An Improved Approach for Extracting Frequently Extracted Code Idioms

Md Borhan Uddin

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 2023

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Md Borhan Uddin**

Entitled: **An Improved Approach for Extracting Frequently Extracted 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. Jinqiu Yang

______________________________ Examiner
Dr. Ian Shang

______________________________ Supervisor
Dr. Nikolaos Tsantalis

Approved by ______________________________
Dr. Leila Kosseim, Graduate Program Director

____________ 2023

____
Dr. Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

An Improved Approach for Extracting Frequently Extracted Code Idioms

Md Borhan Uddin

Source code refactoring is a process of restructuring or changing the existing codes without changing their external behaviour. This is a continuous process done by the developers to improve code quality, readability, maintainability of the source code, and address technical debt. There have been studies and tools to aid developers to refactor effectively their source code and to understand the motivations behind refactorings applied by developers. We aim to find Code Idioms that developers tend to refactor more frequently and investigate whether there are unique refactored code idioms for production code and test code. We use the RefactoringMiner tool to detect and collect EXTRACT METHOD refactoring from the commit history of the projects and propose a technique to represent the code fragments as structure-preserving context-free independent graphs and apply graph-similarity measure techniques to find similar code idioms among 65,742 EXTRACT METHOD instances. We measure both exact matching and partial matching with constraint checking from the associated metadata of the nodes and edges of the graphs. We divide our data set into production code and test code and found a total of 489 code idiom patterns. We present in detail 22 of the most frequently refactored code idioms. There are unique patterns to production code and test code and patterns shared among them. We limit our study to only Java-based open-source projects and EXTRACT METHOD refactoring, but we believe the approach can be applied to other object-oriented languages or refactorings. The findings can be useful to design an effective refactoring recommender system, help developers gain confidence in refactoring recommendation tools, and help researchers understand refactoring motivations and API usage patterns.

Acknowledgments

I would like to take this opportunity to express my deepest gratitude to my supervisor Dr Nikolaos Tsantalis for his guidance, support, and expertise throughout my graduate journey. His invaluable feedback and insightful comments have played a crucial role in shaping my thesis into its final form and fulfilling my academic requirements. I am grateful for the knowledge and skills he has imparted to me.

I would like to extend my heartfelt appreciation to my family, especially my wife, for her understanding and support throughout my years of study. The encouragement and patience have been instrumental in keeping me motivated during challenging times.

I would also like to extend my appreciation to my lab mates for their valuable thoughts, insights and encouragement to keep up the quality of my thesis work. It was my privilege to work with the team.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 What is Refactoring

Refactoring is the process of improving the internal structure or design of existing software code without changing its external behaviour [2]. It involves making changes to the codebase to improve its quality, readability, maintainability, and performance while preserving its functionality [3]. The main objective of refactoring is to reduce technical debt, which is the cost of maintaining software that arises from poor code quality, architecture, or design decisions [4]. Refactoring helps to keep codebases maintainable and easy to work with over time, even as the codebase grows in size and complexity [5], easier to tune for performance, test and debug or extend for new features [6]. The activity around refactoring involves locating the code fragment that could benefit from refactoring, determining the type of refactoring that can be applied to preserve the behaviour of the program, and applying and assessing the effect of the refactoring on design quality [3].

1.2 What motivates to Refactor source code?

According to the book titled *Refactoring Improving the Design of Existing Code* by Martin Fowler [6], refactoring can involve a wide range of source code transformations, such as renaming variables or functions to improve their clarity, splitting large functions into smaller ones to improve their readability and modularity, and removing duplicated code to improve maintainability and reduce the risk of errors. Refactoring is an essential practice in software development, as it helps to ensure that codebases remain maintainable and adaptable over time [6]. It is often performed as part of a larger software development process, such as agile development, and is usually done incrementally to avoid introducing bugs or breaking existing functionality [7].

To understand what motivates developers to refactor Silva et al. [8] conducted a large-scale study by interviewing developers right after applying refactoring edits in open-source projects hosted on GitHub, and found that refactoring is mainly driven by changes in requirements (i.e., bug fixes and feature requests) rather than code smell resolution, as most researchers believed. The study also found that EXTRACT METHOD refactoring has 11 different motivations and only two of them are driven by resolving a code smell. Another study shows that developers tend to do refactoring as part of a bug fix or feature implementation [9] instead of dedicating long time periods on focusing refactoring activity. While simple refactoring such as renaming variables is a common practice, complex type refactoring, such as extracting methods, is typically only performed when developers need to address a specific issue or complete a particular task that necessitates such changes [10]. Weißgerber and Diehl [10] studied three open-source projects and found that there were no single-day developers who applied only refactoring changes. Though, refactoring changes are less susceptible towards introducing bugs whereas a feature implementation is more prone to introducing a bug [11], Murphy-Hill and Black [9] show that developers avoid refactoring-only changes and rather wait until a bug fix is required or another feature needs to be implemented around the code they intended to refactor.

1.3 What prevents the use of Refactoring tools?

Murphy-Hill et al. [9] pointed few qualities of a good refactoring tool, such as being able to choose the desired code to be refactored quickly, switching seamlessly between program editing and refactoring, viewing and navigating the code while using the tool, avoiding providing explicit configuration information and access to other tools available in the development environment. Unfortunately, most of the tools do not qualify for these. Developers complain about difficulties while using the tools to perform certain refactorings. The tools are either difficult to use or require additional effort to work with. In spite of these problems, Silva et al. [8] found that 71% of the interviewed IntelliJ IDEA users were using the IDE refactoring automation tools, while 50% or less of the interviewed Netbeans and Eclipse users were using the IDE refactoring automation tools.

Negara et al. [12] shows that developers often use tools to perform small refactoring but for more complex refactoring they opt for a manual process. For example, EXTRACT METHOD refactoring is predominantly performed manually despite the size of the code change. According to Murphy-Hill et al. [9], developers may not be satisfied with the way the tool performs the EXTRACT METHOD refactoring. Silva et al. [8] identified a few reasons for developers opting for manual refactoring, with the top reasons being that developers do not trust automated support for complex refactoring types, developers believe automatic refactoring is unnecessary for a trivial refactoring type, the IDE does not support the required modification, and developers are not familiar with the capability of the tool. So, it is important to have tools which can make a reliable and relevant refactoring recommendation and are easy to use.

1.4 Problem Definition

To address the problem of tools suggesting reliable and relevant refactoring recommendations, it is important to understand how developers perform a specific type of refactoring. There have been numerous studies to understand the repetitiveness of code changes [13] in software evolution, mining object usage patterns in source code [1], mining code idioms from a predefined corpus [14], studies to identify the most common refactorings [3] or how refactoring can improve code quality or reusability [15]. None of the studies investigated if there are specific code fragment patterns that developers tend to refactor most commonly. Such patterns can be viewed as *Code Idioms* that are more likely to be refactored by developers. A relevant study from Allamanis et al. [16] mining semantic loop idioms focuses on finding loop idioms in source code. There is a thesis [17] that tried to identify this kind of code idioms and reported some useful patterns. We focus on extending and improving the methodology used in [17] to create a more comprehensive list of code idioms that developers tend to refactor more frequently.

1.5 Scope of our Study

According to the catalogue of refactorings by Martin Fowler [18], there are 66 different types of refactoring that can be performed. Here, in our study, we limit our focus on the EXTRACT METHOD refactoring type. EXTRACTED METHOD is one of the most frequently performed refactoring operations developers perform and there are 11 identified purposes behind its application [8]. EXTRACT METHOD is a refactoring technique that involves taking a piece of code within a larger method and turning it into a separate method [6]. The extracted method can then be called from the original method, reducing duplication and improving the overall readability and maintainability of the code [19]. The process of extracting a method [20] involves identifying a coherent block of code within a larger method that performs a specific task. This block of code is then extracted into a new method, and the new method is given a descriptive name that indicates its purpose. By breaking down large methods into smaller, more focused methods, it becomes easier to understand and reuse the code, making EXTRACT METHOD refactoring a valuable tool for developers. The benefits of using the EXTRACT METHOD refactoring technique include increased reusability, improved readability simplified testing and enhanced maintainability.

We also limit our study to only Java-based projects hosted as GitHub repositories. The reason is that the tool we use for mining refactoring instances from repositories, RefactoringMiner [21], supports only Java code.

1.6 Contribution

Despite having a wide range of studies related to refactoring in source code and different aspects of refactoring, as well as studies mining code idioms, there is no study investigating refactored code idioms in source code. The previous study titled *Frequently Refactored Code Idioms* [17] was the first work of this kind. We extend this work by improving the methodology and tooling to identify refactored code idioms in source code. We speculate that there are code fragment patterns, which are more commonly refactored across the source code repositories and can be defined as code idioms. Identifying these code idioms will increase our knowledge about how developers practice refactoring. This knowledge would also help researchers to design more effective EXTRACT METHOD recommendation systems for the tool users, by prioritising the recommendations based on the popularity and relevance of particular code idioms. Our findings of frequent code idioms can be useful to show that certain API usage patterns tend to be placed in separate methods by developers. Patterns of API calls grouped together to perform a specific task in a separate function can also help developers to understand the use case of an API or a group of APIs.

We have mined the commit history of 110 Java source code repositories and collected a large set of EXTRACT METHOD refactoring instances for this purpose. We used the Abstract Syntax Tree (AST) representation of the methods to generate the Control Flow Graph [22] and Program Dependence Graph [23]. We have used a graph-based representation GROUM [1] to eliminate context-specific information and preserve the basic semantic structure of the code. We have extended the original GROUM implementation and required abstraction to make it more suitable for our purpose. We have used a graph similarity measure technique to relate each graph with one another. Graph pattern recognition techniques are computationally expensive and to make them computationally viable for a large set of graph data we have devised our graphs as sequences and applied algorithms to find best-matched sequences. To find the sequences we used exact matching, subsequential matching and substring type matching. From our findings, we have presented the 22 most frequent and interesting code idioms. We considered EXTRACT METHOD refactorings in both production code and test code in our experiment. The frequently refactored code idioms include different object creation patterns, collection usage patterns, and exception handling patterns used across the examined Java projects. We have used a total of 65,742 EXTRACT METHOD refactoring instances from 31,427 commits.

1.7 Thesis Structure

The rest of the thesis is organised as follows: Chapter 2 has the required background, Chapter 3 has a brief discussion on related works of different aspects of our study, Chapter 5 and Chapter 4 explains our experimental methodology and implementation, Chapter 6 explains our findings and conclusions.

Chapter 2

Background

The generation of a graph-based representation from source code without building the program can play an important role in software engineering research. The Program Dependency Graph [23], a graph-based representation, plays an essential role in source code analysis, and source code optimization, as it helps us to understand the control structure as well as the data dependencies. GROUM [1], another graph representation, is very useful for recognizing patterns with control and conditional structure that can be helpful for anomaly detection and source code pattern recognition.

The Control Flow Graph (CFG) [22] is a directed graph representation of a program in which the nodes represent the basic blocks and the edges represent control flow paths in a computer program. It models the sequence of statements executed in a program and the decision points where control may branch to different parts of the code. In a CFG, each node represents a statement in the program, and edges represent the flow of control between statements. The edges in a CFG are typically directed, indicating the order in which statements are executed, and the nodes may contain information about the type of statement, the variables used, and any control decisions made. CFGs are useful for visualizing the structure of a program and for analyzing its behaviour, such as detecting errors, measuring performance, and optimizing the code. CFGs are also used as a basis for other software development tools, such as program slicing, which analyzes the relationships between statements and variables in a program. Additionally, CFGs play an important role in compiler design, where they are used to generate code for target architectures. Overall, CFGs provide a simple and intuitive representation of a program's behaviour and are widely used in software engineering and programming language research. Any static, global analysis of the expression and data relationships in a program requires a knowledge of the control flow of the program. Control Flow Graphs can accurately represent the flow inside of a program unit and are able to point out the codes which are unreachable.

The program dependence graph (PDG) is a graphical representation of the relationships between statements in a computer program. It provides a compact representation of the control and data flow of a program and is used to analyze the behaviour of software systems. A PDG is constructed by modelling the dependencies between statements in a program and capturing the relationships in a graph structure. Each node in the graph represents a statement in the program and edges represent the flow of control or data between statements. The use of PDGs is particularly helpful in detecting errors in a program, such as dead code, uninitialized variables, and memory leaks, as well as for optimizing program performance. It can also be used for program transformations and for visualizing program behaviour. PDGs are widely used in compilers, program analysis tools, and other software development tools to provide a deeper understanding of software systems.

Listing 1 Running example Code snippet

```java
private static void method() {
    StringBuffer strbuf = new StringBuffer();
    BufferedReader in = new BufferedReader(new FileReader(""));

    String str;

    while((str = in.readLine()) != null ) {
        strbuf.append(str + "\n");
    }

    if(strbuf.length() > 0) {
        outputMessage(strbuf.toString());
    }

    in.close();
}
```

Listing 1 shows a code snippet of a method implementation. Figure 2.1 represents the program dependence graph. For simplicity, we used the line numbers to represent the node IDs.

