PurityChecker: A Tool for Detecting Purity of Method-level Refactoring Operations

Pedram Nouri

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

November 2023

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Pedram Nouri**

Entitled: **PurityChecker: A Tool for Detecting Purity of Method-level Refactoring Operations**

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

Master of Applied Science (Computer Science)

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

Signed by the Final Examining Committee:

_____ Chair
Dr. Shin Hwei Tan

_____ Examiner
Dr. Jinqiu Yang

_____ Examiner
Dr. Shin Hwei Tan

_____ Supervisor
Dr. Nikolaos Tsantalis

Approved by _____
Dr. Leila Kosseim, Chair
Department of Computer Science and Software Engineering

_____ 2023

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

Abstract

PurityChecker: A Tool for Detecting Purity of Method-level Refactoring Operations

Pedram Nouri

Software refactoring is a vital practice in software engineering, aiming to improve code quality and maintainability. However, different refactoring instances serve different purposes. Some are purely intended to enhance code health by preserving program behavior, while others serve to eliminate defects and enable new functionality. Determining the purity of a refactoring instance, whether it is behavior-preserving (a.k.a. "pure") or not, is an essential but often challenging task. This research introduces PurityChecker, a novel tool designed to automatically detect the purity of method-level refactoring instances in Java code through static source code analysis.

The contributions of this thesis are twofold. First, a tool has been created to assess the purity of refactoring instances, enabling the investigation of refactoring purity at a large scale. Extensive evaluations based on manually validated oracles demonstrated PurityChecker's effectiveness, with precision and recall exceeding 90% in most cases. Second, a large-scale empirical study was conducted, resulting in two meticulously validated oracles, providing invaluable resources for research in software refactoring.

PurityChecker opens up new avenues for research and development. Its potential applications range from improving code reviewing processes and code quality maintenance to enabling empirical studies that leverage the concept of refactoring purity. This work is not only an important contribution to the field of software evolution analysis, but also a stepping stone for future research into refining and expanding refactoring assessment techniques.

Acknowledgments

I would like to express my heartfelt appreciation and deep gratitude to my supervisor, Prof. Nikolaos Tsantalis. It was an absolute delight to work alongside him in a collaborative and highly productive manner. His invaluable mentorship and unwavering support played a pivotal role in helping me overcome the challenges I encountered during my research.

Furthermore, I would like to express my thanks to my colleagues, Pourya Alikhani Fard, Mosabbir Khan Shiblu, and Mohammad Sadegh Alizadeh. Their generosity in sharing their invaluable experiences and their constant support throughout my research journey have been instrumental in shaping the outcome of this thesis.

Finally, I would like to extend my sincere appreciation to my family for their unwavering support during the challenging times. Their constant encouragement and belief in me have been invaluable, and I am deeply grateful for their presence in my life.

Thank you.

Pedram Nouri

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Refactoring is a widely adopted technique among software developers that aims to enhance the overall design quality of the source code while preserving the original program behavior. It is a frequently utilized along with other software maintenance activities, such as eliminating faults, bugs, and defects within the software. As a result, the system becomes more comprehensible and manageable, leading to reduced costs and efforts during the maintenance phase (Agnihotri & Chug, 2020, 2022). As numerous studies suggested, software refactoring is one of the most important parts of software evolution as it can improve the cohesion, maintainability (Kolb, Muthig, Patzke, & Yamauchi, 2005), evolvability (Ratzinger, Fischer, & Gall, 2005), and reusability (Moser, Sillitti, Abrahamsson, & Succi, 2006) of existing software systems. In a close-to industrial, agile development environment, refactoring is a key practice and allows for the progression of software with limited initial design (Sillitti & Succi, n.d.).

In the literature, two refactoring strategies have been observed, known as "floss refactoring" and "root canal refactoring", which serve to describe the objectives behind the application of refactoring operations. The floss refactoring involves making additional program modifications, such as fixing bugs, adding new features or modifying existing ones, alongside the refactoring process. It is commonly used to maintain the overall health and quality of the code. On the other hand, developers apply root canal refactoring with a specific focus on enhancing the source code quality and maintainability. Root canal refactoring refers to the type of refactoring that is performed on software once it has become unhealthy or difficult to maintain (E. Murphy-Hill & Black, 2007, 2008).

Each refactoring type involves specific mechanics that help developers safely modify code while preserving program behavior (Fowler, 2018). Following these mechanics ensures that the refactoring is correct and does not introduce bugs or unintended changes. However, there are certain modifications that may occur during a refactoring process that are not directly related to the mechanics of that specific refactoring type. Despite this, the overall refactoring can still be considered behavior-preserving as long as the intended behavior of the code remains unchanged. For example, during a complex refactoring like PULL UP METHOD, there might be other smaller changes like an EXTRACT VARIABLE refactoring within it. Although the change caused by the EXTRACT VARIABLE refactoring is not directly part of the mechanics of PULL UP METHOD, the entire process is still considered behavior-preserving as long as the final behavior of the code remains consistent.

There is an abundance of research and practical tools available that can effectively detect refactorings between two versions of code. However, the number of tools that can determine whether a refactoring instance is pure or not is quite limited. Determining whether a refactoring is pure or not is crucial in various scenarios and studies, such as manual refactoring validation (X. G. E. Murphy-Hill, n.d.), refactoring-aware code review tools (Alves, Song, & Kim, 2014), empirical studies investigating the usage of automated refactoring tools (Black & Murphy-Hill, 2007), and assessing the impact of refactoring on the internal quality of code (Chávez, Ferreira, Fernandes, Cedrim, & Garcia, 2017).

In this study, our objective is to define and automatically assess **Refactoring Purity**. A refactoring is considered pure if it does not include any modifications that alter the behavior of the refactored code. It is essential to note that a single refactoring operation can have an impact on multiple program elements simultaneously. Therefore, when assessing refactoring purity, we refer to the set of elements directly affected by the specific refactoring operation.

We have created a tool named PurityChecker that can automatically determine whether a refactoring is pure or not. This tool relies heavily on RefactoringMiner by Tsantalis et al. (Tsantalis, Ketkar, & Dig, 2020), which is the state-of-the-art refactoring detection tool. To make this determination, PurityChecker utilizes statement mapping information and the list of refactorings happened in the two code revisions provided by RefactoringMiner and analyzes the replacements that occur between the mapped statements through static source code analysis. This process allows us to infer

whether a refactoring meets the criteria for being classified as pure or impure.

1.1 Motivation

1.1.1 Code Reviewer Perspective

Code and change understanding is the key aspect of code reviewing (Bacchelli & Bird, 2013). The ability to determine whether a refactoring is pure or impure can significantly benefit code reviewers in comprehending the code modifications. Specifically, as purity indicates whether a refactoring preserves the behavior of the code or not, having this knowledge during the code review process is undoubtedly advantageous. Having this insight empowers code reviewers to better grasp the implications and consequences of the code changes being made, thereby enhancing the overall review process.

In this section, we present an illustrative scenario that underlines the significance of refactoring purity in aiding the reviewer's comprehension of code changes. The scenario is depicted in Figure 1.1, which showcases an actual example from the ratpack project[1], slightly modified for the sake of simplicity. The code revision involves two distinct refactorings that account for the code transformation. Firstly, an EXTRACT METHOD refactoring (indicated by the black arrow) is observed, wherein lines 7 to 13 are extracted into a new method named `foo`. Secondly, an INLINE VARIABLE refactoring (indicated by the red arrow) takes place, leading to the inlining of the `lConFut` variable, and the deletion of the variable declaration (line 8) in the next version.

