`  }`
	`}`

S6	`void initializeActionBar(){`
	`  getSupportActionBar().setDisplayHomeAsUpEnabled(true);`
	`  getSupportActionBar()`
	`  .setCustomView(R.layout.conversation_title_view);`
	`  getSupportActionBar().setDisplayShowCustomEnabled(true);`
	`  getSupportActionBar().setDisplayShowTitleEnabled(false);`
	`}`

Table 4.4: Frequently Refactored Idioms (2)

Id	Idiom instance

S7	`Object newExtensionObject(Class<?> extensionClass){`
	`  try {`
	`  return extensionClass.newInstance();`
	`  }`
	`  catch (  Exception e) {`
	`  throw new RuntimeException(e);`
	`  }`
	`}`

S9	`void copyRestoredViewStateInstanceIntoNew`
	`  (AbsParcelableLceViewState<D,V> old){`
	`  this.loadedData=old.loadedData;`
	`  this.currentViewState=old.currentViewState;`
	`  this.exception=old.exception;`
	`  this.pullToRefresh=old.pullToRefresh;`
	`}`

S10	`void handlePushRegistration(){`
	`  Intent intent = new Intent(this,RegistrationActivity.class);`
	`  intent.putExtra("next_intent",getConversationListIntent());`
	`  startActivity(intent);`
	`  finish();`
	`}`

S13	`static AnnotationConfigApplicationContext createContext(){`
	`  AnnotationConfigApplicationContext context`
	`  = new AnnotationConfigApplicationContext();`
	`  context.register(PropertyPlaceholderAutoConfiguration.class,`
	`  ZipkinUiAutoConfiguration.class);`
	`  context.refresh();`
	`  return context;`
	`}`

S14	`void initializeConfiguration(){`
	`  title=DEFAULT_TITLE;`
	`  message=DEFAULT_MESSAGE;`
	`  buttonYes=DEFAULT_YES;`
	`  buttonNo=DEFAULT_NO;`
	`  targetApplications=TARGET_ALL_KNOWN;`
	`}`

S15	`static void close(Cursor cursor, SQLiteDatabase database){`
	`  if (cursor != null) {`
	`  cursor.close();`
	`  }`
	`  if (database != null) {`
	`  database.close();`
	`  }`
	`}`

4.3 Discussion

The motivation behind our experiments was to find out the answer to our research question: *Are there some code idioms that are refactored more frequently than others?* We deployed a well-known Type-III clone detection tool, named SourcererCC, to help us find out the answer. However, after analyzing our reasonably large set of over 47K refactorings, the tool was able to report only 96 cases of duplicate extracted methods. 54.17% of the reported matches were found not to be functionally similar after manual inspection. This inspection enabled us to see the fact that traditional code clone detection techniques might not be the right technique to identify frequently refactored code idioms.

We further continued our experiments by utilizing a graph-based matching technique with some relaxations on the Groum representation of the refactored code fragments. Using a graph isomorphism testing algorithm, we found 185 functionally similar code structures that were extracted in the same manner in different projects. These refactored code idioms were repeated in between 2 (minimum) to 45 (maximum) repositories, which confirmed our hypothesis that developers tend to repeat some specific refactoring operations. Then, we investigated the most commonly refactored idioms to understand the motivations behind the refactoring activities. Various motivations were observed. We found that improving code reusability is the most common one.

Chapter 5

Threats To Validity

5.1 External Validity

In this study, we only focused on Extract Method refactorings. There are many other kinds of refactorings that developers perform. To limit the scope of this study we had to select one refactoring for extensive investigation. We selected Extract Method, because it is one of the most frequently performed refactorings (Negara et al., 2013). Many different design problems, such as eliminating duplicate code and god classes, can be addressed by applying Extract Method refactorings. That is why we decided it is a good choice for our experiment.