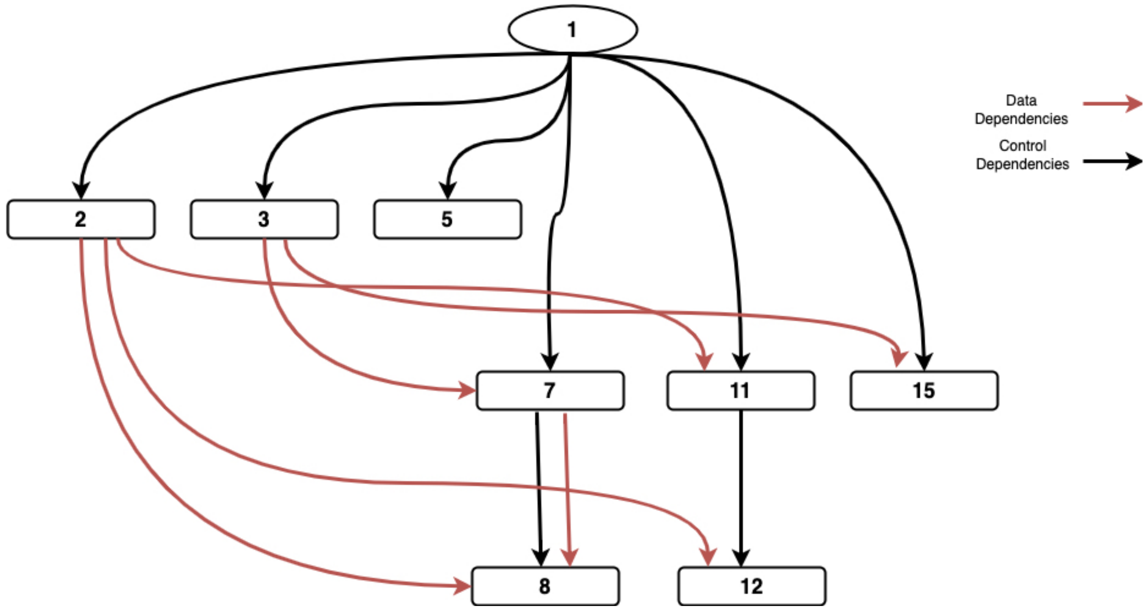


Figure 2.1: PDG Representation (using line numbers to represent nodes)

GROUM is a direct acyclic graph where nodes can be of two types (i.e., action node and control node). Action node represents an invocation of a constructor or a method or access to a field of one object. A control node represents the branching point of a control structure. The label of the node can be represented as `C.m` where `C` will be the class name and `m` will be the method or field name. The edges among the nodes will represent either temporal usage order or a data dependency. Edges do not have any labels. Figure 2.2 shows the GROUM representation of that code snippet (Listing 1). The figure includes various elements such as large rectangles that represent action nodes and small rectangles that contain variable names associated with that action node. Control nodes are indicated by decision symbols and are grouped together with the nodes around them in a large dashed rectangle known as the control block area. The background colour of the action nodes is used to indicate nodes that are related to a specific variable.

In a computer program, an abstract syntax tree (AST) is a tree representation of its abstract syntactic structure. Therefore it provides a high-level view of the code, abstracting away from the details of the programming language's syntax and focusing on the underlying logic and structure of

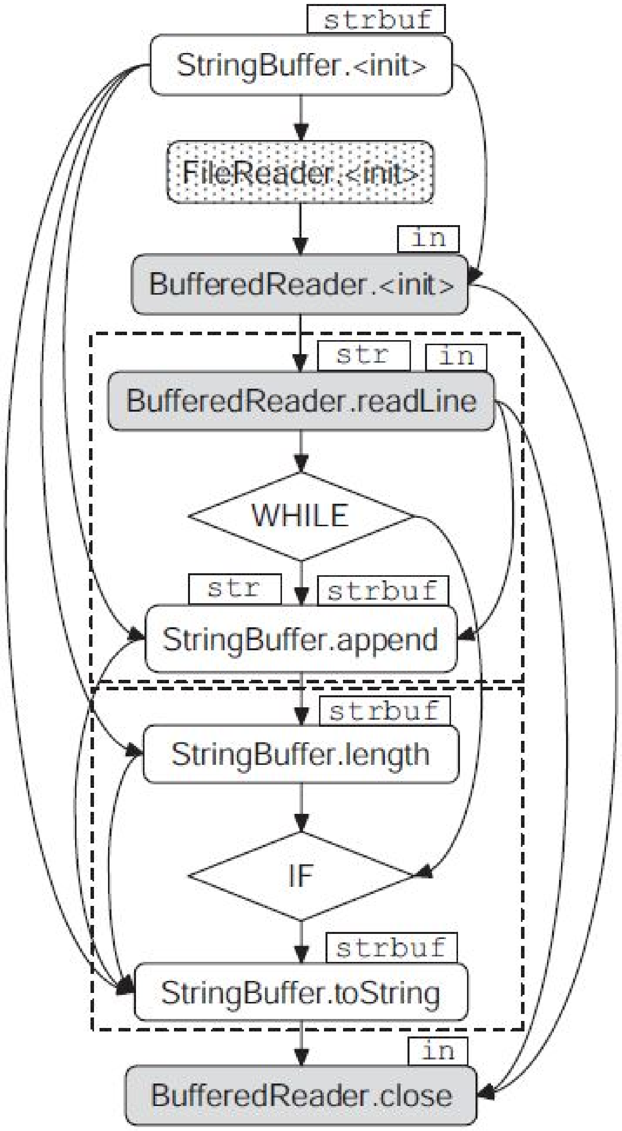


Figure 2.2: GROUM Representation (Image taken from [1])

the program. In an AST, each node represents an operation or a construct in the program, such as a variable declaration, a loop, or a method invocation or class instantiation. The edges in the AST tree represent the relationships between operations and constructs, such as the arguments of a function call or the statements in a loop. ASTs are used for various purposes in software development, including code analysis, code generation, and code optimization. For example, compilers use ASTs to analyze the syntax and semantics of a program, detect errors, and generate code for a target architecture. ASTs are also used in code editors and IDEs for syntax highlighting, code completion, and other features. Overall, the use of ASTs provides a convenient and flexible way to analyze and manipulate computer programs and is a fundamental concept in compiler design and software engineering. We use an abstract syntax tree when we generate the control flow graph, which in turn is used to generate the program dependence graph. During the generation of the GROUM structure, we also use abstract syntax tree representation to capture the node types.

Chapter 3

Related works

The mining of code idioms and frequent patterns in software engineering has been a topic of interest for researchers in recent years. Idioms are patterns of code that are frequently used to solve a particular problem or to implement a particular feature. They are often refactored frequently by developers to improve code quality and maintainability. Graph patterns, on the other hand, are structures that represent relationships between objects and actions in the code fragment, such as class objects, the method used, and variables. Frequent graph pattern mining can reveal important insights into the underlying structure and organization of software, as well as identify areas for improvement. We have looked into several aspects and steps of our study and studied related works on those aspects. We organised related works into three sub-sections: Mining idioms from source code, Partial program analysis techniques, and Techniques for pattern mining from the graph structure. The papers in these sub-sections cover a range of techniques for mining idioms, frequent graph patterns, and code refactoring. These techniques involve a variety of data mining algorithms and representation of source code, as well as approaches for code analysis and optimization.

3.1 Mining Idioms

A code idiom is a common and recognizable pattern or expression that is used to accomplish a particular programming task. There have been a handful of works related to understanding and mining code idioms from open-source projects. Allamanis, Barr et al. [16] conducted a large-scale study with a 277K loop corpus and proposed a specialised technique for mining loop idioms. The study represented the idioms as a tree substitution grammar, an extension of context-free grammar. The study shows that `foreach` loops are most popular followed by the `for` and `while`. Another work by Allamanis and Sutton [14] employed a language-agnostic non-parametric Bayesian tree substitution grammar developed for natural language to mine interesting code idioms from an idiomatic corpus. The paper presents a tool that automatically mines code idioms from the given corpus and it can identify interesting program concept which includes object creation, exception handling and resource management. Language modelling and statistical model-based technique were applied for enabling massive-scale mining of source code repositories [24]. This paper shows that training models with large-scale training data can increase their predictive capabilities.

The API developers often do not document the API usage pattern or ordering rule when they develop their APIs. Acharya et al. [25] presented a framework to automatically extract usage scenarios as partial orders from the static API traces. That is a novel application of a miner to mine such API usage order and specification. In software systems, code clones are similar program structures recurring in different forms. Basit et al. [26] propose a data mining technique to detect higher-level similar structural clones. The process starts with finding smaller clones and incrementally higher-level similarity. Code similarities can be learned from a diverse representation such as identifiers, CFGs, ASTs and bytecode of the code. Tufanu et al. [27] evaluated the performance of such representations and showed that these are complimentary to each other. The paper also shows that relying on multiple representations with a combined approach can be more effective and accurate to detect code clones. Livshits and Zimmermann proposed a tool named DyneMine [28] to discover usage patterns and their violation in large software systems. It is a data mining strategy for large software systems to detect common usage patterns by analysing software revision histories. The mining strategy focuses on incremental changes between the revisions. This is an example of incremental analysis, where the analysis is performed only on the parts of the program that have changed since the last analysis.

3.2 Partial Program Analysis

Partial program analysis is a technique used in program analysis that aims to analyze only a portion of a program rather than the entire program. Partial program analysis typically involves selecting a subset of the program, such as a module or function, and analyzing it in isolation from the rest of the program. This technique is particularly useful when dealing with large software systems where analyzing the entire system can be time-consuming and resource-intensive. It is even more difficult when a program analysis tool requires to compile the project in order to resolve binding information of the certain object. Tufanu et al. [29] shows that on average only 38% of the revision history of the analyzed systems is successfully compilable, and broken snapshot occurs in 96% of the analysed projects in the study. The main problem behind this is related to the dependency resolution of the revision. Dagenais et al. [30] proposed a technique to produce a complete and typed intermediate representation for the source code of the partial Java program. Java compiler creates the intermediate representation based on type declarations, but this is not possible when the tool does not have access to the complete program or the complete dependencies of the program. The technique recovers the declared types by performing partial type inference.

One example of partial program analysis is dynamic program slicing, where a slice of a program is created based on a specific input or output. This approach has been used in a variety of applications, such as fault localization and program debugging. Horwitz et al. [31] propose a dynamic slicing technique to create a representation of a system dependence graph from a program dependence graph. A system dependence graph is an extension of a program dependence graph and the slice consists of all statements and predicates of the program that might affect the value of a certain variable. It is challenging to resolve unknown types, fields, methods and variables, and also to determine whether these unknowns are resolved sufficiently. To address these challenges Zhong et al. [32] proposed an approach called Graphs for the partial program (GRAPA) to boost tools to analyze partial programs. This approach also takes advantage of system dependency graphs. It identifies a compiled release of the complete program whose version is closest to the partial program by taking information from the complete program to resolve unknown code names. The limitation of this work is that it depends on a release version of the program which successfully compiles and hence depends on the compilation of the program. This is different from our case where we have to work with each commit of the source repository and compiling each of those could be a real challenge and at some level impossible.

Considering the compilation constraint of the project, in our work, we considered an approach which does not require compiling the complete program and tries to resolve binding information of types, methods, fields and variables based on the external dependencies of a particular commit revision of a source repository, as specified in build files (i.e., Maven or Gradle scripts) [33].

3.3 Graph Pattern Mining

Finding frequently refactored code idioms can be challenging for several reasons. Two of the major challenges are finding a suitable representation and a technique to find similarities among the representations. A representation should be free of context-specific information, require higher level abstraction and at the same time have enough information to preserve the structural execution flow of the code. For these reasons, we choose a graph representation called GROUM [1] with some additional enhancement, deviation and abstraction. In the process of GROUM generation, we generate intermediate representation as CFG and PDG and we take influence from JDeodorant [34] to verify that our CFG and PDG generation from the partial source code is effective and accurate.

Now that we have a representation it is challenging to find a suitable technique to find the similarities among them. Graph Isomorphism may be an obvious choice for that. However, subgraph isomorphism can be a computationally intensive task, as there is no known polynomial time algorithm for this [35]. There is a handful of research to improve the performance of the isomorphism algorithm and to reduce the computation time and memory space. A brute-force approach for a graph isomorphism algorithm results in a depth-first search. The most well-known algorithm is Ullmann's [36] algorithm for subgraph isomorphism. This algorithm reduces the search space by applying backtracking. The time complexity of this algorithm at best is $O(N^3)$. We presume that the graph sizes for extracted methods may not be very large, but we have a large number of graphs to compare with. Also, we may need need to be flexible with the matching approach to finding more appropriate similarities.

There has been a considerable number of research efforts to find similarities among graphs. Several techniques have been developed to identify common patterns in metadata graphs, such as different clustering, the use of graph-based statistical language models and sequence-based graph similarity measures. Reza et al. [37] propose a technique based on constraint checking. The approach views a search template with a set of specific constraints which must be respected by the vertex and edges of the graph. The approach iterates over the constraints to eliminate or accept vertices and edges. This iteratively prunes the original graph and reduces it to a subgraph which is the union of all matched vertices. Chen et al. [38] proposed a technique for centroid-based clustering for similar graphs. The clustering is done based on structural dissimilarity and code correspondence between graphs. The graphs are represented as vectors by space transformations and the mean is calculated as a centroid. Eventually, this is an application of k-means clustering. Clustering consists of partitioning a graph set into groups of graphs that have high similarity. There is a difficulty in the clustering approach to determine the number of clusters, which means the value of k. Graph-based k-means clustering [39] tries to address this problem by defining an approach to determine the number of clusters and an efficient centroid value. The method relies on the properties of a minimal spanning tree (MST) by Prim's algorithm. The algorithm is based on data-driven hierarchical classification and can be applied to a variety of distance metrics or similarity measures in feature space. GraLan [40] is a graph-based statistical model which learns from a corpus of graphs to compute the probability that a graph would be observed in a given set of graph contexts. ASTLan an AST-based language model is developed based on GraLan to detect a common syntactic template. These are based on the idea that in a usage pattern, it is not always required to preserve the execution order between two APIs. The empirical evaluation on a large corpus using these tools shows high accuracy in suggesting API and common templates.

Based on the studies of graph pattern mining, we endeavour our effort to define a search topology so that we can apply an approximate matching technique with constraint consideration. Since the GROUM representation is a directed acyclic graph, we try to convert our graphs as a sequence of nodes with connecting edges, where each node and edge has meta information to be used as a constraint while matching. Thus, we were able to apply the algorithm to find common subsequence and substring type patterns. Our implementation section provides more details about our approach.

Chapter 4

Research Methodology

Our approach takes as input a list of refactored code fragments and applies two main steps. In the first step, we generate for each refactored code fragment a graph structure representing the control and data flow between the statements of the code fragment, along with metadata regarding external API method calls (i.e., calls involving third-party libraries and the standard Java libraries available in JDK). In the second step, we apply an algorithm that we designed to compare and group the closest matching graphs based on their computed similarity values.

Our approach is implemented in a project named RefactoringMatcher [41], which is open-source and available on GitHub. Figure 4.1 summarises the steps of our approach.