Upon the validation of this case, it becomes evident that there are no alterations to the functionality of this code, although the changes might not seem behavior-preserving in the first sight. It is vital to discern and analyze these changes in behavior to gain a comprehensive understanding of the code alterations. The ability to identify and classify such refactorings as "pure" becomes invaluable in this context, as it assures the reviewer that the code changes are behavior-preserving despite the apparent changes involved.

The INLINE VARIABLE refactoring occurs within the EXTRACT METHOD refactoring. Notably,

[1] GitHub commit link

Figure 1.1: An Example of Overlapping Refactorings Caused a Pure Code Modification

this combination of refactorings does not alter the program behavior, given that, concerning the INLINE VARIABLE refactoring, the variable initializer in the previous version remains the same after the inlining process. Additionally, the EXTRACT METHOD refactoring itself introduces no changes that affect the program behavior in the extracted statements. The only potentially concerning aspect lies in the alterations to the statement mapping between line 10 and 13 in the parent and child versions, respectively. By leveraging information from these two refactorings, alongside the statement mapping and replacement data provided by RefactoringMiner, our objective is to develop a tool capable of identifying and classifying the aforementioned EXTRACT METHOD refactoring as a pure instance. This will provide reassurance to reviewers that the code modification is behavior-preserving.

Undoubtedly, the provided example is simplistic and merely serves as a motivating scenario. It may be contended that for a proficient code reviewer, recognizing such a pure refactoring would be straightforward, rendering an automated tool unnecessary. However, we have encountered countless intricate scenarios where comprehending code changes necessitates the aid of an automated tool. These situations have reinforced the need for a supporting tool that can automatically detect the purity of refactoring, as it can greatly enhance our understanding of the code changes being made.

1.1.2 Developer Perspective

Recent research has indicated that many software developers prefer manual code refactoring over automated tools (Negara, Chen, Vakilian, Johnson, & Dig, 2013; Vakilian et al., 2012). However, manual refactorings are prone to errors. A study by Kim, Zimmermann, and Nagappan (2012) shows

Figure 1.2: Parameterization mistake in manually applied EXTRACT METHOD refactoring.

that developers often struggle to perform manual refactorings accurately. Additionally, Weißgerber and Diehl (2006) highlight that incorrect refactorings can lead to various types of bugs. PurityChecker can serve as a tool that informs developers about unintentional mistakes done during manual refactoring efforts that change program behavior.

For instance, in commit[2], the developer extracts many instances of duplicated code concerning different sorting algorithms into `runTest()` in class `SortsTiming`. The refactoring essentially parameterizes the sorting algorithm with the `Testable` parameter. However, because the refactoring is done manually, the developer makes a mistake. She does not replace the algorithm "`InsertionSort`" with parameter "`testable`" in all places in the extracted method.

In Figure 1.2, the orange-highlighted references to the sorting algorithms `InsertionSort`, `BubbleSort` should have been parameterized in the extracted method `runTest()`. Unfortunately, during this code transformation, the developer overlooked replacing the red-highlighted references, which resulted in a non-behavior-preserving EXTRACT METHOD refactoring.

The incorrect manual refactoring is fixed in a later commit[3], with the commit message "Fixed a small mis-type in sort timing code" by replacing "`InsertionSort`" with parameter "`testable`". In Figure 1.3, we can see the corrective commit made by the developer. Upon realizing the mistake, the developer replaced the references highlighted in red with the ones highlighted in green. PurityChecker would be able to warn the developer about non-behavior-preserving manual refactoring

[2] GitHub commit link
[3] GitHub commit link

```
insertionCount = runTest(new Insertion(), unsorted, sorted, insertionResults, insertionCount);
bubbleCount = runTest(new Bubble(), unsorted, sorted, bubbleResults, bubbleCount);

private static final int runTest(Testable testable,
 Integer[] unsorted, Integer[] sorted, double[] results, int count) {
    // ...

    result = InsertionSort.sort(sorted.clone());
    if (checkResults && !check(result))
        System.err.println(testable.getName()+" failed.");

    aInsertion = System.nanoTime();
    diff = (aInsertion - bInsertion) / 1000000d / 1000d;
    System.out.println("Sorted: "+testable.getName()+"=" + FORMAT.format(diff) + " secs");
    // ...
    result = InsertionSort.sort(reverse.clone());
    if (checkResults && !check(result))
        System.err.println(testable.getName()+"InsertionSort.");

    // ...
}
```

```
insertionCount = runTest(new Insertion(), unsorted, sorted, insertionResults, insertionCount);
bubbleCount = runTest(new Bubble(), unsorted, sorted, bubbleResults, bubbleCount);

private static final int runTest(Testable testable,
 Integer[] unsorted, Integer[] sorted, double[] results, int count) {
    // ...

    result = testable.sort(sorted.clone());
    if (checkResults && !check(result))
        System.err.println(testable.getName()+" failed.");

    aInsertion = System.nanoTime();
    diff = (aInsertion - bInsertion) / 1000000d / 1000d;
    System.out.println("Sorted: "+testable.getName()+"=" + FORMAT.format(diff) + " secs");
    // ...
    result = testable.sort(reverse.clone());
    if (checkResults && !check(result))
        System.err.println(testable.getName()+" failed.");

    // ...
}
```

Figure 1.3: Commit correcting manually applied EXTRACT METHOD refactoring.

before committing the refactoring to the repository.

The main motivation for developing tools to assess the purity of refactorings is to validate manual refactoring efforts. Some existing tools attempt to assess the correctness of refactorings by identifying non-behavioral code changes, but their definition of purity is incomplete. In this thesis, our goal is to expand the definition of purity to encompass a wider range of genuinely behavior-preserving refactoring operations. It is crucial to emphasize that the research tools we have encountered so far would inaccurately classify the EXTRACT METHOD refactoring, illustrated in Figure 1.1, as impure. Detailed explanations of these tools are provided in Section 2.2.

1.1.3 Researcher Perspective

The presence of a tool capable of reporting the purity of refactorings opens up a multitude of research avenues. The tool's ability to assess the purity of refactorings becomes particularly relevant as it aligns closely with the concept of floss vs. root canal refactoring operations. As a result, researchers are incentivized to conduct various empirical studies that leverage the notion of refactoring purity, exploring its implications and applications in diverse contexts within software development.

Although several refactoring detection tools have been developed and tested using different refactoring oracles, none of these oracles provide any information regarding the purity of the refactorings. To address this gap and further advance refactoring research, we created two refactoring datasets with instances found in open source projects. For each refactoring instance we manually validated and documented its purity. These datasets were used to compute the precision and recall of our tool.

1.1.4 Application in Regression Testing

Regression testing ensures that recent modifications to a project do not disrupt previously functioning features. While crucial, it becomes expensive when changes occur frequently. Regression test selection (RTS) addresses this by selectively executing tests whose outcomes could be impacted by a given change.

Numerous studies propose diverse approaches for Regression Test Selection (RTS). Having a tool capable of determining whether a refactoring is pure or not can significantly benefit this field. Predicting a refactoring as pure enables the skipping of tests related to the changes influenced by these refactorings. This practice can profoundly impact regression testing by substantially reducing the number of tests that need to be executed. Since pure refactorings preserve the program's behavior, changes related to them do not require retesting.