Our study is restricted to Java open source projects. Therefore, we cannot claim that our findings apply to other industrial software or software implemented in other programming languages. However, the Groum representation can be extracted for any programming language following the object-oriented paradigm. Moreover, object-oriented refactoring practices are quite common in all object-oriented languages. Therefore, we believe that focusing on only one programming language is not a major weakness of this study.

We had to limit our study on a certain number of projects. We wanted to have projects from multiple domains in our database. However, determining the domains and categorizing projects according to that is another problem that is not related to our research effort. Therefore, we decided to select some criteria and we selected all projects that met our criteria. We selected all Java repositories from Github that have 500 or more stargazers, because we wanted to have all popular projects

in our data set. We found 2,121 projects (as of 18 May 2018). Then we filtered out projects that are immature or inactive. The maturity conditions we set are: 1) At least 2 years old, 2) At least 10 releases, and 3) At least 100 commits. We also removed the projects that were not active for more than a year, and projects that do not have any other contributor except its owner. Finally, we ended up with a set of 1,025 repositories. Even though we can not ensure we have projects from every possible application domain, we can certainly say that we have projects that are mature, active, with many contributors and users.

5.2 Internal Validity

The result of this experiment depends highly on the accuracy of the refactoring detection tool we used, called RefactoringMiner. This tool performs commit-based code analysis to identify refactoring instances. The tool consumes source code files that changed in a commit without compiling the source code of the projects. The lack of binding information from compiling the code may lead to some erroneous refactoring detection, because binding information can help to infer relationships between types (e.g., inheritance) and methods (e.g., method calls and overrides). We did not verify the 47,647 extract methods RefactoringMiner detected. Since this tool has the highest precision (0.93) among all refactoring detection tools (Tan & Bockisch, 2019; Tsantalis et al., 2018), we considered all refactorings reported by it to be accurate.

For our first experiment, we depend on SourcererCC Clone Detector for matching refactored code blocks. SourcererCC is a threshold-based matching system and its result depends on chosen values. Therefore, the results of our approach might vary if the threshold is changed. We tried to understand how much the threshold impacts our results by performing the same study for different thresholds. We found that, for a threshold of 60-90% similarity, the result does not differ much. That's why we opted for the default threshold value of 80%.

In our experiments, we only considered methods consisting of more than one line of code. This decision might have impacted the results we obtained. However, we had to take this decision to find more meaningful matches. With single line extract methods included in our data set, we were flooded by getter and setter method matches across all the projects, because the Groums for all

getters and setters are identical in most cases. The match counts raised to a number so high that it was not feasible for us to analyze those matches manually. Therefore, we had to leave out the single line methods. In this process, we may have missed some interesting Extract Method refactorings.

For calculating Groums we needed to resolve the qualified names of variables, types, and methods. For this purpose, we used released jars from all the repositories. However, the codes we inspected are from individual commits, not releases. Therefore, there is a chance that some of the codes from commits between two releases were not resolved properly.

For our Groum matching experiment, we used the Ullman directed graph isomorphism test technique. However, we omitted the contents of the graph nodes. This way we ensured that the contextual information from the code is ignored. This abstraction enabled us to find inter-project matches but eliminated some critical information. For example, a chain of method calls without any argument is the same as a single method call without any argument.

Chapter 6

Conclusion

Refactoring is an integral part of modern-day software engineering. Developers are encouraged to put more time and effort into refactoring than ever before. However, since the developers have to switch back and forth between bug fixes, feature implementations, and other software development tasks, they often find it difficult to determine what to refactor. Therefore, we decided to find what are the code idioms that developers refactor most to help design better refactoring recommendation systems in the future.

Our motivation behind this study was to understand if some code idioms are refactored more frequently in software repositories than others. For this purpose, we proposed and developed a tool that identifies similar refactorings by analyzing repositories. Using this tool we analyzed over 1025 open-source java projects (the one with most stargazers) and gathered a dataset of 47 thousand refactorings. We generated different representations (AST, bags of tokens, CFG, PDG, and Groum) for each of the gathered refactored code segments and cross-matched those across different projects. Through this experiment, we were able to detect 185 code idioms that were refactored at least in two different projects. Some of these idioms were refactored up to 45 times across different projects and by different developers.