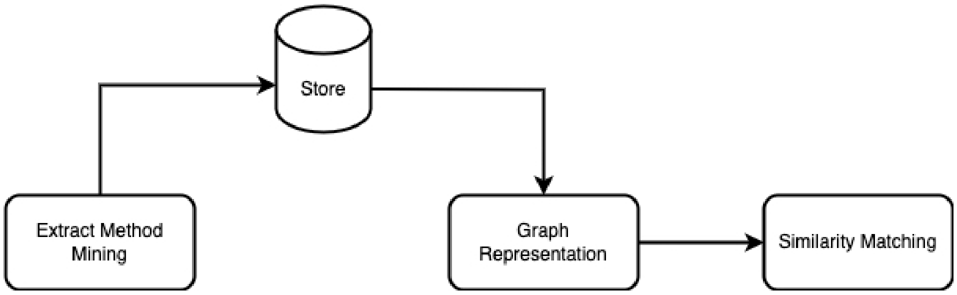


Figure 4.1: Main steps of our approach

4.1 Graph Generation

Figure 4.2 summarises the steps required to generate a graph for each refactored code fragment. The input is the raw source code fragment corresponding to an extracted method, which is recovered based on the location of the method in its source file, as provided by the refactoring mining process (more details are given in Section 5.1). This process involves, generating the control-flow graph (CFG) of the code fragment, followed by generating the program-dependence graph (PDG) of the code fragment. Next, the PDG is enhanced with binding information, by recovering the method and field declarations corresponding to method calls and field accesses within the code fragment. This information is necessary to generate the GROUM graph representation, as the GROUM graph includes action nodes corresponding to external (i.e., library) method calls, class instantiations, and field accesses. Therefore, it is very important to know whether a method call, class instantiation, or field access is internal or external, and if it is external to which API type it corresponds. Once the GROUM graph is generated, we further optimize it to a form that is more suitable for our pattern-matching approach.

4.1.1 Program Dependence Graph

The program dependence graph (PDG) is a representation of the control and data dependencies (i.e., graph edges) between the statements (i.e., graph nodes) within the body of the extracted method. There can be multiple edges between two nodes if the nodes have multiple relationships, such as direct control dependency and also data dependency.

First, we retrieve the EXTRACTMETHOD information from the Neo4j database for each commit of a project. Then, with the help of a utility function provided by the API Finder tool [33], we recover the dependencies (jar list) of the project for the specific commit in which the EXTRACT METHOD refactoring is found. Next, we generate the AST for the entire Java file containing the extracted method using the Eclipse JDT AST Parser. From the AST, we retrieve the method declaration corresponding to the extracted method and the field declarations existing in the class containing the extracted method, and we generate our own internal Method object representation. The Method object stores the statements within the body of the extracted method using the Composite

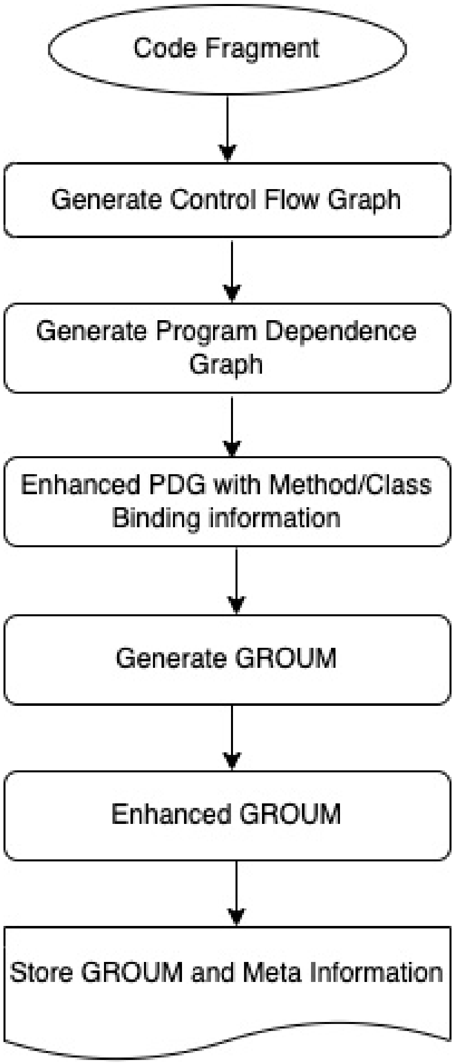

Code Fragment
Generate Control Flow Graph
Generate Program Dependence Graph
Enhanced PDG with Method/Class Binding information
Generate GROUM
Enhanced GROUM
Store GROUM and Meta Information

Figure 4.2: Graph Representation Steps

design pattern [42], i.e., composite statements are the statements that include other statements in their body, and leaf statements are the statements that do not have a body. Moreover, the Method object includes information, such as the parameter list, return type, access modifier of the method, list of local variables and their types, list of method invocations inside the method, list of class instantiations, list of field accesses, and the fully qualified name of the class containing the extracted method. Based on the method object, we generate the control flow graph where every node is a statement and every edge indicates which statement will be executed next. The control flow of the edges is marked as true control flow, false control flow and loopback flow. A true control flow directing to a statement means that it will be executed next if the condition of the previous branch block is evaluated as true. A false control flow directing to a statement means that it will be executed next if the condition of the previous branch block is evaluated as false (i.e., the else branch of an if statement is reached, or the loop termination condition is reached). A loopback flow directed from a statement means that it was the last statement executed in a loop. Loopback flow is important to determine loop-carried data dependencies between statements within a loop.

Once we have the control flow graph generated, we use it to compute the data dependency information for the program dependency graph. The control dependencies in the program dependency graph are created based on the nesting structure of the code, i.e., all composite statements have a direct control dependency to the statements nested within their body. To generate the data dependencies between the nodes (i.e., statements) of the program dependence graph, we examine the variables being declared, assigned and used in each statement and apply the following rule for all pairs of statements: A data dependency edge from node p to node q due to variable x exists if and only if

- statement p defines variable x,
- statement q uses variable x, and
- there exists a control flow path from statement p to q without an intervening definition of x.

4.1.2 Enhanced PDG

In this Section, we explain how we enhance the Program Dependence Graph (PDG) with binding information for the method invocations, class instantiations, and field accesses within the statements of the code fragment corresponding to the PDG nodes. Binding information is necessary to distinguish between internal and external method invocations, class instantiations, and field accesses, as the GROUM graph representation includes action nodes corresponding only to external APIs.

4.1.2.1 Resolve Binding Information

When generating a GROUM graph, it is required to identify binding information for the invoked methods, created objects, and accessed fields, including package and class type, in which they are declared. Typically, such information is readily available when we build and compile the source code before executing it. However, there are many obstacles to successfully building commits. Tufano et al. [29] found that only 38% of the change history of software systems can be successfully compiled. Therefore, we must find an alternative way to resolve and obtain binding information.

To illustrate, when encountering a statement that invokes a method `m()`, we require information regarding the type `T` declaring this method and its corresponding package name. Additionally, we need to determine if a field access, method invocation, or class instantiation belongs to the local project or an external library/package. To obtain this information without building the source project, we have utilized the API Finder tool [33]. The API we are using takes parameters such as the dependent jar list of the project, the Java version and `MethodInvocation` or `ClassInstanceCreation` or `FieldAccess` Eclipse JDT Parser AST node. This API provides essential information such as the fully qualified name, class type, and package name for a given AST node. It can determine whether the fragment is internal to the project or from an external library or project. Additionally, the API Finder library has an API that generates a list of dependent JARs for any commit revision of a project, supporting both Maven and Gradle build system-dependent projects. Hence, the ability to use the API Finder tool was one of the selection criteria for the projects in our experiment.

4.1.2.2 Handle unresolved cases

API Finder can resolve the binding information for most cases. There are cases for which it cannot resolve the binding information, such as

(1) When the Java version of the project is not supported by the API Finder or the JDK related to that Java version is not available in the system.

(2) When the project is not using either the Maven or Gradle build system.

(3) When API Finder cannot get the dependent jar of the MethodInvocation or ClassInstanceCreation from Maven central or Gradle repository properties.

(4) When the `MethodInvocation` or `ClassInstanceCreation` is part of a call chain and some API Finder fails to resolve bindings for some parts of the call chain.

We have implemented a fallback mechanism for such cases when API Finder fails. The fallback mechanism is implemented in a way to find the minimum required information, such as the object type and relevant package it belongs to. In our mechanism, we try to find if the object is declared in the extracted code fragment, passed as a parameter, or returned by another method invocation or class instantiation, for which we already resolved the binding information. We can also infer the type of field accesses from their corresponding field declaration within the class the extracted method belongs to.

4.1.3 GROUM Generation

The Program Dependence Graph (PDG) could have been used to check for graph isomorphism. There are other works that suggest using PDG to find graph isomorphism or code similarities. However, that would require having exactly similar statements between two code fragments and similar incoming and outgoing dependencies for each statement. It would be very context-specific to the code. Moreover, it would be very rare to have two exactly similar code snippets in different projects. By removing context-specific information (e.g., variable names), we can have similar incoming and outgoing dependencies for different code fragments that can be matched by graph isomorphism algorithms. We can further abstract the statements by extracting more granular-level nodes, such as method invocations, class instantiations, field access, conditional statements, loop, try-catch nodes, throw exceptions, the return statement and form a graph representation.

To generate the GROUM graph representation, for each PDG node we identify method invocations, class object creations, field accesses, and control/conditional keywords, and create a GROUM node along with its meta information, such as associated variable information, whether it is an internal or external class object creation or method invocation, its fully qualified name, and the nested block to which it belongs.

We classify the GROUM nodes into two types: action nodes and control nodes. Action nodes include object creation, method invocation, field access, return statement, throw statement, and try block, while control nodes include branching nodes like loops, if-else statements, and switch statements. Our GROUM generation approach extends the original implementation to better serve our purpose. We label all nodes in a way that removes context-specific information, making them more abstract and generic. For example, if a `java.lang.String` variable named `str` is present and method `length()` is called, we label the node as `javalang.length()`. Here, we loosen up the concrete class type and variable name to make it more abstract and generic. We simplify the labelling process by considering the first two parts of the sub-package section for any fully qualified name starting with `java` or `javax` and the first three parts for non-JDK libraries. For instance, `java.lang.String` becomes `javalang`, while `org.apache.commons.net` becomes `orgapachecommons`. Additionally, if a method is invoked consecutively multiple times, we consider it as a single method invocation action node.

The first step in constructing the GROUM graph is to create a preliminary version consisting only of control edges named temporal GROUM. Variable-associated edges are added later. These edges are labelled to accurately capture the relationships between nodes. There are three labels: `sink-control`, `variable-associated`, and `nesting`. Two nodes have a sink-control edge if node `A` is generated before node `B` and the execution control flows from `A` to `B`. Two nodes have a variable-associated edge if there is a related variable that is either defined or used, `A` is generated before `B`, and the two nodes are associated with that variable. A `nested` labelled edge indicates that node `B` belongs to the branch block of node `A`. The edges are directed and the graph is acyclic.

To generate the GROUM representation for a given code fragment, we create a separate GROUM for each statement in the code. If a statement contains a complex expression, we also generate a separate GROUM for that expression. Once we have generated the GROUMs for all the statements and expressions in the code, we merge them together using two merging techniques: sequential merging and parallel merging. Sequential merging between two nodes or two GROUM `A` and `B` simply denotes that `A` gets executed before `B` and control flows from `A` to `B` or there is a dependency relationship between the two nodes or GROUM. In the case of parallel merging, two nodes or GROUM blocks are merged together when they are not dependent on each other and can be executed independently. This happens when there is a branch in the control flow and either branch can be executed. In either case, sink nodes (nodes with no outgoing edge) connect with the starting nodes (nodes with no incoming edge) via a sink-control edge type.

After generating the temporal GROUM graph, we establish a variable-associated edge between nodes that share a common variable usage, meaning the variable is either defined or used by both nodes. These edges are only created between nodes that are sequential in the control flow, not between nodes that are merged in parallel. Figure 2.2 shows the complete GROUM graph for the running code example for Listing 1.

4.1.4 Enhanced GROUM structure

Our initial implementation of GROUM graph generation resulted in a large number of unrelated code idioms grouped together, i.e., there was some “noise” within our expected result. Due to this “noise”, it was difficult to manually verify our matching algorithm properly. Also, it was difficult to find similar code idioms for which the code structure and purpose are aligned. Therefore, we decided to make some changes to the original GROUM graph to ensure we can avoid such “noise”. We have made the following adjustments to our GROUM generation process.

(1) Change the node labelling convention for Collections and Exceptions

(2) Include additional types of action and control nodes

(3) Label the edges to be able to preserve the original code structure

(4) Avoid the generation of variable-associated edges to make two structurally different codes have a similar graph representation.

(5) Simplify Builder and Stream coding style graph generation to identify builder and stream patterns

We consider these deviations as our contribution to the original GROUM implementation. Our experiment shows better results in finding code idioms with a similar purpose. We shall briefly explain each of these changes.

4.1.4.1 Change the Node labelling convention

A concrete node label would make the graph more specific to that particular code fragment. During the pattern matching, the algorithm would look for an exact match for a method invocation or class object creation. Consider two code examples from listing 2 from pentaho-kettle and 3 from apache-zookeeper.

Listing 2 Naming Convention Example 1

```java
private static Set<String> getUsedStepsForReferencendTransformation( TransMeta transMeta ) {
    Set<String> usedStepNames = new HashSet<String>();
    for ( StepMeta currentStep : transMeta.getUsedSteps() ) {
      usedStepNames.add( currentStep.getName().toUpperCase() );
    }
    return usedStepNames;
}
```

Listing 3 Naming Convention Example 2

```java
private MultiTransactionRecord generateMultiTransaction(Iterable<Op> ops) {
    // reconstructing transaction with the chroot prefix
    List<Op> transaction = new ArrayList<Op>();
    for (Op op : ops) {
        transaction.add(withRootPrefix(op));
    }
    return new MultiTransactionRecord(transaction);
}
```

Both methods are taking input parameters, creating an object, iterating over an iterable item, adding elements to a newly created collection and finally returning it. Here both methods are serving the same purpose, but one is working with `HashSet` and the other with `List` type. Although the methods use two different data structures, their actual purpose is the same. So, if we label the action nodes as `ClassType.add()` then these two codes would not match as a similar pattern. However, if we consider the type as the more generic `Collections` instead of the concrete Class type, then we would get these two code fragments matched, as expected. Similarly, for any type of `Exception` thrown, we name it simply as `Exceptions` to group all types of exceptions into a single type and make them less context-specific and more purposeful. There are many different types of exceptions thrown in Java code, but the purpose of Exception handling remains the same. Let's consider the following two Listing 4 and 5 code examples.