In a study by Wang et al. (Wang et al., 2018), the authors introduced refactoring-aware regression test selection. They argue that changes associated with refactorings are behavior-preserving, and tests covering these changes can be skipped. While this holds true for refactorings deemed behavior-preserving, it may not always be the case. Impure refactorings modify the program's behavior, necessitating testing for changes within this category during regression testing. PurityChecker can greatly assist this work by focusing their study on pure refactoring operations, aiding in the selection of tests to be excluded from regression testing.

1.1.5 Limitation of Existing Tools

As previously mentioned, the current tools and methods available are unable to recognize the straightforward example illustrated in Figure 1.1 as a pure refactoring operation. Several factors contribute to this limitation, which we summarize as follows:

- **Inability to handle changes resulting from overlapping refactorings**: Existing approaches struggle to effectively deal with situations where multiple refactorings overlap, leading to challenges in accurately assessing the purity of the refactoring.
- **Lack of accurate statement mapping and AST node replacement information**: The existing approaches use inadequate methods for obtaining a fine-grained and accurate source code

diff. As a result, they fail to assess numerous behavior-preserving refactorings as pure ones. This limitation hinders the proper recognition of genuinely behavior-preserving transformations, leading to missed opportunities for classifying such refactorings correctly.

PurityChecker's primary objective is to overcome the aforementioned limitations and introduce novel techniques to effectively and precisely identify pure refactoring cases, even in complex scenarios. By addressing these existing shortcomings and incorporating innovative approaches, PurityChecker aims to offer a more comprehensive and accurate assessment of refactorings, ensuring that behavior-preserving transformations are correctly identified and categorized, even in challenging situations. In contrast to previous tools, our objective is to introduce an extended refactoring purity catalog that incorporates a broader spectrum of refactorings recognized as pure cases.

1.2 Objectives and Contributions

In this thesis, we introduce a tool called PurityChecker, which is designed to assess whether a refactoring operation is pure or not. To make this determination, the tool analyzes information about refactorings that occurred in a specific code revision. It relies on information from RefactoringMiner 2.0 (Tsantalis et al., 2020), which provides details about statement mapping and replacements during the refactoring process. PurityChecker not only considers the information about the refactorings but also takes into account the changes caused by the mechanics of each specific refactoring type. By performing static source code analysis, PurityChecker can infer whether a refactoring preserves the behavior of the code or not. Moreover, PurityChecker generates a concise report explaining the reasons behind its purity assessment.

This thesis makes the following primary contributions:

- It introduces a novel tool capable of identifying pure and impure refactoring operations in Java code through static source code analysis. This tool is the first of its kind to offer such functionality.
- To validate the tool's effectiveness, the thesis conducts a manual analysis of two separate oracles—one for training and one for testing. These oracles consist of a total of 700 commits.

The training oracle is sourced from RefactoringMiner 2.0 (Tsantalis et al., 2020) dataset, while the testing oracle is sourced from a different study conducted by Pantiuchina et al. (Pantiuchina et al., 2020).

- The thesis presents and discusses the empirical results obtained from the manual analysis of the commits. These results shed light on the tool's performance and the characteristics of refactoring operations in real-world scenarios.

1.3 Outline

The remaining sections of this thesis are structured as follows. Chapter 2 provides a comprehensive overview of related tools and studies that have explored the concept of code modification purity. Chapter 3 outlines the methodology used to develop PurityChecker, the novel tool for detecting pure and impure refactoring operations, explaining its design and implementation. Chapter 4 presents the outcomes of a large-scale study conducted using PurityChecker, including the tool's performance evaluation and the results obtained from the manual analysis. Chapter 5 addresses the threats to both internal and external validity in the study, highlighting potential challenges or limitations in the research. Finally, Chapter 6 concludes the thesis by summarizing the key findings, discussing their implications, and proposing potential avenues for future research in this area.

Chapter 2

Literature Review

In this chapter, we begin with a broad perspective by investigating various research areas that are relevant to software refactoring. Subsequently, we delve into a more detailed examination of research studies and tools that focus on the concept of refactoring purity.

The term *refactoring* was initially introduced in (Opdyke, 1990), where it was defined as any transformation applied to the source code with the aim of improving codebase reusability and understandability, and potentially being automated. Since then, numerous other research endeavors have explored different facets of refactoring activity. These include researches in the refactoring process itself, how developers refactor (E. Murphy-Hill, Parnin, & Black, 2011), the usage of refactoring tools (E. Murphy-Hill et al., 2011; Negara et al., 2013), the motivations driving refactoring (Kim et al., 2012; Kim, Zimmermann, & Nagappan, 2014; Pantiuchina et al., 2020; Silva, Tsantalis, & Valente, 2016), the risks of refactoring (Bavota et al., 2012; Kim, Cai, & Kim, 2011; Kim et al., 2012, 2014; Weißgerber & Diehl, 2006), the impact of refactoring on code quality metrics (Chávez et al., 2017; Vashisht, Bharadwaj, & Sharma, 2018), and various other related aspects.

Kaya, Conley, Othman, and Varol (2018) examine situations in which refactoring becomes necessary to ensure the maintainability and cleanliness of the codebase. Additionally, they highlight the importance of integrating refactoring information into third-party tools, particularly code visualization tools.

The impact of refactoring activity on software internal quality attributes have been studied by Chávez et al. (2017). This study differentiates between two types of refactoring tactics, namely

root canal and floss refactoring, based on the varying emphasis on code structural quality for each refactoring tactic. The authors of the study focused on five distinct internal quality attributes that are closely associated with the specific refactoring types under investigation. It can also lead to enhancements in external software quality attributes, which are indirectly evaluated through code metrics such as maintainability and understandability (Vashisht et al., 2018).

Ouni et al. (2017) address the under-utilization of automated refactoring tools and present a novel automated approach called MORE for the refactoring recommendation task. The primary objectives of their approach are threefold: (1) to enhance design quality based on software quality metrics, (2) to fix code smells, and (3) to incorporate design patterns.

2.1 Refactoring Mining Tools

Refactoring operations can be performed either manually or with assistance from automated tools available within Integrated Development Environments (IDEs) or external software. Automatically detecting the refactorings that occurred between two program versions is a significant research challenge. This is crucial because identifying these refactorings can aid developers in comprehending the changes made by other developers. Furthermore, refactorng detection can be utilized to update client applications that may have been affected by refactorings performed in library components, helping to address potential issues and maintain software compatibility (Dig, Comertoglu, Marinov, & Johnson, 2006; Henkel & Diwan, 2005; Xing & Stroulia, 2007). Numerous researchers have developed refactoring detection tools capable of automatically detecting various refactoring types in the change history of software systems. These tools employ various algorithms and methodologies to achieve high precision and recall in their detection capabilities. In this section, we provide a concise overview of the most recent and successful refactoring detection tools.