The findings of this study can be utilized to aid developers as well as software researchers in many ways. For example, we have identified that developers tend to move codes related to object creation or exception handling to separate methods. Using such information, refactoring tool creators, as well as IDE developers, will be able to prioritize their refactoring recommendations

more effectively.

Even though this study focused on extract methods only, the same principle is applicable for any method level refactoring. For example, this can be applied for the inline method refactorings without making any change. Any refactoring that happens inside one method can be matched and analyzed using the same principles. This includes the move statement refactoring, split loop refactoring, decompose conditional refactoring, and many more.

The results of this study can also be used to gain developers' trust. Developers are often skeptical about refactoring recommendations provided by automated tools. If the recommendations come with a message like this - "45 developers performed this refactoring in other projects" or "10 developers in your team have performed this refactoring", the developers may accept the suggestion more easily. This way the knowledge about refactored code idioms can be utilized.

The main goal of this study was to see if developers tend to repeat refactorings on similar code fragments. The next step will be experimenting with other similarity matching techniques where we can implement a higher level of code abstraction, but preserve the structural and behavioral properties of the code. This will allow us to learn more about the popular refactoring idioms in the developer community. Besides this, experiments on other types of refactorings should be performed. That will allow us to generalize our findings.

References

Aho, A. V., Garey, M. R., & Ullman, J. D. (1972). The transitive reduction of a directed graph. *SIAM Journal on Computing*, *1*(2), 131–137.

Allamanis, M., Barr, E. T., Bird, C., Devanbu, P., Marron, M., & Sutton, C. (2018, July). Mining semantic loop idioms. *IEEE Transactions on Software Engineering*, *44*(7), 651-668. Retrieved from doi.ieeecomputersociety.org/10.1109/TSE.2018.2832048 doi: 10.1109/TSE.2018.2832048

Allamanis, M., & Sutton, C. (2013). Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th working conference on mining software repositories* (pp. 207–216). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2487085.2487127

Allamanis, M., & Sutton, C. (2014). Mining idioms from source code. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering* (pp. 472–483). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2635868.2635901 doi: 10.1145/2635868.2635901

Beckmann, N., Kriegel, H.-P., Schneider, R., & Seeger, B. (1990). The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 acm sigmod international conference on management of data* (pp. 322–331). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/93597.98741 doi: 10.1145/93597.98741

Brown, W. H., Malveau, R. C., McCormick, H. W., & Mowbray, T. J. (1998). *Antipatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.

Brown, W. J., McCormick, H. W., & Thomas, S. W. (2000). *Anti-patterns project management*. John Wiley & Sons, Inc.

Cordy, J. R., & Roy, C. K. (2011). The nicad clone detector. In *Proceedings of the 2011 ieee 19th international conference on program comprehension* (pp. 219–220). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dx.doi.org/10.1109/ICPC.2011.26 doi: 10.1109/ICPC.2011.26

de Sousa, R. R., Soares, G., D'Antoni, L., Polozov, O., Gulwani, S., Gheyi, R., ... Hartmann, B. (2016). Learning syntactic program transformations from examples. *CoRR*, *abs/1608.09000*. Retrieved from http://arxiv.org/abs/1608.09000

Du Bois, B., Demeyer, S., & Verelst, J. (2005). Does the" refactor to understand" reverse engineering pattern improve program comprehension? In *Ninth european conference on software maintenance and reengineering* (pp. 334–343).

Farhadi, M. R., Fung, B. C., Fung, Y. B., Charland, P., Preda, S., & Debbabi, M. (2015, December). Scalable code clone search for malware analysis. *Digit. Investig.*, *15*(C), 46–60. Retrieved from http://dx.doi.org/10.1016/j.diin.2015.06.001 doi: 10.1016/j.diin.2015.06.001

Ferrante, J., Ottenstein, K. J., & Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *9*(3), 319–349.