Listing 4 Exception Handling Example

```java
public void SampleMethod()
{
    try {
        String a = null; //null value
        System.out.println(a.charAt(0));
    } catch(NullPointerException e) {
        System.out.println("NullPointerException..");
    }
}
```

Listing 5 Exception Handling Example

```java
public void SampleMethod()
{
    try {
        String a = "This is like chipping "; // length is 22
        char c = a.charAt(24); // accessing 25th element
        System.out.println(c);
    }
    catch(StringIndexOutOfBoundsException e) {
        System.out.println("StringIndexOutOfBoundsException");
    }
}
```

Even though the example codes are a little bit different, their actual purpose is similar, despite the use of different exception types. So we consider them as part of the same code idiom. Similarly, for all types of loop actions such as `for`, `enhanced-for`, `while` and `do-while` we name the node as `LOOP` so that it allows us to identify the generic action of loop-back action.

4.1.4.2 Include additional types of action and control nodes

We have added additional node types as action and control nodes to better capture the code structure. Action nodes, such as `try-catch` statement, `return` statement and `throw` statement are included as they are important to understand the code control flow. Similarly, the `switch` block and enhanced-for (also known as for-each) loop are also considered. These additional nodes give better similarity and relevant results when finding code idioms.

4.1.4.3 Edge labelling to preserve original code structure

During our experiments, we found cases where identical GROUM graphs with nodes having exactly the same labels and edges connecting these nodes. However, the corresponding code fragments had differences in their functionality. We realized that this problem occurs because the edges are unlabelled. Listings 6 and 7 demonstrate the unlabelled edge problem.

Listing 6 Unlabelled Edge Example 1

```java
public String SampleMethod()
{
    ....
    if (str.length() > 0) {
        // Do something
        return str;
    }
    ....
}
```

Listing 7 Unlabelled Edge Example 2

```java
public String SampleMethod()
{
    ...
    if (str.length() > 0) {
        // Do something
    }
    return str;
    ....
}
```

In both code examples, the GROUM graph is javalang.length-->if-->return. But from the graph, we don't have enough information to distinguish that for the first code snippet the return node is inside the if conditional, while for the second code snippet, the return node is outside of the if block. Instead of using unlabelled edges, if we label the edges correctly to encode nesting information, then we can easily distinguish that the graphs are not identical, and thus should not be matched and grouped together.

4.1.4.4 Eliminate variable-associated edge heuristic

In the GROUM graph, there are edges connecting nodes expressing implicit data dependencies. There is a belief that implicitly related objects tend to be used in near locations of the code. In Figure 2.2 the edge between StringBuffer.<init> and BufferedReader.<init> is based on this heuristic. In our experiment, we have found that these edges result in having unrelated code idioms marked as similar. This is found by manually inspecting the similar code idioms reported by our pattern-matching algorithm. After eliminating these edges our pattern-matching algorithm is able to find more relevant and better results.

4.1.4.5 Simplify Builder and Stream graph representation

The Builder pattern [42] is a creational design pattern in Java that is used to create complex objects with a well-defined sequence of steps. The main purpose of the Builder pattern is to separate the construction of an object from its representation, allowing for greater flexibility and ease of use. The Stream API is a fluent API for processing collections of data in Java. It provides a functional programming style for performing various operations on collections, including filtering, mapping, sorting, and reducing, among others. Both Builder and Stream Fluent APIs are common and popular among developers. For both cases, the sequence and type of APIs may vary among different implementations in projects. But they carry the intention to create objects or process collections. Therefore, we have decided for any Builder and Stream fluent API to label them simply as BuilderPattern and StreamPattern regardless of the length of the call chain. This approach helped us to find similar builder and stream patterns among different projects with our pattern-matching algorithm.

4.1.5 Storing Graphs

Graph generation and graph pattern matching are performed in two separate stages. This is to achieve flexibility in generating GROUM graphs from additional projects and extending our dataset. Therefore, we needed to store the generated GROUM graphs along with all required metadata associated with each graph. We had the option to store these graphs in the Neo4j database as we already use it in our project. But we needed an easy way to manually inspect the generated graphs to check their correctness if needed, and also inspect the outcome of the pattern-matching algorithm to verify the results. After careful thought, we have chosen to use JSON formatted file-based storage of the graphs. The reason is that JSON is a lightweight, flexible and simple data format that is easy to understand and can be easily adapted to fit different needs. It requires minimal setup and can be quickly integrated into the project. It can be easily transferred between different platforms and can be used with a variety of programming languages. We have defined three plain old Java object classes to write and read from the JSON file. One JSON file is generated for each graph.

Listing 8 GROUM Graph POJO

```
public class GroumPojo {
    private String url;
    private String source;
    private List<GroumNodePojo> nodes;
    public List<GroumNodePojo> getNodes() { return nodes; }
    public String getSource() { return source; }
    public void setSource(String source) { this.source = source;
     ↪   }
    public String getUrl() { return url; }
    public void setUrl(String url) { this.url = url; }
    public void setNodes(List<GroumNodePojo> nodes) { this.nodes
     ↪   = nodes; }
}
```

Listing 8 is used for the GROUM graph general information, Listing 9 for node-specific information and Listing 10 for edge-specific information. In this way, we are able to manage all graphs for validation and verification, and it also allows us to experiment by including or excluding specific node types or edge types.

Listing 9 GROUM Node POJO

```
public class GroumNodePojo {
    private int id;
    private String value;
    private String typedValue;
    private String nodeType;
    private List<GroumEdgePojo> incomingEdges;
    private List<GroumEdgePojo> outgoingEdges;
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getValue() { return value; }
    public void setValue(String value) { this.value = value; }
    public String getTypedValue() { return typedValue; }
    public void setTypedValue(String typedValue) {
     ↪  this.typedValue = typedValue; }
    public String getNodeType() { return nodeType; }
    public void setNodeType(String nodeType) { this.nodeType =
     ↪  nodeType; }
    public List<GroumEdgePojo> getIncomingEdges() { return
     ↪  incomingEdges; }
    public void setIncomingEdges(List<GroumEdgePojo>
     ↪  incomingEdges) { this.incomingEdges = incomingEdges; }
    public List<GroumEdgePojo> getOutgoingEdges() { return
     ↪  outgoingEdges; }
    public void setOutgoingEdges(List<GroumEdgePojo>
     ↪  outgoingEdges) { this.outgoingEdges = outgoingEdges; }
}
```

4.2 Pattern Matching Approach

Graph pattern matching is the process of finding subgraphs in a larger graph that match a given pattern. This is a common problem in computer science and has applications in many fields, such as social network analysis, bioinformatics, and data mining. There are several approaches to graph pattern matching, including:

(1) Subgraph Isomorphism: This approach involves searching for subgraphs that are isomorphic to a given pattern. An isomorphic subgraph has the same structure as the pattern, with the same nodes and edges, but may have different labels. Subgraph isomorphism can be computationally expensive, particularly for large graphs.

Listing 10 GROUM Edge POJO

```java
public class GroumEdgePojo {
    private String edgeType;
    private int srcId;
    private String srcValue;
    private int dstId;
    private String dstValue;
    public String getEdgeType() { return edgeType; }
    public void setEdgeType(String edgeType) { this.edgeType = edgeType; }
    public int getSrcId() { return srcId; }
    public void setSrcId(int srcId) { this.srcId = srcId; }
    public String getSrcValue() { return srcValue; }
    public void setSrcValue(String srcValue) { this.srcValue = srcValue; }
    public int getDstId() { return dstId; }
    public void setDstId(int dstId) { this.dstId = dstId; }
    public String getDstValue() { return dstValue; }
    public void setDstValue(String dstValue) { this.dstValue = dstValue; }
}
```

(2) Graph Isomorphism: This approach involves searching for two finite graphs to be matched exactly. It requires nodes, edges and their labels to be matched with one another. Graph isomorphism can be solved in polynomial time and is efficient in practice.

(3) Regular Expression: This approach involves using regular expressions to describe the desired pattern, and searching for subgraphs that match the regular expression. Regular expressions can be powerful for describing complex patterns but may be more difficult to construct and interpret than the other approaches.

(4) Similarity Measure: A similarity measure is a way to quantify the degree of similarity or dissimilarity between two graphs or graph nodes. Similarity measures are often used in graph mining, graph clustering, and graph classification to compare different graphs and find commonalities between them. There are many different types of similarity measures for graphs, and the choice of measure depends on the specific application and the characteristics of the graphs being compared. Some common similarity measures for graphs include graph-edit

distance, Jaccard similarity, cosine similarity and graph kernels.

Overall, the graph pattern-matching approach that is most appropriate will depend on the specific problem being solved, as well as the size and complexity of the graph and pattern involved. Due to the usual size of extracted methods, our initial assumption is that GROUM representation will not be a large graph and will easily fit in the memory during computation. So, we have focused our endeavours to find a scalable approach to search GROUM-based representation in our graph space. We have opted for two types of graph pattern matching, such as exact matching and approximate or partial matching.

There has been an evolution of pattern-matching techniques and algorithms we used for our experiment until we reached a version which we can use reliably. We experimented with three different approaches, including Motif Finding, Clustering and Sequence Similarity Measure. However, we had to discontinue the Motif Finding and Clustering approach due to its limitations. The following sections discuss our experimentation with different graph pattern-matching approaches.

4.2.1 Motif Finding

GraphFrame is a graph processing library for Apache Spark that allows users to work with large-scale graphs. GraphFrame's motif-finding functionality enables to search for patterns or subgraphs in a given graph. Motifs are small subgraphs that appear frequently in a larger graph and are often used to identify patterns or relationships in the data. GraphFrame's motif-finding algorithm uses a pattern-matching technique to search for these motifs. To create the graph space we have appended all our GROUM graphs into a single large graph and inserted this into GraphFrame. To append we have to number the nodes sequentially for all the graphs we have. To specify a motif pattern, we passed each GROUM graph representation as a motif and searched for occurrences in GraphFrame. The pattern can be specified using a set of vertices and edges with specific labels and properties. GraphFrame's motif-finding algorithm searches the input graph for instances of the specified motif pattern. It would return the subgraph occurrences from the input graph. Motif Finder will perform exact matching for the pattern with the input graph.

We have encountered several limitations while using this approach.

(1) We had to create the input graph by appending all the target graphs we wanted to check with. It requires the entire input graph to be loaded into memory, which was challenging when we have a very large graph.

(2) It would be very unlikely to get exact matching for the code fragment from different projects or the occurrences would be very low. As the motif finder tries to match the entire pattern we search for. If we wanted to find subgraph isomorphism then we had to create the pattern for a combination which was not a scalable or efficient approach.

(3) GraphFrame framework implements the searching algorithm so we didn't have much control over it to make any changes to the rules we want to consider for matching.

As we experimented with this approach, it became evident that an exact matching approach won't be a good choice to find the graph patterns across different source projects, as it is highly unlikely that there will be an exact refactoring operation consisting of the same objects, control structure and data dependencies. Hence, an approximate matching approach is more suitable here. We have found only a few control structures matching as frequently refactored code idioms. So, we abandoned this approach and focused on finding a better approach. The efficiency, scaling consideration and lack of control over search heuristics influenced us to make this decision.

4.2.2 Clustering Approach

Clustering graphs is an approach of grouping graphs in a way that similar graphs will be together. Graphs from similar code idioms can be clustered together to find frequently refactored code idioms. In this approach, we have adopted centroid-based k-means clustering and the Jaccard similarity index. For k-means, we have considered both Graph-based and Centroid-based k-means clustering in our approach. The usual way is to represent the data items as a collection of n numeric values in a vector form. In our approach, we have considered all the unique vertices of a graph and edges represented as `src_dst` in the vector form and check for each graph, which has similar nodes and edges. In this step, we have filtered out all graphs that contain only a single node. For the initial centroids, we have selected k random graphs and applied the k-means algorithm until we find a convergence. The usual Euclidean distance is considered to measure the distance to assign a graph into a cluster and to select new centroids, the mean value is used. We have also attempted another approach that considers graph-edit-distance to measure the distance instead of Euclidean distance.

For the Jaccard similarity index, there is a problem with directly using the Jaccard index, because we are considering subgraph isomorphism. This means that when a small graph is matched with part of a larger one, the Jaccard similarity index value will be very small. To avoid this problem we have considered using Szymkiewicz–Simpson coefficient, also known as the overlap coefficient, which is related to the Jaccard similarity index.

$$oc(A, B) = \frac{|A \cap B|}{min(|A|, |B|)}$$

where A and B are two GROUM representations and possibly isomorphic to each other. Graphs that have just a single control or action node were not considered for pattern similarity matching, because they have no edges. This approach gives us a similarity index value between 0 to 1. For our experiment, we used a threshold value of 0.75 to consider two graphs as matched.

With this approach, we could not guarantee that the matching sequences will preserve the control structure of the code, as the edges are matched individually (not in sequence). Even though we did not continue with this approach, we had some positive results. We experimented with this approach with 529 GROUM instances. For Shingles based k-means, the first challenge is to determine the number of clusters for k-means. To determine the optimal value for k, we compute the clustering algorithm for different values of k, varying from 1 to 20. As the value of k changes, there is a discernible pattern in the clustering of graphs at the point of convergence. With a higher k value, the graphs within a cluster tend to be more closely related. Manual evaluation reveals that graphs clustered together have a higher rate of false positives, and very few similar code fragments clustered together.

4.2.3 Sequence Similarity Measure

In the previous approaches, there is a common problem with how we used the GROUM graphs to compare with each other. The subgraph isomorphism problem is a computationally challenging problem in computer science, and the complexity of the problem is heavily dependent on the size and structure of the input graphs. In the worst case, the subgraph isomorphism problem is NP-complete, which means that it is not known to have a polynomial time algorithm. Although the size of our graphs is not that large, the number of graphs we have is relatively large, which makes pattern mining computationally expensive.

Also, we cannot adopt a solution, which would just find exact graph matches, or exact matches considering both nodes and edges, or even exact sub-graph matches. We have to relax the matching rule in a way that preserves the basic code structure and gives us the general pattern which developers follow.