RefactoringCrawler, developed by (Dig et al., 2006), utilized the approach of record-and-replay of refactorings. This tool aimed to help refactoring engines to re-apply previously detected refactorings to newer versions more efficiently. Their refactoring detection algorithm employs a

two-step approach, incorporating a rapid syntactic analysis to identify potential refactoring candidates and a more complex and expensive semantic analysis to further refine and validate the results. Their syntactic analysis utilizes Shingles encoding, a technique from Information Retrieval, to swiftly identify similar fragments in text files. By applying shingles to source files, the algorithm efficiently detects refactorings that involve repartitioning of the source code, leading to similar text fragments across different versions of a component. On the other hand, their semantic analysis relies on reference graphs, which represent the connections between source-level entities, such as method calls. This analysis takes into account the semantic relationship between candidate entities to determine whether they constitute a refactoring.

In **REF-FINDER** (Prete, Rachatasumrit, Sudan, & Kim, 2010), the authors tried to address the main limitations found in previous refactoring detection tools, which can be summarized into the difficulties in automatically identifying certain types of refactorings that involve changes to program structure, method implementations, and combinations of multiple refactorings. These challenges may require manual intervention or more sophisticated analysis techniques to be properly detected and applied. Inspired by the concept of logic-based program representation, the authors utilized template logic rules and encoded dependencies among refactoring types to define structural constraints before and after applying refactorings to a program. They developed a fact extractor to gather information about code elements, structural dependencies, and code content, represented as a database of logic facts. The tool then inferred specific refactoring instances by converting template logic rules into logic queries and executing them on the database. Essentially, they used a logic-based approach to identify refactorings automatically based on the extracted facts from the program's abstract syntax tree.

RefDiff (Silva & Valente, 2017) utilizes a blend of heuristics based on static analysis and code similarity to identify refactorings between two versions of a system. The tool takes two system revisions as input and generates a list of detected refactorings as output. At its core, RefDiff treats code fragments as a collection of tokens using a modified version of the TF-IDF approach, seeking out similar code elements by applying a similarity threshold. The detection algorithm comprises two main phases: Source Code Analysis and Relationship Analysis. In the first phase, the algorithm parses and examines the system's source code, creating models to represent high-level source code

entities such as types, methods, and fields before and after the changes. To optimize efficiency, only the code entities from modified source files are analyzed. In the second phase, the algorithm focuses on establishing relationships between the source code entities before and after the changes. This is accomplished by constructing a bipartite graph with two sets of vertices representing code entities before and after the changes, with the graph's edges symbolizing the relationships between these entities. In a subsequent work by (Silva, da Silva, Santos, Terra, & Valente, 2020), **ReffDiff 2.0** was introduced, which shares similarities with its predecessor at its core. This is the first multi-language refactoring detection tool, which supports JavaScript and C programming languages, in addition to Java.

(Tsantalis, Mansouri, Eshkevari, Mazinanian, & Dig, 2018) developed, implemented, and evaluated **RMiner**, a tool that addresses two key limitations observed in many existing refactoring detection tools. Firstly, RMiner overcomes the need for a predefined similarity threshold, which can be challenging to determine in different scenarios. Secondly, unlike other tools, RMiner does not require two fully built versions of a software system for refactoring detection. Instead, it accepts two code revisions, typically a commit and its parent from the commit history in git-based version control repositories, and then produces a list of refactorings that occurred between these revisions. To achieve this, RMiner utilizes two major techniques: Abstraction and Argumentization. Abstraction handles changes in Abstract Syntax Tree (AST) types caused by refactorings, while Argumentization deals with changes in sub-expressions within statements due to parameterization. By employing an AST-based algorithm for statement matching along with these novel techniques, RMiner successfully identifies 14 high-level refactoring types with relatively high precision and recall rates.

In a more recent work (Tsantalis et al., 2020), RMiner has been extended to **RefactoringMiner 2.0** [1]. The primary enhancement in this new version, as highlighted by the authors, lies in its matching function, which incorporates additional replacement types and heuristics. To assess the tool's performance, the authors conducted an evaluation by comparing it with existing tools, including its predecessor RMiner and RefDiff 2.0. They utilized a dataset comprising 7,226 confirmed instances of 40 distinct refactoring types, all validated by experts. The results demonstrated the superiority

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

of the updated RefactoringMiner 2.0, as it achieved the highest precision (99.6%) and recall (94%) among all the evaluated tools. These findings emphasize the effectiveness of the new version in accurately detecting and identifying various refactoring operations in software code.

2.2 Refactoring Purity

As previously mentioned, determining whether a refactoring is pure or not holds significant advantages in various scenarios. This information provides valuable insights to code reviewers, helping them understand the nature and impact of the applied refactorings. Additionally, the ability to identify pure refactorings has numerous applications in empirical studies, enabling researchers to gain deeper understanding and make informed assessments about the applied refactorings on the codebase.

While the literature on different aspects of refactoring activity encompasses numerous research papers and tools, there is a limited number of studies that specifically focus on the detection of pure and impure refactorings. In this section, we conduct a thorough analysis of these few studies to gain a deeper understanding of this particular field of research. By examining these works in detail, we aim to explore the advancements and challenges associated with identifying whether refactorings are behavior-preserving or not, thus contributing to the existing knowledge in this specialized area.

In the study by (Kawrykow & Robillard, 2011), a novel term, "non-essential modification" is introduced and defined. The authors have developed a tool specifically designed to identify instances of non-essential code differences present within the version histories of software systems. These differences encompass a range of changes, including local variable refactorings and textual disparities resulting from rename refactorings. Although the primary focus of their investigation is not on changes occurring within refactorings, it holds a substantial connection to the concept of purity. Their intention is rooted in the aim of achieving a more meaningful representation of code changes, with the goal of discerning and highlighting modifications that may not significantly affect the overall behavior of the software.

Initially, they established distinct criteria that outline non-essential modifications in code revisions, encompassing five primary categories: (1) Trivial Type Updates, (2) Rename-Induced Modifications, (3) Trivial Keyword Modifications, (4) Local Variable Renames, and (5) Whitespace and Documentation-Related Updates. As their research is not dependant on any existing refactoring detection tool that offers information about changes occurring within statements and expressions, the authors developed their proprietary change analysis tool, termed **DIFFCAT**. DIFFCAT processes a set of committed source files obtained from a software repository (a change set) and provides a detailed account of the diverse structural modifications present in that particular change set. Their detection method utilizes an AST-based Partial Program Analysis to facilitate type resolution, enabling more accurate identification and categorization of structural changes that are deemed non-essential. During the validation phase, they assessed the precision of their technique and discovered that 98.8% of the method updates identified as non-essential were correctly classified. While DIFFCAT is accessible online, we refrained from comparing it to PurityChecker due to the distinct scopes of these tools, rendering them incomparable. A detailed explanation of this decision is provided in Section 4.1.2.

In (Alves et al., 2014), the authors introduced a specialized tool named **RefDistiller**, designed explicitly for validating manual refactoring activities during code reviews. Unlike general code review tools, RefDistiller is tailored to identify potential behavioral changes introduced by manual refactoring edits. Notably, this paper includes the first mention of the concept of "purity" in the context of refactoring operations. RefDistiller is specifically focused on six of the most prevalent method-related refactoring types, which include move method, extract method, inline method, rename method, pull up method, and push down method.

The fundamental approach of RefDistiller centers around recognizing discrepancies between a manually refactored code and its corresponding pure refactoring counterpart. The tool comprises three core components:

- RefFinder: This refactoring detection component takes an original version and a manually refactored version as inputs, deducing potential refactoring types and their locations automatically. Note that a detailed description of this particular refactoring detection tool is provided in section 2.1.