Fokaefs, M., Tsantalis, N., & Chatzigeorgiou, A. (2007, Oct). Jdeodorant: Identification and removal of feature envy bad smells. In *2007 ieee international conference on software maintenance* (p. 519-520). doi: 10.1109/ICSM.2007.4362679

Foster, S. R., Griswold, W. G., & Lerner, S. (2012). Witchdoctor: Ide support for real-time autocompletion of refactorings. In *Proceedings of the 34th international conference on software engineering* (pp. 222–232). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2337223.2337250

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Ge, X., DuBose, Q. L., & Murphy-Hill, E. (2012). Reconciling manual and automatic refactoring.

In *Proceedings of the 34th international conference on software engineering* (pp. 211–221). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2337223.2337249

Gharehyazie, M., Ray, B., & Filkov, V. (2017). Some from here, some from there: Cross-project code reuse in github. In *Proceedings of the 14th international conference on mining software repositories* (pp. 291–301). Piscataway, NJ, USA: IEEE Press. Retrieved from https://doi.org/10.1109/MSR.2017.15 doi: 10.1109/MSR.2017.15

Göde, N., & Koschke, R. (2009, March). Incremental clone detection. In *2009 13th european conference on software maintenance and reengineering* (p. 219-228). doi: 10.1109/CSMR.2009.20

Hecht, M. S., & Ullman, J. D. (1972). Flow graph reducibility. *SIAM Journal on Computing*, *1*(2), 188–202.

Hua, L., Kim, M., & McKinley, K. S. (2015). Does automated refactoring obviate systematic editing? In *Software engineering (icse), 2015 ieee/acm 37th ieee international conference on* (Vol. 1, pp. 392–402).

Jiang, L., Misherghi, G., Su, Z., & Glondu, S. (2007). Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on software engineering* (pp. 96–105). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dx.doi.org/10.1109/ICSE.2007.30 doi: 10.1109/ICSE.2007.30

Kamiya, T., Kusumoto, S., & Inoue, K. (2002, July). Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, *28*(7), 654–670. Retrieved from http://dx.doi.org/10.1109/TSE.2002.1019480 doi: 10.1109/TSE.2002.1019480

Ke, Y., Stolee, K. T., Goues, C. L., & Brun, Y. (2015). Repairing programs with semantic code search (t). In *Proceedings of the 2015 30th ieee/acm international conference on automated software engineering (ase)* (pp. 295–306). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dx.doi.org/10.1109/ASE.2015.60 doi: 10.1109/ASE.2015.60

Kim, D., Nam, J., Song, J., & Kim, S. (2013). Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 international conference on software engineering* (pp. 802–811). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2486788.2486893

Kim, M., Cai, D., & Kim, S. (2011). An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd international conference on software engineering* (pp. 151–160).

Koschke, R. (2014, August). Large-scale inter-system clone detection using suffix trees and hashing. *J. Softw. Evol. Process*, *26*(8), 747–769. Retrieved from http://dx.doi.org/10.1002/smr.1592 doi: 10.1002/smr.1592

Lee, M.-W., Roh, J.-W., Hwang, S.-w., & Kim, S. (2010). Instant code clone search. In *Proceedings of the eighteenth acm sigsoft international symposium on foundations of software engineering* (pp. 167–176). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1882291.1882317 doi: 10.1145/1882291.1882317

Li, W., Saidi, H., Sanchez, H., Schäf, M., & Schweitzer, P. (2016). Detecting similar programs via the weisfeiler-leman graph kernel. In *International conference on software reuse* (pp. 315–330).