We want to formulate our pattern-matching problem in a way that would be computationally effective and result in the best possible matched patterns. The GROUM graph is a directed acyclic graph and is generated in two stages. The first stage has all the nodes connected with edges based on the control sequence (`temporal-graph`). In the second stage, we include additional data-related edges. So, we could just use the temporal graph and consider it as a tree, which is basically a sequence of nodes, and keep the other edges as metadata. Now, we can try to find such an algorithm which would give us the longest-matched path between two trees. With the output of this approach, we can perform additional checks for the metadata to find the best matches. The temporal graph for our running example shown in Listing 1 would look like Figure 4.3.

If we consider each node from the temporal graph as a character in a string, then we can apply the longest common substring algorithm to find the longest continuous sequence matched between two graphs. We opt for a dynamic programming approach to implement this. We create a matrix of size $(m + 1) \times (n + 1)$, where m and n are the lengths of the two input strings. The $(i, j)^{\text{th}}$ entry of the matrix represents the length of the longest common substring that ends with the i^{th} character of the first string and the j^{th} character of the second string. We initialize the first row and first column of the matrix to zeros since the longest common substring between an empty string and any other string is empty. For the remaining entries, we fill in the matrix iteratively using the following recurrence relation. If the i^{th} character of the first string is the same as the j^{th} character of the second string, then the $(i, j)^{\text{th}}$ entry of the matrix is equal to the $(i - 1, j - 1)^{\text{th}}$ entry plus one. Otherwise, the $(i, j)^{\text{th}}$ entry is zero. After filling in the matrix, we can find the longest common substring by identifying the entry with the maximum value in the matrix and then tracing back to

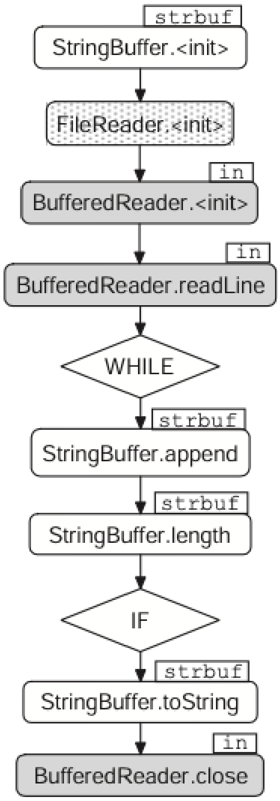


Figure 4.3: GROUM Temporal Representation (Image taken from [1])

the start of the substring. The length of the longest common substring is the value of the maximum entry. The time complexity of this dynamic programming algorithm is $O(mn)$, where m and n are the lengths of the two input graph sequences.

Additionally, we would apply the longest common subsequence algorithm between two GROUM temporal graphs represented as strings. The longest common subsequence problem is solvable in polynomial time. The problem is to find the longest subsequence that is present in two given sequences. A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements. We are implementing a dynamic programming algorithm to solve the longest common subsequence problem that has a time complexity of $O(mn)$, where m and n are the lengths of the two input sequences. We build a matrix of size $(m + 1) \times (n + 1)$ and use it to iteratively calculate the length of the longest common subsequence between the two sequences.

The matrix is initialized with zeros and then filled in row-by-row and column-by-column. At each position (i, j) in the matrix, the algorithm compares the i^{th} element of the first sequence with the j^{th} element of the second sequence. If the elements are equal, the value at position (i, j) in the matrix is set to the value at position $(i - 1, j - 1)$ plus one. Otherwise, the value is set to the maximum of the values at positions $(i - 1, j)$ and $(i, j - 1)$. Once the matrix is filled in, the length of the longest common subsequence is the value at the bottom-right corner of the matrix. The actual subsequence is reconstructed by backtracking through the matrix from the bottom-right corner to the top-left corner.

With this approach, we will find the longest-matched sequence between the two graphs, but the sequence may not necessarily be continuous. We apply a threshold to accept tolerable non-continuous subsequences. We are considering a 60% match of the smallest graph as a tolerable limit

Listing 11 Pattern Matching Steps

```
GroumSimilaritiesPojo result =
 ↪  GroumComparator.GroumExactMatcher(lGroumPojo, rGroumPojo);
if (result.isSimilar()) {
    // Process the result
}
else {
    result = GroumComparator.GroumLCSubStringMatcher(lGroumPojo,
     ↪  rGroumPojo);
    if (result.isSimilar()) {
        // Process the result
    } else {
        result = GroumComparator.GroumLCSMatcher(lGroumPojo,
         ↪  rGroumPojo);
        if (result.isSimilar()) {
            // Process the result
        }
    }
}
```

Besides these two partial matching approaches, we also check for exact matching by checking if the length of the two graphs is the same and if the nodes from the two graphs match with a similar number of incoming and outgoing edges. So, for any two graphs, we first check for an exact match.

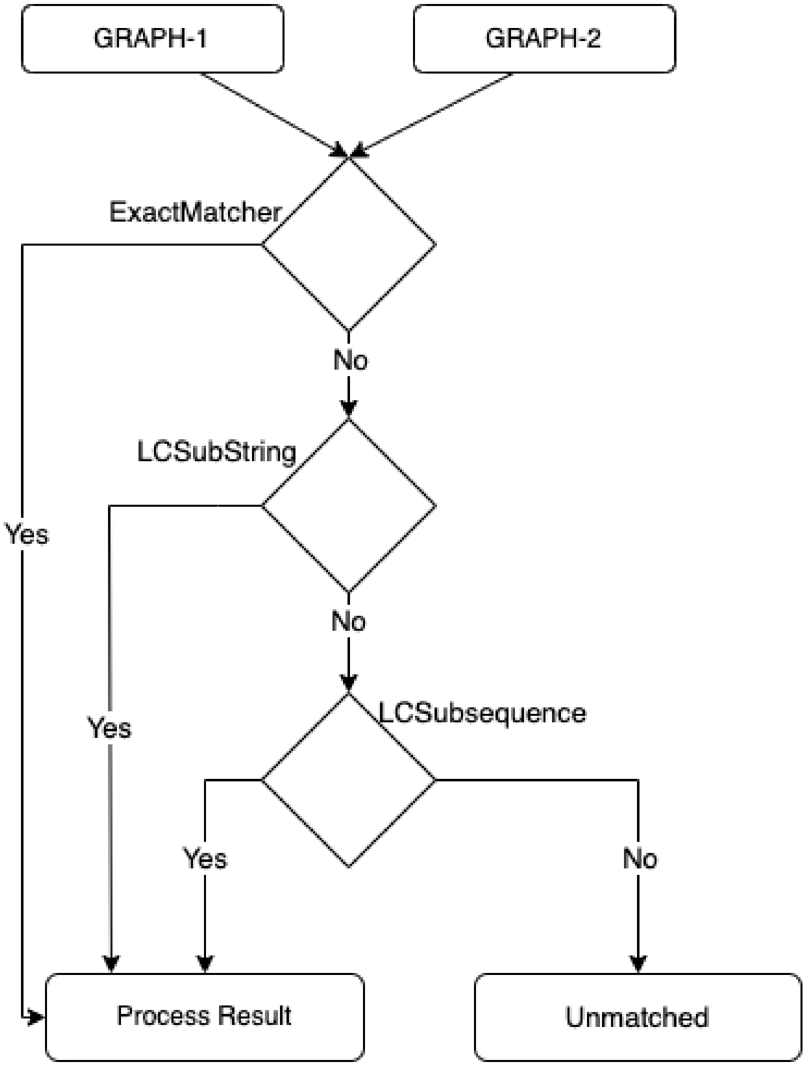


Figure 4.4: Pattern Matching Steps

If we have a positive result we do not proceed with substring matching. Otherwise, we first apply the longest common substring matching approach and only if no substring is found, then we apply the longest common subsequence approach. Listing 11 shows the sequence of steps.

Chapter 5

Experiment Setup

5.1 Mining Refactorings

In this Section, we explain our approach for collecting refactoring instances from open-source projects. These instances will be used to extract frequently refactored code idioms.

5.1.1 Project selection

We have considered several criteria to select projects for our experiment, as shown in Table 5.1, which are commonly used in Mining Software Repositories (MSR) studies [43]. Next, we searched for Java projects hosted on GitHub that meet these criteria. To do that we have considered projects with 500 or more stargazers. The total number of projects was more than 5000 with 500 or more stargazers. Among these projects, we have filtered projects which are not mature, not maintained or not active anymore. This filtering process considered different conditions such as projects with 100 or more commits, at least 2 contributors, the minimum number of releases is 10 and the project is at least 2 years old. Among the resulting 742 projects after applying the aforementioned criteria, we randomly picked 110 projects for our experiment.

Table 5.1: Project Selection Criteria

Language	Stargazers	Contributors	Commits	Release	Activity
Java	≥ 500	≥ 2	≥ 100	≥ 10	$\leq 365 days$

5.1.2 Project Mining for Refactoring

According to Fowler [18] catalogue, there are at least 66 types of refactoring operations that can be performed on a source code. Among these, we are considering only the EXTRACT METHOD refactoring for various reasons:

(1) It is one of the most commonly applied refactorings by developers [44]

(2) It is well supported by all IDEs

(3) According to Silva et al. [8], EXTRACT METHOD refactoring is motivated by a large number of different reasons (11 in total).

(4) Refactoring mining tools, such as RefDiff [45] and RefactoringMiner [21] can detect EXTRACT METHOD refactoring instances with very high precision and recall.

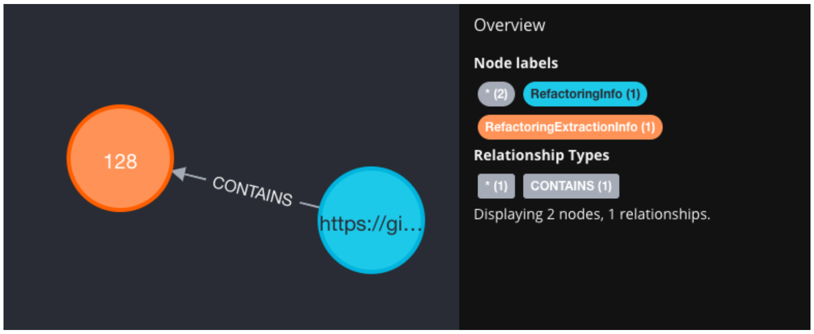
For our experiment, we have decided to use RefactoringMiner as it is considered the state-of-the-art tool for mining refactoring with an overall precision of 99.6% and recall of 94%, and a precision of 99.8% and recall of 96% for EXTRACT METHOD in particular [21]. We ran RefactoringMiner in the entire commit history of the 110 selected projects. A total of 31,427 commits were processed and a total of 65,742 EXTRACT METHOD refactoring instances were extracted. These are all unique instances of EXTRACT METHOD refactorings, as we ignore duplicate instances returned by RefactoringMiner, such as duplicated code extracted from multiple methods into a single common method. For our purpose, we avoid duplicates within the commit as our goal is to find refactored code idioms within different projects, hence it will be redundant to compare the duplicated code fragments. RefactoringMiner reports duplicated code extracted to a single method, as separate EXTRACT METHOD instance, one for each duplicated code fragment. In such cases, we include only one of the reported instances in our dataset. Table 5.2 shows the total number of unique EXTRACT METHOD refactoring instances we have collected.

Table 5.2: Mined Extract Method Refactoring Instances

No. of Projects	No. of Commits	No. of Extract Method
110	31,427	65,742

5.1.3 Storing Refactorings

During the collection process, we also gather some metadata related to an Extract Method refactoring instance and store it in a graph-based database. We have used Neo4j [46] as our database system. The reason we choose Neo4j is that it is the most dominant, active and widely used graph database, it provides high performance for both read and write operations. It is also simpler and more expressive than relational or other NoSQL databases. The model we designed for the related metadata for an Extract Method instance is more suited for a graph-based database rather than a relational database, as it is easy to search and easy to extend for any additional information we wanted to store. In a later step, we needed a graph-based database to calculate subgraph-isomorphism, so we decided to stick with a single database system instead of having multiple. The relationship type between the nodes is defined as CONTAINS where two nodes describe different sets of information. Figure 5.1 shows a graph view representation of a relation between two nodes exported from the Neo4j database. The blue and orange colours represent two different types of nodes, where the blue node is labelled as RefactoringInfo and the orange node is labelled as RefactoringExtractionInfo, and the edge indicates that RefactoringInfo contains a relation with RefactoringExtractionInfo.



Displaying 2 nodes, 1 relationships.

Figure 5.1: Representation of Node Relations

Table 5.3: Node Details

Node	Properties
RefactoringInfo	`id, commitId, projectUrl, uuid`

RefactoringExtractionInfo	`id, filePath, length, sourceCode,`
	`startLine, endLine, startOffset, uuid`

Table 5.3 shows the details of the nodes with the property values we store. We also keep the list of commits with GitHub URLs for a project in a CSV (Comma-separated values) formatted file. This is just for recording purposes allowing us to manually check a particular commit from a project on GitHub commit page.

5.2 Pattern Finding Experiment setup

As explained in Section 5.1, we have selected a total of 110 Java project repositories from GitHub, and processed 31,427 commits to collect 65,742 unique EXTRACT METHOD refactoring instances. After the first phase of generating a GROUM representation for each EXTRACT METHOD refactoring instance, we have a total of 44,631 GROUM graphs. There is a number of reasons for not getting the same number of GROUM graphs as the number of EXTRACT METHOD refactoring instances.

(1) We have considered an EXTRACT METHOD for GROUM generation only if it has at least one external method invocation, class instance creation or field access. The extracted methods with only internal method invocations, class instance creations or field accesses have not been considered. The explanation behind this decision is that we do not expect to find similar ACTION nodes (i.e., method invocations, class instance creations and field accesses) that are internal to a project in another project.

(2) A GROUM graph without any ACTION node is not considered. The only exception is for graphs containing a single CONTROL node, which is a `switch` statement.

(3) We also pruned out graphs having a single node. This is to avoid having a large number of insignificant true positive matches.

Besides these constraints, we are also missing graphs due to the limitation imposed by the API-Finder library we are using to resolve binding information. To address API Finder's limitation discussed in Section 4.1.2.1, we have added a fallback mechanism (Section 4.1.2.2) to handle the cases for which API Finder cannot return a given method invocation, class instance creation or field access binding.