- RefChecker: For each identified refactoring edit, RefChecker examines a set of essential code modifications. It validates the preservation of method or field reference bindings using predefined template rules. These rules mainly consider the validation of bindings and missing components (statements and methods) and rename-induced changes. If expected changes are absent in the actual edits, these omissions are highlighted as deviations from pure refactoring, raising concerns about potential behavior changes.

- RefSeparator: This module applies an equivalent pure refactoring using a modified version of the Eclipse refactoring engine to generate a pure refactoring version. This version is then compared to the manually refactored one using ChangeDistiller's syntactic differencing technique (Fluri, Wursch, PInzger, & Gall, 2007). If discrepancies arise between the versions, RefSeparator pinpoints the locations of additional edits, which are again flagged as potential alterations to the program's behavior.

According to their evaluation, RefDistiller can correctly identify 97% of the erroneous refactoring edits. We made the decision to access the tool in order to conduct a more thorough evaluation of its strengths and weaknesses. Unfortunately, the tool is not available through the links referenced in the paper or on GitHub.

In their work (X. G. E. Murphy-Hill, n.d.), a novel technique named **GhostFactor** is proposed. It introduces an innovative static analysis approach that identifies and validates manually performed refactorings. The tool is specifically designed to address a limited set of refactoring types, namely Extract Method, Change Method Signature, and Inline Method. The study's main motivation arises from the significant under-utilization of automated refactoring tools, which often results from developers' lack of trust in their reliability and potential to introduce errors. GhostFactor comprises distinct components, namely history preservation, refactoring detection, condition assessment, and refactoring warning functionality.

The history preservation component is responsible for monitoring and storing the alteration history of various files. The refactoring detection module identifies refactorings carried out manually by developers. It takes a snapshot list of a source file as input, dynamically loading available refactoring detectors and applying them to the snapshot list. Developers can specify these refactoring

detectors. The condition assessment component dynamically incorporates condition checkers related to the specific refactoring type. These checkers assess both pre-conditions and post-conditions for the given refactoring, as proposed by Opdyke (Opdyke, 1992). Lastly, the refactoring warning element tracks identified condition violations and reports them to the user.

Of particular interest is the condition assessment component, which closely aligns with the concept of refactoring purity. This component determines whether a manually-detected refactoring maintains the behavior of the transformed code or not. It examines whether a particular manual refactoring adheres to a predefined set of conditions. GhostFactor dynamically integrates condition checkers during runtime, allowing developers to easily incorporate new custom checkers. The study outlines four primary conditions that need to be fulfilled to classify a refactoring as behavior-preserving: (1) Return Value Checker, (2) Parameter Checker, (3) Stale Invocation Checker, and (4) Modified Variable Checker.

In their assessment, the tool evaluated through a human study involving eight expert software developers. The results indicated that GhostFactor enhanced the accuracy of manual refactorings by 67%. Even though the tool is intended for validating manual refactorings carried out in the C# programming language, we opted to examine it more closely to analyze the tool's features and study their approach. Unfortunately, the tool is not available for installation either as Visual Studio plugins or on GitHub.

2.3 Limitation of the Existing Approaches

In this section, we highlight several notable constraints within reviewed methodologies and studies, which we aim to address in our proposed approach.

Reliance on Predefined Purity Rules without further statement mapping and replacement analysis: A significant limitation encountered in several tools, whether their primary objective is to identify pure refactorings or this task is part of their intent, is their dependence on predefined conditions without extensive analysis of different scenarios in which the rules can be refined. Developers often undergo shifts in their programming decisions and ideas. Establishing fixed rules for these situations is challenging, unless a thorough analysis of extensive datasets and oracles is conducted.

However, without such extensive analysis, these conditions and predefined rules lack universality and may not be applicable to every conceivable refactoring scenario due to the varied and sometimes unpredictable refactoring practices adopted by developers during the evolution of software systems. Relying solely on these predefined strict rules, without incorporating comprehensive statement mapping and replacement analysis, leads to a substantial number of false negative predictions. In these cases, refactoring operations are erroneously flagged as impure, even though they genuinely preserve the software's behavior.

Inability to Address Overlapping Refactoring Scenarios: A notable limitation within many existing tools and methodologies is their incapacity to handle scenarios involving overlapping refactoring activities. Many instances of refactoring involve the stacking of two or more refactorings on top of each other. Regrettably, the majority of the tools mentioned earlier do not adequately consider the potential changes introduced by overlapping refactoring operations. In various instances, such overlapping refactorings result in behavior-preserving alterations to the codebase. Given the vast array of possible refactoring combinations, rule-based refactoring purity detection tools would need an impractical number of rules and conditions to encompass the effects of overlapping refactorings.

As exemplified earlier in Figure 1.1, a code modification features two refactorings occurring on top of one another. Almost all the tools and approaches aimed at identifying pure refactorings are likely to miss scenarios like this one, where overlapping refactorings take place. As a consequence, they would mistakenly fail to correctly identify such cases as examples of pure refactoring.

Under-utilization of Statement Mapping and Replacement Analysis: Most of the tools and techniques for detecting purity do not harness the valuable insights offered by statement mapping and replacement analysis. Given that a multitude of changes occur at the statement and expression level, neglecting to incorporate this critical analysis leads to a significant loss of valuable information. This omission can hinder the ability to effectively rationalize various modifications made within the codebase.

Poor refactoring detection tools: A fundamental prerequisite for accurately identifying pure refactorings is the precise detection of refactorings themselves. Regrettably, most current methodologies suffer from the relatively low precision and recall rates of their refactoring detection components, which affects negatively the purity detection process.

2.4 PurityChecker Improvements over Existing Approaches

PurityChecker stands out as an advancement over existing tools like RefDistiller and GhostFactor by broadening the scope of accepted changes within the statement mappings of method-level refactorings. Unlike its counterparts, PurityChecker goes beyond supporting only trivial changes within the method body, offering a more comprehensive approach.

While RefDistiller and GhostFactor are limited to handling straightforward alterations in the body of method-level refactorings, PurityChecker's strength lies in its extensive manual validation and specific design for purity detection. This enables PurityChecker to establish robust and general rules that cover a wide range of changes within method bodies. For instance, GhostFactor can handle changes resulting from a RENAME METHOD refactoring but lacks the ability to justify modifications in a statement mapping affected by multiple RENAME METHOD refactorings in a single commit due to the absence of fine-grained AST-based replacement information. As we know, there could be more than one method invocation within an statement mapping that have been affected by several RENAME METHOD refactorings in a commit.

Furthermore, both tools overlook changes occurring within printing and logging statements, disregarding them as behavior-preserving modifications. PurityChecker, on the other hand, considers these changes, recognizing that they do not impact the program's behavior.

A critical distinction lies in the handling of overlapping refactoring analysis. PurityChecker tolerates code modifications resulting from the application of overlapping refactorings, acknowledging that they do not alter the behavior of the refactored code—a feature absent in RefDistiller and GhostFactor, as detailed in Section 3.3.

Chapter 3

Approach

In this chapter, we outline the fundamental process for automatic detection of refactoring purity. Subsequently, we delve into an in-depth exploration of each section of the process, elaborating on every step in a comprehensive manner. As each method-related refactoring type possesses unique characteristics, we provide a detailed explanation specific to each refactoring type towards the end of the chapter. This procedural framework has been implemented within an open-source tool called PurityChecker, built on top of RefactoringMiner 2.0 (Tsantalis et al., 2020). The tool's code is publicly accessible on GitHub (Nouri, 2023).