Long, F., & Rinard, M. (2016, January). Automatic patch generation by learning correct code. *SIGPLAN Not.*, *51*(1), 298–312. Retrieved from http://doi.acm.org/10.1145/2914770.2837617 doi: 10.1145/2914770.2837617

Lopes, C. V., Maj, P., Martins, P., Saini, V., Yang, D., Zitny, J., ... Vitek, J. (2017, October). Déjàvu: A map of code duplicates on github. *Proc. ACM Program. Lang.*, *1*(OOPSLA), 84:1–84:28. Retrieved from http://doi.acm.org/10.1145/3133908 doi: 10.1145/3133908

Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

Mazinanian, D., Tsantalis, N., Stein, R., & Valenta, Z. (2016). Jdeodorant: Clone refactoring. In *Proceedings of the 38th international conference on software engineering companion* (pp. 613–616). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2889160.2889168 doi: 10.1145/2889160.2889168

Meng, N., Kim, M., & McKinley, K. S. (2011). Sydit: creating and applying a program transformation from an example. In *Proceedings of the 19th acm sigsoft symposium and the 13th european conference on foundations of software engineering* (pp. 440–443).

Mens, T., & Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on software engineering*, *30*(2), 126–139.

Mondal, M., Roy, C. K., & Schneider, K. A. (2016). An empirical study on ranking change recommendations retrieved using code similarity. In *Software analysis, evolution, and reengineering (saner), 2016 ieee 23rd international conference on* (Vol. 3, pp. 44–50).

Moser, R., Sillitti, A., Abrahamsson, P., & Succi, G. (2006). Does refactoring improve reusability? In *International conference on software reuse* (pp. 287–297).

Murphy, G. C., Kersten, M., & Findlater, L. (2006). How are java software developers using the elipse ide? *IEEE software*, *23*(4), 76–83.

Murphy-Hill, E., & Black, A. P. (2008). Refactoring tools: Fitness for purpose. *IEEE software*, *25*(5), 38–44.

Murphy-Hill, E., Parnin, C., & Black, A. P. (2011). How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, *38*(1), 5–18.

Murphy-Hill, E., Parnin, C., & Black, A. P. (2012, Jan). How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, *38*(1), 5-18. doi: 10.1109/TSE.2011.41

Negara, S., Chen, N., Vakilian, M., Johnson, R. E., & Dig, D. (2013). A comparative study of manual and automated refactorings. In *Proceedings of the 27th european conference on object-oriented programming* (pp. 552–576). Berlin, Heidelberg: Springer-Verlag. doi: 10.1007/978-3-642-39038-8_23

Nguyen, A. T., & Nguyen, T. N. (2015). Graph-based statistical language model for code. In *Proceedings of the 37th international conference on software engineering - volume 1* (pp. 858–868). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2818754.2818858

Nguyen, H. A., Nguyen, A. T., Nguyen, T. T., Nguyen, T. N., & Rajan, H. (2013, Nov). A study of repetitiveness of code changes in software evolution. In *2013 28th ieee/acm international conference on automated software engineering (ase)* (p. 180-190). doi: 10.1109/ASE.2013

.6693078

Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J. M., & Nguyen, T. N. (2009). Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th joint meeting of the european software engineering conference and the acm sigsoft symposium on the foundations of software engineering* (pp. 383–392).

Opdyke, W. F. (1992). *Refactoring object-oriented frameworks* (Unpublished doctoral dissertation). University of Illinois at Urbana-Champaign, Champaign, IL, USA. (UMI Order No. GAX93-05645)

Qu, W., Jia, Y., & Jiang, M. (2014, February). Pattern mining of cloned codes in software systems. *Inf. Sci.*, *259*, 544–554. Retrieved from http://dx.doi.org/10.1016/j.ins.2010.04.022 doi: 10.1016/j.ins.2010.04.022

Rachatasumrit, N., & Kim, M. (2012). An empirical investigation into the impact of refactoring on regression testing. In *2012 28th ieee international conference on software maintenance (icsm)* (pp. 357–366).