Table 5.4: GROUM Graph distribution

Total	Production Code	Test Code
44,631	36,386	8,245

We have separated the total graphs into two categories, production code and test code. We have a total of 36,386 production code graphs and 8,245 test code graphs. The test code graphs are separated by checking if the EXTRACT METHOD belongs under the project's `test` directory. During pattern matching, we check production code graphs against another project's production code graphs and test code graphs against another project's test code graphs. This is because production and test code are of a different nature and serve completely different purposes. We have also considered a few constraints during the pattern matching. The constraints are:

(1) We only compare a graph from one project with other projects. This is to allow finding similarities across projects and avoid inner-project similarities.

(2) We exclude any graph which has less than 3 nodes, which means we only consider a graph to check for similarity if the number of nodes is 3 or more.

(3) A similarity result of fewer than 3 nodes is not counted towards the result.

(4) Given two graphs, a similarity of 60% or higher is required to consider their common subgraph as a code idiom pattern.

After conducting an initial investigation of our results without constraints or with different constraint values, we have added the above constraints based on our findings. We have determined that a graph with only 2 nodes or fewer is not a reliable indicator of a code idiom pattern. Additionally, matched sub-graphs with less than 60% similarity tend to represent different code idioms rather than similar ones. Our manual investigation has also revealed that many of these patterns overlap with other patterns that are representative towards actual code and seem more purposeful.

Chapter 6

Experiment Results

Our exploratory study was done in two phases. The first phase includes the generation of our extended version of GROUM graph representation and in the second phase, we apply a pattern-matching algorithm to find frequently refactored code idioms. This chapter will present the results of our pattern-finding phase and will focus on addressing two research questions from our study.

RQ1. What are the most frequently refactored code idioms?

RQ2. Are there different frequently refactored code idioms for test and production code?

The **RQ1** is the main objective of our study and **RQ2** examines if there are different refactored code idioms for production code and test code.

6.1 Frequently Refactored Code Idioms

We have found 447 code idiom patterns from the production code dataset and 42 patterns from our test code dataset. For each code idiom pattern, the occurrences range between 2 to 62. We count 1 occurrence from each project, which means the number of occurrences is the total number of projects where we have that particular refactored code fragment. In this section, we have listed 22 frequently refactored code idioms. 21 of them have been found in 4-62 projects and one of them has been found in 3 projects. We included the latter code idiom, because it was found only in test codes. We have assigned an Id as idiom-`x`, where `x` indicates the serial of the code idiom. We also gave a short summary of the functionality that the code idioms perform. The list includes unique and diverse types of purposeful code fragments. For each code idiom we provide slight variations among its instances found in various projects.

Listing 12 Collection Generation from Input 1

```
private ArrayList<Symbol> getProcessedArgs(List<Symbol>
 ↪  arguments, SourceSymbols sourceSymbols) {
    ArrayList<Symbol> args = new ArrayList<>(arguments.size());
    for (Symbol arg : arguments) {
        args.add(process(arg, sourceSymbols));
    }
    return args;
}
```

Listing 13 Collection Generation from Input 2

```
private static List<FilterChain> validateAndSelectFilterChains(
    List<io.envoyproxy.envoy.config.listener.v3.FilterChain>
     ↪  inputFilterChains)
    throws InvalidProtocolBufferException {
    List<FilterChain> filterChains = new
     ↪  ArrayList<>(inputFilterChains.size());
    for (io.envoyproxy.envoy.config.listener.v3.FilterChain
     ↪  filterChain :
        inputFilterChains) {
        if (isAcceptable(filterChain.getFilterChainMatch())) {
            filterChains.add(FilterChain.fromEnvoyProtoFilterCha⌋
             ↪  .in(filterChain,
             ↪  false));
        }
    }
    return filterChains;
}
```

Among the instances, we have found there are a few overlapping or shared patterns with some level of differences. We avoid listing code idioms that exhibit a minor variance or achieve closely similar purposes with a slight difference. For example, let's consider two code snippets from Listing 12 from repository create and Listing 13 from repository grpc-java, These two methods are apparently doing the same thing, creating a new collection based on the size of the input collection, iterating over the input collection, adding to the newly created collection a new object created based on the element of the current iterator, and finally return the newly populated collection. There is one difference between the two methods, as Listing 13 has a `if` filtering condition before adding to the collection. The top-22 most frequently refactored code idioms are presented in Tables 6.1 and 6.2.

Table 6.1: Code Idioms

Id	Name	Actions	#
Idiom-1	Iterate and Create collections	- Loop through a collection input - Populate another collection - Return the newly created collection	16
Idiom-2	Switch Block	- switch expression - Check value for case - Return	43
Idiom-3	Stream pipeline	- Stream generator - Zero or more intermediate operations - Terminal operation - Return	15
Idiom-4	Collection builder	- Create a collection type - Manipulate the newly created collection - Return	13
Idiom-5	Safely Execute a startup routine	- Perform a sequence of actions inside the TRY block - Catch exception for unintended behaviour - Throw exception	4
Idiom-6	Get conditional value	- Check for one or more condition - Get the value if satisfy - Return	29
Idiom-7	Properties Builder	- Create an object for key-value pair - Populate properties - Return	8
Idiom-8	Safely create an Object	- Create an object inside the TRY block - Handle if Exception occur - Return	17
Idiom-9	Create a File or Directory	- Create File object - Make directory - Return	16
Idiom-10	String Builder	- Create String object - Populate string - Return	30
Idiom-11	Iterative Exception checking	- Loop over a condition - Check for unexpected value - Throw Exception	18

Table 6.2: Code Idioms (continued)

Id	Name	Actions	#
Idiom-12	Container Iterator	- Loop over a container - Perform some action - Return	22
Idiom-13	Builder Fluent Pattern	- Create Builder instance - Sequence of setter methods - Build - Return	6
Idiom-14	Value sanitiser	- Check for a value - Throw Exception if unexpected - Return	62
Idiom-15	Find a value	- Iterate over items - Check for expected value - Return	28
Idiom-16	Safe Method call	- Invoke a method inside TRY block - Catch Exception - Return	43
Idiom-17	String manipulation	- Accept String as parameter - Manipulate String - Return	34
Idiom-18	Object with Unique Identifier	- Generate UUID - Create object - Return	6
Idiom-19	Check and Assert	- Take parameters - Assert for values of parameters	3
Idiom-20	Get Resource	- Get a class object - Call a method with parameter - Return	18
Idiom-21	Thread Safety	- Call lock method - Perform an action inside TRY block - Finally Unlock	5
Idiom-22	Safely Close Resource	- Call close inside TRY block - Handle Exception	12

6.2 Code Idioms

We have presented the top-22 frequently refactored code idioms in Tables 6.1 and 6.2 with their occurrences in our data set. We shall discuss each of our code idioms and their primary tasks with some code examples.

6.2.1 Iterate and Create Collections

The extracted method for this idiom usually takes a parameter of a collection type. Inside the method, it populates another collection type by iterating over the collection passed as a parameter. We have found a total of 16 instances for this code idiom. The collection types typically include HashMap, Map, List, ArrayList, Set or any other Java data structure included in the Java collection framework.

Listing 14 Iterate and Create Collections

```
private List<HasMetadata> createItemsInKubernetesList(KubernetesList list) {
    List<HasMetadata> createdItems = new ArrayList<>();
    for (HasMetadata r : list.getItems()) {
      HasMetadata created = create(r);
      createdItems.add(created);
    }
    return createdItems;
  }
}
```

Listing 14 is from kubernetes-client and shows an extracted method, which creates a new `ArrayList` named `createdItems`, iterates over the elements of `list` passed as a parameter in the extracted method, and adds newly created objects based on the element of the current iteration `r`. The reason for extracting this code idiom is to generate a collection containing new objects generated from the objects included in the original collection.

6.2.2 Switch Block

Extracting a method including a switch block is one of the most frequent code idioms in our data set. We have found 43 instances of switch block code fragments. There are differences among the implementations regarding the number of handled cases, additional processing for the matched case, presence of return statement or throwing exceptions. Despite having differences, developers refactor switch code blocks to separate methods. Listing 15 shows an instance for this code idiom.

Listing 15 Switch block with parameter 1

```
public static int getButtonTextResId(int buttonType) {
    switch (buttonType) {
        case BUTTON_EDIT:
            return R.string.button_edit;
        case BUTTON_VIEW:
            return R.string.button_view;
        case BUTTON_PREVIEW:
            return R.string.button_preview;
        case BUTTON_STATS:
            return R.string.button_stats;
        case BUTTON_TRASH:
            return R.string.button_trash;
        case BUTTON_DELETE:
            return R.string.button_delete;
        case BUTTON_PUBLISH:
            return R.string.button_publish;
        case BUTTON_MORE:
            return R.string.button_more;
        case BUTTON_BACK:
            return R.string.button_back;
        default:
            return 0;
    }
}
```

6.2.3 Stream pipeline

Listing 16 Stream pipeline 1

```
private List<HasMetadata> performOperation(
  Function<? super NamespaceableResource<HasMetadata>, ? extends
   ↪  HasMetadata> operation) {
    return getResources().stream().map(operation)
    .collect(Collectors.toList());
}
```

The stream pipeline code idiom may differ from one instance to another by the type of intermediate stream operations, and the number of stream operations. We generalised the stream pipeline pattern and grouped all relevant instances together. We have found a total of 15 such extracted

Listing 17 Stream pipeline 2

```java
private List<Node> nodesFilter(Predicate<PrioritizableNode>
 ↪  predicate) {
    return nodes.stream()
            .filter(predicate)
            .map(n -> n.node)
            .collect(Collectors.toList());
}
```

methods. A stream is a sequence of elements supporting the parallel and sequential aggregate operation. From our findings, it seems that the developers want to have this functionality in a separate method. Listing 16 from kubernetes-client and Listing 17 from vespa-engine show two instances of the Stream pipeline code idiom.

6.2.4 Collection builder

This pattern is different from Idiom-1 (Iterate and Create Collections). In this pattern, a collection is created with its elements generated internally. As we discussed in our methodology section if a method is invoked multiple times repeatedly we consider this as one invocation. This helped us to group instances of this pattern where the main objective is to create a list, populate its elements and return it. We have found a total of 13 instances for this code idiom. Among the instances, the type of collection may differ, as well as the number of elements added to the collection. Listing 18 from apache-hbase shows an extracted method with this code idiom.

Listing 18 Collection builder

```java
protected List<CandidateGenerator> createCandidateGenerators() {
    List<CandidateGenerator> candidateGenerators = new
     ↪  ArrayList<CandidateGenerator>(4);
    candidateGenerators.add(new RandomCandidateGenerator());
    candidateGenerators.add(new LoadCandidateGenerator());
    candidateGenerators.add(localityCandidateGenerator);
    candidateGenerators.add(new
     ↪  RegionReplicaRackCandidateGenerator());
    return candidateGenerators;
}
```

6.2.5 Safely execute a startup routine

We have found this code idiom in Android and Database related projects. Typically, the code idiom starts some activity inside a `try` block, handles any thrown exception in a `catch` block, and returns success or failure. We have found this code idiom recurring in 4 different projects. Listing 19 from facebook-android-sdk shows an extracted method handling an activity startup. Developers tend to have this functionality in a separate method for reuse and to manage the startup process safely.

Listing 19 Safely execute a startup routine

```
static boolean tryKatanaIntent(Activity activity, Intent intent)
 ↪  {
    if (intent == null) {
        return false;
    }

    try {
        activity.startActivityForResult(intent,
         ↪  DEFAULT_REQUEST_CODE);
    } catch (ActivityNotFoundException e) {
        return false;
    }

    return true;
}
```

6.2.6 Get conditional value

This is a simple code idiom that computes a variable based on some conditional logic. In this code idiom, there are one or more conditional `if-else-if` statements checking for some expected value or expression and then returning the computed variable. A return statement may exist in each conditional block or once at the end of the method. This method extraction typically occurs when the developer wants to add new conditions to an existing conditional logic. As the code gets more complex, the developers tend to extract this functionality in a separate method. Listing 20 from hyphanet-fred shows an extracted method for this code idiom. We have found this idiom in

29 different projects.

Listing 20 Get conditional value

```
public static String getSimplifiedArchitecture() {
    String arch;
    if(System.getProperty("os.arch").toLowerCase().matches("(i?[⌋
     ↪   .x0-9]86_64|amd64)"))
     ↪   {
        arch = "amd64";
    } else if(System.getProperty("os.arch").toLowerCase().matche⌋
     ↪   .s("(ppc)"))
     ↪   {
        arch = "ppc";
    } else {
        arch = "i386";
    }

    return arch;
}
```

6.2.7 Properties Builder

This code idiom starts with creating a key-value object and then adds a set of key-value pairs. Once populated the key-value object is returned. This is mostly done to keep the configuration or property values mapped with a key. Java `Map` or `HashMap` type collections are typically used. This method keeps all such configurations in a single place. We have found a total of 8 instances of this code idiom pattern. For this idiom, the number of intermediate `put` method calls may vary from one implementation to another. Listing 21 from apache-kafka project shows an instance of this pattern.

6.2.8 Safely create an Object

We have found 17 instances for this code idiom pattern. The objective of the extracted method code fragment is to create an object safely. This pattern has different variations. The object is either created inside a `try` block any thrown exception is handled in a `catch` block, and finally, the created object is returned. In another way, the object is created first and the is some conditional

Listing 21 Properties Builder

```
static Properties getStreamsConfig() {
    final Properties props = new Properties();
    props.put(StreamsConfig.APPLICATION_ID_CONFIG,
     ↪  "streams-wordcount");
    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
     ↪  "localhost:9092");
    props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 0);
    props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
     ↪  Serdes.String().getClass().getName());
    props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
     ↪  Serdes.String().getClass().getName());
    // setting offset reset to earliest so that we can re-run
     ↪  the demo code with the same pre-loaded data
    // Note: To re-run the demo, you need to use the offset
     ↪  reset tool:
    // https://cwiki.apache.org/confluence/display/KAFKA/Kafka+S⌋
     ↪  .treams+Application+Reset+Tool
    props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
     ↪  "earliest");
    return props;
}
```

logic checking for the validity of the created object. If the object is invalid then an exception is thrown, otherwise, the object is returned. Listing 22 from repository geotools shows an instance of the safe object creation code idiom. Here a variable is initialized with a `null` value and then it is assigned with an object returned by invoking a method. The object can also be created by an object creation statement for this pattern. The validity of the object often is determined by checking whether the object is null or not. The purpose behind this code fragment being extracted is that developers do not want to duplicate the exception handling or validity checking every time a new object is created and prefer to reuse the extracted method.