3.1 Refactoring Purity Definition

As previously mentioned, a refactoring may involve several overlapping changes, such as bug fixes or feature implementations, which makes the refactoring non-behavior preserving. A refactoring is labeled as pure if it does not involve alterations that modify the behavior of the refactored code. This thesis concentrates on method-related refactoring operations, as we posit that purity can be clearly defined within this context.

Three primary categories of behavior-preserving code modifications are pertinent to the concept of refactoring purity: (1) Changes justified by the specific mechanics of a refactoring type, (2) Non-essential adjustments or changes within statement mappings, and (3) Alterations stemming from overlapping refactoring activities. By leveraging information about statement mappings and

the replacements within them, we assess whether code changes in a refactoring align with these categories, ultimately determining its purity.

Our definition of purity remains open-ended to underscore that the genuine "purity" of code changes is contingent on the specific contexts of their examination. As earlier indicated, our focus lies on method-related refactorings and the delineation of three key categories of justifiable changes. This approach allows us to contend that these changes are less likely to yield behavior-changing modifications into the developmental effort behind each alteration.

3.2 Automatic Refactoring Purity Detection

PurityChecker performs automatic assessment of refactoring purity for a set of 10 method-related refactorings. These refactorings include EXTRACT METHOD, EXTRACT AND MOVE METHOD, INLINE METHOD, MOVE METHOD, MOVE AND RENAME METHOD, MOVE AND INLINE METHOD, PULL UP METHOD, PUSH DOWN METHOD, SPLIT METHOD, and MERGE METHOD. We illustrate the core process for automated refactoring purity detection in Figure 3.1. The initial stage, marked as ⓪ in Figure 3.1, involves executing RefactoringMiner 2.0 on a specific commit. This step provides us with a list of all refactorings carried out in the commit, accompanied by statement mapping and replacement details. In the subsequent step, labeled as ① in the diagram, PurityChecker examines the replacements made during a given refactoring, determining whether these replacements adhere to purity criteria. Finally, the last stage, denoted as ②, involves PurityChecker's analysis of non-mapped statements within the refactoring to ascertain whether these statements have the potential to be neglected according to the refactoring's purity.

In the upcoming sections, we will delve into the mentioned stages and provide a more comprehensive explanation of the process for automatically assessing refactoring purity.

3.3 Step 1: Replacement Analysis

Replacements refer to the specific changes or alterations made to individual code statements during the process of refactoring. These replacements are crucial for tracking the modifications applied

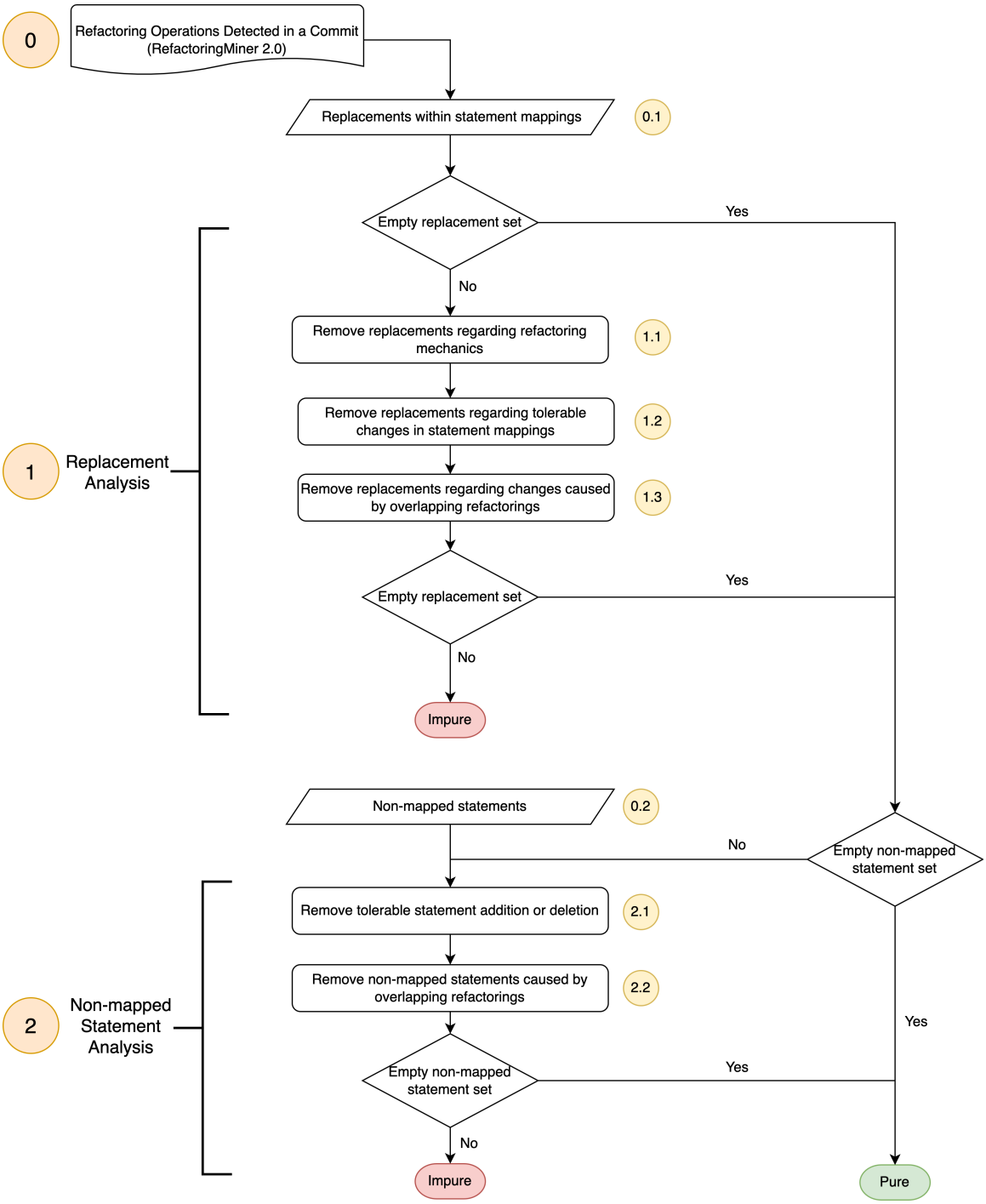

Figure 3.1: Refactoring Purity Detection Process

to the code as part of the refactoring operation. There are different replacement types further explaining the change that happened within a statement. Different types of replacements are required because refactoring operations can encompass a wide array of modifications, such as variable renaming, method extraction, parameter changes, and more. By categorizing these changes into different replacement types, the system can effectively represent and interpret the nuanced alterations introduced by the refactoring process. According to the RefactoringMiner 2.0, there are a total of about 80 different replacement types.