Raychev, V., Schäfer, M., Sridharan, M., & Vechev, M. (2013, October). Refactoring with synthesis. *SIGPLAN Not.*, *48*(10), 339–354. Retrieved from http://doi.acm.org/10.1145/2544173.2509544 doi: 10.1145/2544173.2509544

Raychev, V., Vechev, M., & Yahav, E. (2014). Code completion with statistical language models. In *Acm sigplan notices* (Vol. 49, pp. 419–428).

Saini, V., Sajnani, H., Kim, J., & Lopes, C. V. (2016). Sourcerercc and sourcerercc-i: Tools to detect clones in batch mode and during software development. *CoRR*, *abs/1603.01661*. Retrieved from http://arxiv.org/abs/1603.01661

Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K., & Lopes, C. V. (2016). Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th international conference on software engineering* (pp. 1157–1168). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2884781.2884877 doi: 10.1145/2884781.2884877

Santos, G., Paixão, K. V., Anquetil, N., Etien, A., de Almeida Maia, M., & Ducasse, S. (2017). Recommending source code locations for system specific transformations. In *Software analysis, evolution and reengineering (saner), 2017 ieee 24th international conference on* (pp.

160–170).

Silva, D., Tsantalis, N., & 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* (pp. 858–870). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2950290.2950305 doi: 10.1145/2950290.2950305

Simon, F., Steinbruckner, F., & Lewerentz, C. (2001). Metrics based refactoring. In *Proceedings fifth european conference on software maintenance and reengineering* (pp. 30–38).

Su, F.-H., Bell, J., Harvey, K., Sethumadhavan, S., Kaiser, G., & Jebara, T. (2016). Code relatives: Detecting similarly behaving software. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering* (pp. 702–714). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2950290.2950321 doi: 10.1145/2950290.2950321

Su, F.-H., Bell, J., Kaiser, G., & Sethumadhavan, S. (2016, May). Identifying functionally similar code in complex codebases. In *2016 ieee 24th international conference on program comprehension (icpc)* (p. 1-10). doi: 10.1109/ICPC.2016.7503720

Tahmid, A. (2019). *RefactoringMatcher*. https://github.com/tahmiid/RefactoringMatcher. ([Online; accessed 13-November-2019])

Tahmid, A., & Tsantalis, N. (2019). *ApiFinder*. https://github.com/tahmiid/JarAnalyzer. ([Online; accessed 13-November-2019])

Tan, L., & Bockisch, C. (2019). A survey of refactoring detection tools. In *Proceedings of the workshops of the software engineering conference* (pp. 100–105). Retrieved from http://ceur-ws.org/Vol-2308/emls2019paper02.pdf

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., & Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th international conference on software engineering* (pp. 483–494). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/3180155.3180206 doi: 10.1145/3180155.3180206

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

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., & Poshyvanyk, D. (2017). There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, *29*(4). Retrieved from http://dx.doi.org/10.1002/smr.1838 doi: 10.1002/smr.1838

Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., & Poshyvanyk, D. (2018). Deep learning similarities from different representations of source code. In *Proceedings of the 15th international conference on mining software repositories* (pp. 542–553). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/3196398.3196431 doi: 10.1145/3196398.3196431

Wang, Y. (2009). What motivate software engineers to refactor source code? evidences from professional developers. In *2009 ieee international conference on software maintenance* (pp. 413–416).

Wasserman, L. (2013). Scalable, example-based refactorings with refaster. In *Proceedings of the 2013 acm workshop on workshop on refactoring tools* (pp. 25–28). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2541348.2541355 doi: 10.1145/2541348.2541355

Weißgerber, P., & Diehl, S. (2006). Are refactorings less error-prone than other changes? In *Proceedings of the 2006 international workshop on mining software repositories* (pp. 112–118).

Yamashita, A., & Moonen, L. (2013, December). To what extent can maintenance problems be predicted by code smell detection? - an empirical study. *Inf. Softw. Technol.*, *55*(12), 2223–2242. Retrieved from http://dx.doi.org/10.1016/j.infsof.2013.08.002 doi: 10.1016/j.infsof.2013.08.002