6.2.9 Create a File or Directory

This code idiom has some similarities with the Safely Create an Object idiom, but the main difference is that it involves the creation of a `File` object. The code idiom starts with creating a File object with some additional information supplied, then a directory is created for the File object

Listing 22 Safely create an Object

```
private static <T> Collection<T> newCollection(Class<? extends
 ↪  Collection> collectionClass) {
    Collection<T> clone = null;
    try {
        clone = collectionClass.getDeclaredConstructor().newInst⌋
         ↪  .ance();
    } catch (Exception e) {
        clone = new ArrayList<>();
    }
    return clone;
}
```

and finally the created directory is returned. This code idiom may also include exception handling. We have found this idiom 16 times in our data set. Listing 23 from aws-sdk-android shows an extracted method corresponding to this code idiom.

Listing 23 Create a File or Directory

```
private static File makeDirectory() {
    Context appContext =
     ↪  InstrumentationRegistry.getTargetContext();
    File directory = new
     ↪  File(appContext.getApplicationContext().getFilesDir(),
     ↪  DIR_NAME);
    directory.mkdir();
    return directory;
}
```

6.2.10 String Builder

The purpose of this code idiom is to create a string using the Java StringBuilder or StringBuffer API. The pattern starts with creating a StringBuilder object, then the StringBuilder object is appended with some data, and eventually, the method returns by calling the `toString` method on the StringBuilder object. This is one of the most frequently refactored code idioms with instances in 30 different projects. Listing 24 from FairEmail project shows an extracted method following this code idiom.

Listing 24 String Builder

```java
static String hex(byte[] bytes) {
    StringBuilder sb = new StringBuilder();
    for (byte b : bytes)
        sb.append(String.format("%02x", b));
    return sb.toString();
}
```

6.2.11 Iterative Exception checking

This code idiom pattern commonly involves a loop, and inside the loop body, a conditional statement throws an exception. This is a high-level pattern with these three common statements nested within each other (i.e., loop → conditional → throw). In our dataset, we have found a total of 18 such instances. Listing 25 shows an instance from project mongo-java-driver. This code idiom may additionally include method invocations within the body of the loop.

Listing 25 Iterative Exception checking

```java
public static void readFully( InputStream in, byte[] b )
    throws IOException {
    int x = 0;
    while ( x<b.length ){
        int temp = in.read( b , x , b.length - x );
        if ( temp < 0 )
            throw new EOFException();
        x += temp;
    }
}
```

6.2.12 Container Iterator

This code idiom varies from the previous idioms iterating over the elements of a collection by using an `Iterator` object to perform the iteration. Instead of providing a direct reference to the collection that would allow modification of the collection by adding or deleting elements, the `Iterator` object guarantees that the original collection will remain unmodified. In this idiom, there is a loop iterating over the elements of the collection by calling the `hasNext()` and `next()`

APIs to perform some action for each item in the collection. We have found a total of 22 such instances in our dataset. This code idiom is extracted with the purpose of performing an iteration over the elements of a collection without modifying the collection. Listing 26 from project apache-tinkerpop shows an extracted method with the Container Iteration code idiom.

Listing 26 Container Iterator

```java
private Long getCount(final Iterator<Long> longs) {
    long count = 0l;
    while (longs.hasNext()) {
        count = count + longs.next();
    }
    return count;
}
```

6.2.13 Builder Fluent Pattern

Builder is a creational design pattern [42], which serves the purpose of creating and initializing an object without calling its constructor. It is particularly useful for objects with a large number of attributes, some of which may have default values. Typically the Builder pattern is utilized as a sequence of method invocations, starting with creating a static Builder type object, then a series of method invocations to set the attributes of the object, and finally a `build()` method invocation that created and returns the built object. For this code idiom, the intermediate method calls may vary in different implementations. The commonality we looked for is starting with a Builder object creation and ending with a `build()` method invocation. This idiom may manifest itself either in fluent API style or a series of statements grouped together. Listing 27 from apache-druid exhibits an instance of the Builder Fluent idiom. We have found a total of 6 instances of this idiom in our dataset.

6.2.14 Value sanitiser

This code idiom is the most frequently refactored one, as we found a total of 62 instances. It has some overlap with the Safe Method call idiom (Section 6.2.16). Typically, in this code idiom, a variable is created by calling a method (the method call could optionally use a parameter of the

Listing 27 Builder Fluent Pattern

```
Response buildNonOkResponse(int status, Exception e) throws JsonProcessingException
{
  return Response.status(status)
                 .type(contentType)
                 .entity(newOutputWriter(null, null, false).writeValueAsBytes(e))
                 .build();
}
```

parent method). Then, the created object is checked whether it has a `null` value, and if so an exception is thrown, otherwise, the variable is returned. Listing 28 from apache-druid shows an instance for this code idiom.

Listing 28 Value sanitiser

```
private VirtualColumn getVirtualColumnForSelector(String columnName)
{
    VirtualColumn virtualColumn = getVirtualColumn(columnName);
    if (virtualColumn == null) {
        throw new IAE("No such virtual column[%s]", columnName);
    }
    return virtualColumn;
}
```

6.2.15 Find a value

This code idiom iterates over a collection and performs a conditional check to find an expected value (e.g., max or min) regarding a property of the collection elements. It may return the computed value either at the end of the method or inside the conditional block. We have found 28 instances of this code idiom. There may exist additional processing statements once the expected value is found. A code fragment in Listing 29 from project apache-hbase shows the basic workflow for this idiom. Here the developer wants to make this functionality reusable in a separate method that computes the highest replicaID region.

Listing 29 Find a value

```java
private static int getMaxReplicaId(List<RegionInfo> regions) {
    int max = 0;
    for (RegionInfo regionInfo: regions) {
        if (regionInfo.getReplicaId() > max) {
            // Iterating through all the list to identify the highest replicaID region.
            // We can stop after checking with the first set of regions??
            max = regionInfo.getReplicaId();
        }
    }
    return max;
}
```

6.2.16 Safe Method call

This is one of the most frequently refactored code idioms found in our dataset with a total of 43 instances. The code idiom calls a method within a `try` block. If there is an exception thrown by the involved method, then it is handled by catching the exception and often throwing another exception or returning failure. Otherwise, the return value of the invoked method is returned. This code idiom shares some commonality with other idioms we have found, but the basic pattern for this code idiom is to invoke a method within a try-catch block. Listing 30 from assertj project has an extracted method representing this code idiom.

Listing 30 Safe Method call

```java
private static <V> boolean containsValue(Map<?, V> actual, V value) {
    try {
        return actual.containsValue(value);
    } catch (NullPointerException e) {
        if (value == null) return false; // null values not permitted
        throw e;
    }
}
```

6.2.17 String manipulation

This pattern has similarities with the Idiom-10 (String Builder), but the difference is it uses a `String` object, manipulates it, and returns another string object. Commonly it is used to customise the input string or construct a message and return the message. There are a variable number of intermediate methods and types to form the string message. There are 34 instances of this refactored code idiom in our data set. Listing 31 from project javaparser shows an instance of this code idiom, which takes a `String` parameter as input, and processes it to return another string.

Listing 31 String manipulation

```java
private static String removeTypeArguments(String typeName) {
    if (typeName.contains("<")) {
        return typeName.substring(0, typeName.indexOf('<'));
    } else {
        return typeName;
    }
}
```

6.2.18 Object with Unique Identifier

This is one of the least frequently refactored code idioms we have found in our dataset. We have found this code idiom in 3 instances in our production code dataset and another 3 instances in the test code dataset. In this code idiom, an object is instantiated by passing a parameter, which is used to create a random UUID as a string. Passing the parameter as a unique identifiable string makes this code idiom different from other object creation idioms. Listing 32 from project palantir-atlasdb exhibits an instance of this code idiom.

Listing 32 Object with Unique Identifier

```java
public static PaxosAcceptorState getAcceptorStateForRound(long
 ↪  round) {
    return PaxosAcceptorState.newState(new PaxosProposalId(
                    round, UUID.randomUUID().toString()));
}
```

6.2.19 Check and Assert

We have found 3 instances of this code idiom in our test code dataset. The method is extracted with parameters and has a sequence of assert statements. The purpose of this extracted method is to facilitate checking additional fields and to be reused for different unit tests. The pattern includes asserting statements with the additional method call. We can see an extracted method representing this code idiom in Listing 33 from the test code of the project Netflix-genie.

Listing 33 Check and Assert

```
private <T> void testOptionalField(
        final Supplier<Optional<T>> getter,
        final Consumer<T> setter,
        final T testValue
) {
    Assertions.assertThat(getter.get()).isNotPresent();
    setter.accept(null);
    Assertions.assertThat(getter.get()).isNotPresent();
    setter.accept(testValue);
    Assertions.assertThat(getter.get()).isPresent().contains(tes⌋
     ↪   .tValue);
}
```

6.2.20 Get Resource

This code idiom is mostly found in our test code dataset. The method takes a parameter to retrieve a particular resource. The code idiom includes getting a class object, invoking a method with the parameter, and returning it. The purpose of the extracted method is to make it reusable for various tests. We have found 18 instances of this code idiom in our data set. Listing 34 from project mockito shows an extracted method representing this code idiom.

Listing 34 Get Resource

```
private Field field(String fieldName) throws
 ↪  NoSuchFieldException {
    return this.getClass().getDeclaredField(fieldName);
}
```

6.2.21 Thread Safety

When a particular task needs to be performed in a synchronised thread-safe way, developers tend to extract the locking functionality in a separate method. A lock object is created and the `lock` method is called. Then, the task is performed inside a `try` block with a `finally` block to `unlock` the lock at the end. We have found this code idiom in 5 different projects. Listing 35 from project eclipse-jetty shows an instance of this code idiom.

Listing 35 Thread Safety

```
private ByteBuffer queuePoll()
{
    Lock lock = _lock;
    lock.lock();
    try
    {
        return _queue.poll();
    }
    finally
    {
        lock.unlock();
    }
}
```

6.2.22 Safely Close resource

This code idiom performs the specific task of closing a resource, connection or exiting from an activity. The higher-level pattern may seem similar to other safe-handling code idioms, but it performs the particular task of closing resources, which makes this code idiom different from others. In this code idiom, there is a `try` block that attempts to close a resource and handles any thrown exceptions. The intention behind extracting such methods is to handle the resource closing safely and separately in a reusable method. We have found a total of 12 different instances of this code idiom. Listing 36 from project apache-phoenix performs a close action inside a `try` block and logs the error in case of a failure.

Listing 36 Safely Close resource

```java
private static void tryClosingResourceSilently(AutoCloseable
 ↪  res) {
    if (res != null) {
        try {
            res.close();
        } catch (Exception e) {
            LOG.error("Closing resource: " + res + " failed :",
             ↪  e);
        }
    }
}
```

6.3 Discussion

Our hypothesis for Research Question #1 is that there exist repetitive code patterns (i.e., code idioms) that developers tend to refactor. The main motivation behind our experiment was to find if this hypothesis is true and what are the most frequently refactored code idioms. Based on our experiment results, we presented a list of the most frequently refactored code idioms in Tables 6.1 and 6.2, which support our hypothesis. The selected projects for this experiment are from various domains and we have found that the frequent refactored code idioms are spread in projects from different domains. Isolating the complexity of a single-purpose code fragment and making it extensible and reusable seems to be the most prevalent motivation behind the refactoring of these code idioms.

Table 6.3: Code Idioms distribution

Production Code only	Test Code only	Both Production and Test Code
8	2	12

Research Question #2 intends to investigate if the refactored code idioms are common between production and test code, or specific for production and test code, respectively. To find this we have separated our dataset into production code and test code. We found 2 code idioms only in the test code, 8 code idioms only in the production code, and 12 code idioms both in the test and production code. The number of total code idioms found in test code is around 10% of the number of code idioms in production code. The finding supports our hypothesis that there exist some specific code idioms that are unique for test and production code, although some code idioms are common for both test and production code.

Initially, there were 65,742 instances of the EXTRACT METHOD. By applying graph selection constraints, the total number of GROUM graph instances was reduced to 44,631. After applying the pattern matching technique, we found a total of 8,754 occurrences distributed over 489 code idioms both in production code and test code. Figure 6.1 shows how the GROUM graph number changes in each step.

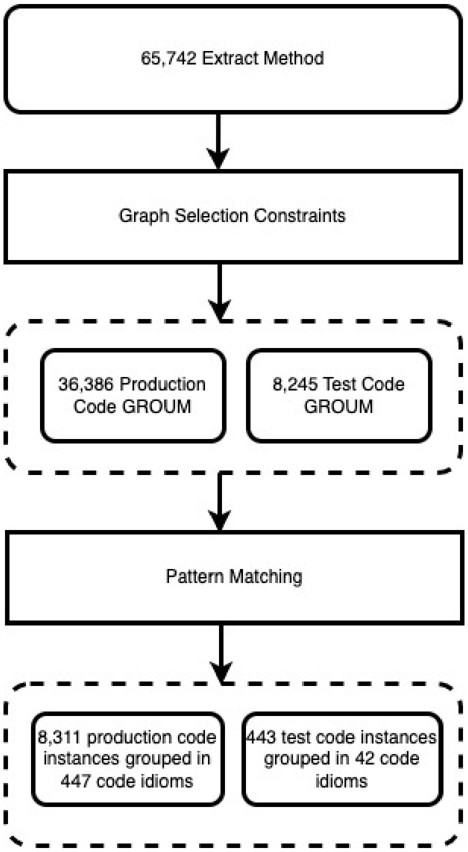

65,742 Extract Method
Graph Selection Constraints
36,386 Production Code GROUM
8,245 Test Code GROUM
Pattern Matching
8,311 production code instances grouped in 447 code idioms
443 test code instances grouped in 42 code idioms

Figure 6.1: Number of graphs in each step of our process.