As previously mentioned, RefactoringMiner 2.0 supplies PurityChecker with replacement information (labeled as 0.1). By utilizing the replacement type in conjunction with the replacement details stored in each statement mapping instance, PurityChecker determines if a provided replacement is justifiable in terms of purity. This decision-making process occurs in three distinct stages, each addressing various scenarios in which a replacement can be deemed justified. Initially, PurityChecker employs a process where it attempts to decide whether replacement alterations are pure, considering the specific refactoring type and its corresponding mechanics (denoted as 1.1). Subsequently, PurityChecker evaluates if a given replacement change qualifies as a pure alteration, considering certain admissible code modifications that do not affect program's behavior (denoted as 1.2). Lastly, due to the potential for overlapping refactorings to yield pure code changes, PurityChecker analyzes the list of refactorings within a specific commit to identify replacement changes as instances of pure code modifications (denoted as 1.3).

To optimize computation time and resource usage, PurityChecker includes a check at the end of stages 1.1 and 1.2. If the set of replacements is empty, it bypasses the subsequent stages and proceeds directly to the analysis of non-mapped statements. This approach prevents unnecessary processing and resource consumption.

In the following sections, we will delve into a more detailed explanation of these stages.

Stage 1.1: Justifying Replacements Regarding Refactoring Mechanics

Refactoring mechanics refer to the specific techniques, steps, or practices used to carry out refactoring in a systematic and controlled way. These mechanics guide the developer through the process, ensuring that the restructuring of the code by the means of refactoring does not alter its external

behavior. Each refactoring type has its own set of mechanics that dictate how code is transformed to achieve a desired result while preserving or improving its quality and maintainability.

Many of the method-related refactorings involve specific mechanics that result in introducing code changes that preserve behavior, making them pure. We conducted a thorough analysis of these mechanics and identified the specific replacement changes that fall into this category. Consequently, we designed PurityChecker to recognize these mechanic-induced pure replacements and classify them as such in the code modifications.

Tsantalis et al. (2020) introduced two innovative techniques: Abstraction and Argumentization. These techniques enhance RefactoringMiner's capability to map statements effectively. They are closely linked to the concept of code changes induced by refactoring mechanics. **Abstraction** refers to deletion or addition of a `return` statement in case of a certain refactoring operations, such as EXTRACT METHOD and INLINE METHOD. For instance, when you extract an expression from a method, it becomes a return statement in the newly created method. **Argumentization** refers to replacement of expressions with parameters and vice versa, in case of some certain refactoring operations. For example, in case of an EXTRACT METHOD refactoring, when you extract duplicated code into a common method, the distinct expressions within the duplicated code become parameters in the new extracted method. The duplicated code is then replaced with calls to this new method, passing each distinct expression as an argument. Argumentization describes the pure replacement change between these parameters and arguments in such cases. Argumentization can be also applied in case of MOVE METHOD and INLINE METHOD refactorings. In case of a MOVE METHOD refactoring, when you relocate an instance method to a different class, there can be two types of changes: (1) removal of target type parameter; you might remove a parameter (or access to a field of the target class) from the original method. This parameter would be of the same type as the target class. (2) addition of source type parameter; you could add a parameter to the original method, and this parameter would be of the same type as the source class.

PurityChecker employs the mentioned techniques, in addition to further analyzing other pure replacement changes induced by refactoring mechanics, which help the tool to label these replacements as behavior-preserving changes in the context of refactoring purity. Table 3.1 provides a breakdown of which refactoring types are linked to specific mechanic-based replacement changes

Table 3.1: Correlation between Refactoring Categories and Their Unique Mechanic Induced Alterations

Refactoring	**Probable Mechanic-induced Pure Replacement**
Extract Method	-Abstraction of Expression(s) in a return Statement Change (Addition) -Parameter to Argument Change
Extract and Move Method	-Abstraction of Expression(s) in a return Statement Change (Addition) -Parameter to Argument Change -Adding Parameters (Induced by Move Method) -Removing Parameters (Induced by Move Method) -Modifications in how a method or field is accessed (Induced by Move Method)
Inline Method	-Abstraction of Expression(s) in a return Statement Change (Deletion) -Parameter to Argument Change
Move Method Move and Rename Method Move and Inline Method Pull Up Method Push Down Method	-Adding Parameters -Removing Parameters -Modifications in how a method or field is accessed

that render the replacement as pure.

To exemplify such instances of pure replacement changes and outline the approach PurityChecker employs to justify them, we have chosen a real-world case which is shown in Figure 3.2. This specific commit[1] is derived from the VoltDB project[2].

In this code alteration, the developer extracted `writeResponseToConnection` method from the source method in the parent version. As indicated in the figure, there is only one statement mapping that involves a change labelled as (6). When RefactoringMiner 2.0 is executed on this commit, it provides PurityChecker with information about the statement mapping and replacements occurring within the EXTRACT METHOD refactoring. As previously mentioned, there is just a single replacement here, characterized by the "VARIABLE_NAME" type, transitioning from

[1] Visit the GitHub commit link
[2] https://github.com/VoltDB/voltdb

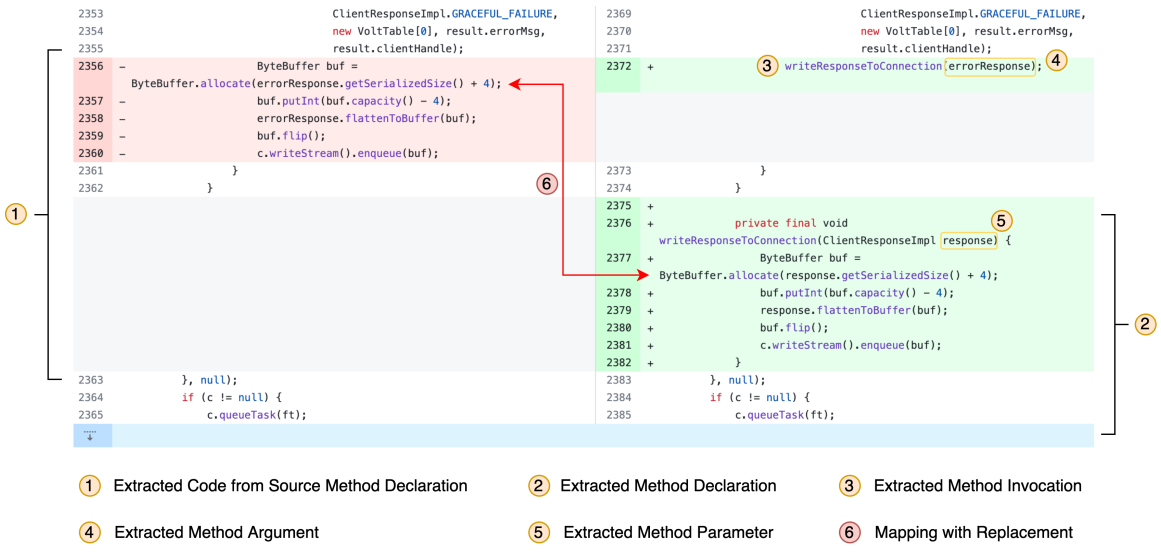

Figure 3.2: Extract Method with a Pure Replacement Caused by Refactoring Mechanics

`errorResponse` to `response`. Notably, these represent the extracted method's argument and parameter, respectively. According to the Table 3.1, a replacement involving a parameter and argument of an EXTRACT METHOD is a type of pure code modification. PurityChecker utilizes this information, considering the replacement values and their types, to conclude that this replacement is pure.