6.3.1 Comparison with previous work

In this thesis, we have built upon the work of *Frequently Refactored Code Idioms* [17]. By continuing this work, we were able to delve deeper into the research questions and make new findings. In our study, we have presented the 22 most frequently refactored code idioms pattern in contrast to the previous work that presented 11 such patterns. We have found all of the previously found code idioms in our study (some of them were exactly the same as our code idioms, while others were subsets of our code idioms, i.e., some of our code idioms are more coarse-grained). Moreover, we have found a wider range of coding styles compared to the previous work. There are 11 new unique code idioms identified by our approach.

The previous work processed a total of 1025 projects to collect 47,647 refactoring instances whereas we have collected 65,742 EXTRACT METHOD refactoring instances from 110 projects. We have also found a total of 8754 instances distributed over 489 code idioms whereas the previous work has found 723 refactoring instances distributed over 185 code idioms. This shows the improvement in the overall approach of our study.

There are distinguishable differences between our approach and the previous approach considering the graph representation and similarity matching technique. We have added support for more language features and improved the GROUM representation and pattern-matching algorithm to find higher-level similarities between GROUM graphs.

(1) We added support for `switch` block, `try-catch` block, `enhanced-for` loop, `return` statement, `throw` statement.

(2) We introduced an abstract code representation for the `stream` pattern, `builder` pattern, and compress sequential repetitive calls.

(3) We introduced edge labelling to preserve nesting and block-related information.

(4) We supported all required edges to complete the GROUM graph representation using our enhanced PDG implementation

(5) We relied on a state-of-the-art tool (API Finder) to resolve binding information without building the code.

(6) We developed a custom pattern-matching algorithm by treating the GROUM temporal graph as a sequence.

Chapter 7

Threats to Validity

7.1 External Validity

We limit our study to only Java open-source projects, as we depend on RefactoringMiner [21] to collect EXTRACT METHOD refactoring instances from each commit of our selected projects. RefactoringMiner works only for Java projects, and thus we cannot claim that our study is language-independent, or can be applied to industrial projects, or projects based on other languages. However, the approach we used to represent the code fragments and our pattern-matching algorithm can be applied to code written in any object-oriented language. Moreover, the EXTRACT METHOD refactoring is very common among projects in all object-oriented languages. We believe that our findings are not language specific, as similar code idioms exist in other programming languages too.

We also limit our study to only EXTRACT METHOD refactoring. There are 66 different types of refactoring [18] and EXTRACT METHOD refactoring is one of the most frequently performed refactoring types [12]. There are many motivations behind performing EXTRACT METHOD refactoring. According to Silva et al. [8] there are 11 motivations behind EXTRACT METHOD refactoring and the most common motivation is to make an extracted code fragment reusable. For these reasons, we focused our experiment on EXTRACT METHOD refactoring.

We have limited our experiment to 110 Java projects. We have specified certain acceptance criteria to ensure that the projects are mature, active and popular open-source projects. We also tried to include projects from various domains such as framework, database, network, cloud, and

Android domains. We believe our results and code idioms are not largely affected by the limited number of projects. However, we expect to have more instances for each code idiom, if we increase the number of projects.

7.2 Internal Validity

We are using RefactoringMiner 2.0 [21] to detect and collect EXTRACT METHOD refactoring instances from the commit history of Java GitHub projects. The validity of the EXTRACT METHOD instances included in the experiment depends on how accurately RefactoringMiner can detect this type of refactoring. We have a total of 65,742 instances of EXTRACT METHOD refactoring. We manually verified a very small number of instances by inspecting the commits, but not all of them. A Survey of Refactoring detection tools [47] shows that RefactoringMiner generally outperforms its competitors in detecting refactoring types. It has the highest average precision of 99.6% and recall of 94% among all competitive tools, and in particular for EXTRACT METHOD RefactoringMiner has 99.8% precision and 95.8% recall [21]. Therefore, we consider the EXTRACT METHOD refactoring instances reported by RefactoringMiner to be accurate for our study.

We are generating the Program Dependence Graph and Control Flow Graph for each extracted method. The binding information for the invoked methods, instantiated objects, and used fields and variables within the body of the extracted method is necessary information to generate these types of graphs from the source code. We have used the API Finder tool [33] to get the binding information for such cases. The tool doesn't compile the source code and generates the binding information by using the project's library dependency information. We may have missed parts of the graph or an entire graph for a given code fragment, due to an internal failure of the tool. The tool has some limitations and in a few cases, it may fail to generate binding information. We have added a fallback mechanism when the tool fails, but yet we cannot guarantee that we can generate our graph representation for all the extracted methods we have collected. This may lead us to miss some interesting code idioms or some instances of our catalogued code idioms.

For the frequent code idioms, we search across different projects, we consider only external APIs such as method invocations, class instantiations or field accesses. We do not consider any internal API usages to be part of the generated graph representation. We have also imposed a few more constraints for graph generation, which might have resulted in missing some interesting code idioms. We have added those constraints after careful investigation to avoid an enormous amount of matching results for unrelated code fragments.

Chapter 8

Conclusion

As a software product grows larger and more complex, identifying where and how to refactor becomes increasingly challenging. With more lines of code and dependencies between different components, it becomes hard to pinpoint specific areas of the codebase that require refactoring. Moreover, as the complexity of the system increases, it becomes harder to predict the impact of any given refactoring on other parts of the system. Refactoring has become a crucial aspect of software development in modern software engineering. Developers are urged to devote regular time and attention to refactoring their code and suggest refactorings during code reviews [48]. Nevertheless, given that developers must juggle various software development tasks, determining what aspects of the code to refactor can be challenging.

Consequently, we have sought to identify the most commonly refactored code idioms to assist in designing more effective refactoring recommendation systems in the future. Such systems could help developers more efficiently and accurately identify areas of the codebase that require refactoring. In this study, we focused on EXTRACT METHOD refactoring for Java open-source projects hosted in GitHub. Our motivation was to study and identify code idioms that developers tend to refactor with EXTRACT METHOD refactoring. We have analysed over 31 thousand commits from 110 projects to collect over 65 thousand EXTRACT METHOD refactoring instances. We have represented each EXTRACT METHOD code fragment as a GROUM graph and used a similarity-based graph isomorphism technique to find similarities between the code fragments. We have found 489 code idioms with frequency from 2 to 62 projects. We have presented 22 code idioms, which broadly cover the most frequent refactored code idioms. We run our experiment for both production and test code and found unique and shared code idioms among these.

Our results show that there are code idioms developers tend to refactor more frequently across different projects and different domains. Also, there are code idioms unique to production code and test code, along with common code idioms in both production and test code. Our experiment scope is limited to Extract Method refactoring, but the approach we followed for the representation and similarity measure can be applied to other types of refactorings involving a code fragment such as the Move Method, Inline Method, Pull Up Method and Push Down Method, and for other statement-level refactorings. This approach is also applicable to other object-oriented languages. One future work can involve finding code idioms for different types of refactoring and different programming languages.

Another future work would be to design and develop a refactoring recommender system with the help of the code idioms found in our experiment. The recommender system can locate similar code idioms in the codebase of a project and suggest refactorings. It can also use numerical information, such as the number of similar refactorings done in `X` number of projects `Y` times by other developers to convince the user about the usefulness of the suggested refactoring. The refactoring tool developers or IDEs can also use the identified code idioms to prioritise the type of refactoring for a given code fragment. Our findings can also help the researchers to understand what code fragments developers tend to refactor more frequently or the motivation of the developers when performing a particular type of refactoring. Our findings can help document usage patterns for a certain sequence of APIs or a group of APIs used to perform a specific task.

Bibliography

[1] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. Amsterdam The Netherlands: ACM, Aug 2009, p. 383–392. [Online]. Available: https://dl.acm.org/doi/10.1145/1595696.1595767

[2] E. Chikofsky and J. Cross, "Reverse engineering and design recovery: a taxonomy," *IEEE Software*, vol. 7, no. 1, p. 13–17, Jan 1990.

[3] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, p. 126–139, Feb 2004.

[4] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, Dec 2014.

[5] A. Arif and Z. Rana, "Refactoring of code to remove technical debt and reduce maintenance effort," Dec 2020, p. 1–7.

[6] M. Fowler, K. Beck, and J. Brant, "Refactoring - improving the design of existing code."

[7] K. Beck, "Embracing change with extreme programming," *Computer*, vol. 32, no. 10, p. 70–77, Oct 1999.

[8] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium*

on Foundations of Software Engineering, Nov 2016, p. 858–870, arXiv:1607.02459 [cs]. [Online]. Available: http://arxiv.org/abs/1607.02459

[9] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE Software*, vol. 25, no. 5, p. 38–44, Sep 2008.

[10] P. Weißgerber and S. Diehl, "Are refactorings less error-prone than other changes?" in *Proceedings of the 2006 international workshop on Mining software repositories*. Shanghai China: ACM, May 2006, p. 112–118. [Online]. Available: https://dl.acm.org/doi/10.1145/1137983.1138011

[11] R. Halepmollasi and A. Tosun, "Exploring the relationship between refactoring and code debt indicators," *Journal of Software: Evolution and Process*, vol. n/a, no. n/a, p. e2447, 2022.

[12] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *ECOOP 2013 – Object-Oriented Programming*, ser. Lecture Notes in Computer Science, G. Castagna, Ed. Berlin, Heidelberg: Springer, 2013, p. 552–576.

[13] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Silicon Valley, CA, USA: IEEE, Nov 2013, p. 180–190. [Online]. Available: http://ieeexplore.ieee.org/document/6693078/

[14] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, Nov 2014, p. 472–483. [Online]. Available: https://doi.org/10.1145/2635868.2635901

[15] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, "Does refactoring improve reusability?" in *Proceedings of the 9th international conference on Reuse of Off-the-Shelf Components*, ser. ICSR'06. Berlin, Heidelberg: Springer-Verlag, Jun 2006, p. 287–297. [Online]. Available: https://doi.org/10.1007/11763864_21

[16] M. Allamanis, E. T. Barr, C. Bird, P. Devanbu, M. Marron, and C. Sutton, “Mining semantic loop idioms,” *IEEE Transactions on Software Engineering*, vol. 44, no. 7, p. 651–668, Jul 2018.

[17] A. Tahmid, “Frequently refactored code idioms,” masters, Concordia University, Jun 2020. [Online]. Available: https://spectrum.library.concordia.ca/id/eprint/986902/

[18] [Online]. Available: https://refactoring.com/catalog/

[19] N. Tsantalis and A. Chatzigeorgiou, “Identification of extract method refactoring opportunities for the decomposition of methods,” *Journal of Systems and Software*, vol. 84, no. 10, p. 1757–1782, Oct 2011.

[20] J. Yamanaka, Y. Hayase, and T. Amagasa, “Recommending extract method refactoring based on confidence of predicted method name,” no. arXiv:2108.11011, Aug 2021, arXiv:2108.11011 [cs]. [Online]. Available: http://arxiv.org/abs/2108.11011

[21] N. Tsantalis, A. Ketkar, and D. Dig, “Refactoringminer 2.0,” *IEEE Transactions on Software Engineering*, vol. 48, no. 03, p. 930–950, Mar 2022.

[22] F. E. Allen, “Control flow analysis,” *ACM SIGPLAN Notices*, vol. 5, no. 7, p. 1–19, Jul 1970.

[23] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The program dependence graph and its use in optimization,” *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, p. 319–349, Jul 1987.

[24] M. Allamanis and C. Sutton, “Mining source code repositories at massive scale using language modeling,” in *2013 10th Working Conference on Mining Software Repositories (MSR)*, May 2013, p. 207–216.

[25] M. Acharya, T. Xie, J. Pei, and J. Xu, “Mining api patterns as partial orders from source code: from usage scenarios to specifications,” in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC-FSE ’07. New York,

NY, USA: Association for Computing Machinery, Sep 2007, p. 25–34. [Online]. Available: https://doi.org/10.1145/1287624.1287630

[26] H. A. Basit and S. Jarzabek, "A data mining approach for detecting higher-level clones in software," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, p. 497–514, Jul 2009.

[27] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, May 2018, p. 542–553. [Online]. Available: https://doi.org/10.1145/3196398.3196431

[28] B. Livshits and T. Zimmermann, "Dynamine: finding common error patterns by mining software revision histories," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, p. 296–305, Sep 2005.

[29] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, p. e1838, 2017.

[30] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," *ACM SIGPLAN Notices*, vol. 43, no. 10, p. 313–328, Oct 2008.

[31] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," vol. 12, no. 1, 1990.

[32] H. Zhong and X. Wang, "Boosting complete-code tool for partial program," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct 2017, p. 671–681.

[33] D. Dam, "API Finder." [Online]. Available: https://github.com/diptopol/apifinder/tree/master

[34] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *2008 12th European Conference on Software Maintenance and Reengineering*, Apr 2008, p. 329–331.

[35] S. Voss and J. Subhlok, "Performance of general graph isomorphism algorithms."

[36] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM*, vol. 23, no. 1, p. 31–42, Jan 1976.

[37] T. Reza, H. Halawa, M. Ripeanu, G. Sanders, and R. A. Pearce, "Scalable pattern matching in metadata graphs via constraint checking," *ACM Transactions on Parallel Computing*, vol. 8, no. 1, pp. 2:1–2:45, Jan 2021.

[38] L. Chen, S. Wang, and X. Yan, "Centroid-based clustering for graph datasets," in *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*, Nov 2012, p. 2144–2147.

[39] L. Galluccio, O. Michel, P. Comon, and A. O. Hero, "Graph based k-means clustering," *Signal Processing*, vol. 92, no. 9, p. 1970–1984, Sep 2012.

[40] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, p. 858–868.

[41] M. B. Uddin (Palash), "Refactoringmatcher," Nov 2021. [Online]. Available: https://github.com/palashborhanuddin/RefactoringMatcher

[42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[43] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for msr studies," 2021.

[44] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *27th European Conference on Object-Oriented Programming*, ser. ECOOP'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 552–576. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39038-8_23

[45] D. Silva, J. P. da Silva, G. Santos, R. Terra, and M. T. Valente, "Refdiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, p. 2786–2802, Dec 2021.

[46] “Neo4j graph data platform – the leader in graph databases.” [Online]. Available: https://neo4j.com/

[47] L. Tan and C. Bockisch, “A survey of refactoring detection tools,” *Living Systems*.

[48] F. Coelho, N. Tsantalis, T. Massoni, and E. L. G. Alves, “An empirical study on refactoring-inducing pull requests,” in *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM ’21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3475716.3475785