Stage 1.2: Justifying Replacements Regarding Tolerable, Behavior-preserving Changes in Statement Mappings

Numerous code modifications occurring at both the statement and expression levels are inherently behavior-preserving. These changes are thus classified as pure code modifications. Employing static source code analysis, we harnessed the data from statement mapping and the replacements themselves to formulate multiple rules for identifying such pure replacements. In our quest to develop and fine-tune these rules, we initially conducted an extensive review of existing literature to gain insights. For instance, a study by Kawrykow et al. (Kawrykow & Robillard, 2011) categorized several non-essential code modifications, suggesting that these alterations preserved the code's behavior. While this study was not comprehensive, it did provide valuable ideas for crafting our own purity rules.

Subsequently, through the meticulous process of manually validating our oracle, which is thoroughly explained in Chapter 4, we systematically devised various purity rules and continually refined them. PurityChecker is designed to recognize such replacements and label them as pure code modifications during its refactoring purity detection procedure. In the following, we will extensively clarify our purity rules and the thinking that guided their development along with one example for each .

Modification of Visibility, Modifiability, and `this` Keywords: During our analysis of Java code history, we came across situations where developers repetitively added or removed instances of the `this` keyword. In Java programming, adding the `this` keyword to a program element has an impact on the program's behavior in only a few specific scenarios. Modifications that include alterations related to the `this` keyword could enhance the code's readability to some extent. However, it is crucial to note that, in the majority of situations, these changes can be deemed pure and do not substantially affect the code's fundamental functionality.

In the context of PurityChecker's operation, which operates at the granularity of statements and expressions within methods, we do not take into consideration Java keywords that pertain to the visibility and modifiability of methods or classes. These keywords, like `public`, `private`, `protected`, and others, are related to the overall access and modifiability of methods or classes, which are not within the scope of PurityChecker's analysis.

However, when it comes to keywords that might be applied to individual statements within methods, such as `final`, `volatile`, and others, we do consider changes involving these keywords as pure code modifications. This decision is based on our analysis of Java code history and our definition of purity, which suggests that in most instances, changes related to these keywords within statements are non-essential and do not significantly impact the behavior of the program.

While there may be scenarios where altering a statement's keyword could affect program functionality, our analysis suggests that these cases are generally rare, and such changes are considered behavior-preserving according to our definition of purity.

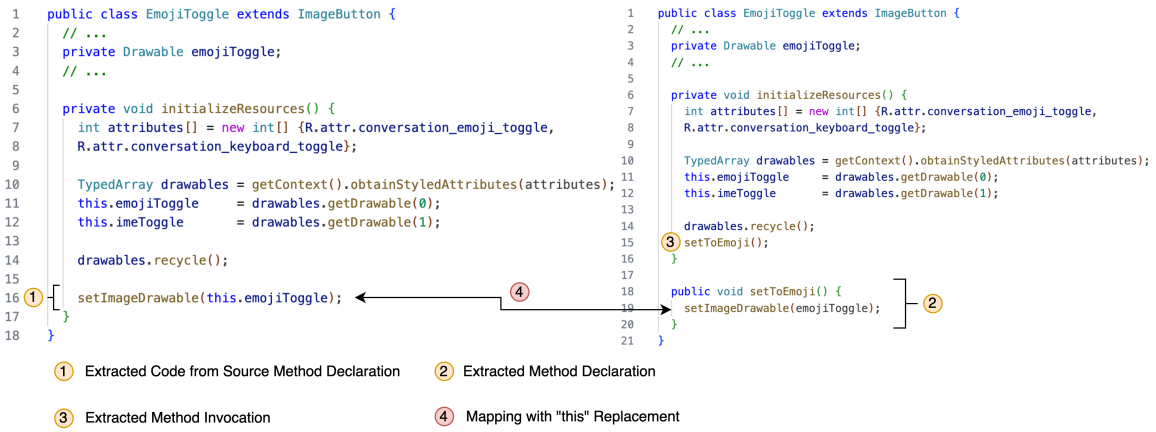

Figure 3.3: Extract Method with a Pure Replacement Caused by a Modification in `this` Keyword

To provide an example of such instances of pure replacement changes and outline the PurityChecker's approach to justify them, we have chosen a real-world commit[3] from Signal Android project[4], which is shown in Figure 3.3. Our focus is on the mapping labeled as ④. In this specific mapping, we observe a replacement involving two terms: `this.emojiToggle` and `emojiToggle`. It is important to note that this replacement is classified as a pure modification. The rationale behind this classification lies in the fact that both terms essentially refer to the same entity, specifically the `emojiToggle` attribute belonging to the `EmojiToggle` class.

As previously mentioned, the introduction or removal of the `this` keyword, as seen in this replacement, can be interpreted as an attempt to enhance the code's readability. In essence, this change does not fundamentally alter the behavior of the code; rather, it serves to make the code more comprehensible to developers.

Modification of Primitive Types: The replacement or modification of primitive types, such as changing an int to a long or vice versa, within the Java language can be classified as a behavior-preserving, pure change. This assertion can be substantiated by understanding how these data type conversions operate in the context of Java programming language.

In Java, primitive types are categorized based on their size and the values they can hold. For example, `int` is a 32-bit signed integer type, whereas `long` is a 64-bit signed integer type. When a developer decides to replace an `int` with a `long`, they are essentially widening the data type.

[3] Visit the GitHub commit link

[4] https://github.com/signalapp/Signal-Android

In this process, the new `long` data type can accommodate the values that the previous `int` could hold. This implies that any integer value that was held by the `int` variable can be easily stored within the `long` variable without any loss of data or precision. The code will continue to compile successfully, and the runtime behavior will remain consistent, as long as the widened range of the `long` type does not introduce unintended consequences due to numeric overflow.

Moreover, Java features automatic type conversion for widening primitive type conversions. This means that when an `int` value is assigned to a `long` variable, Java automatically handles the type conversion behind the scenes. The conversion process does not involve any additional logic or operations that could affect the program's behavior. Therefore, the code remains functionally equivalent before and after the replacement, as it continues to produce the same results for the same inputs. In essence, this replacement constitutes a pure change because it does not alter the code's functionality; it merely adjusts the data type to accommodate a broader range of values without introducing any behavioral changes.

This principle can be broadened to encompass additional legitimate substitutions of primitive types. For instance, it applies to situations where `byte` is substituted with `short` or `int`, `float` is swapped with `double`, or when smaller numeric types are replaced with `BigInteger`.

Modification of Printing and Logging Statements: Printing and logging statements in Java serve various purposes, primarily facilitating debugging, monitoring, or providing feedback to developers and end-users. These statements are designed to display information on the console or log files. When substitutions like switching from `System.out.println` to `System.err.println` or from `log.info` to `log.warn` are made, they essentially adjust the output level or destination while retaining the original intent of conveying information. Importantly, these changes only affect the manner in which messages are displayed or logged, not the program's underlying logic or functionality.

PurityChecker's assessment of these replacements as pure aligns with the principle of maintaining the essential purpose of the print or log statement. Whether transitioning between different output levels or adjusting the output destination, the core functionality of the program remains unaltered. The same reasoning extends to similar pairs of changes, such as transitioning between

different log levels, where the modifications pertain to the level of importance attributed to a message without affecting the program's fundamental functionality.

PurityChecker's classification of replacements within printing and logging statements in Java as pure is firmly grounded in their behavior-preserving nature. These modifications focus solely on adjusting the display, logging level, or output mechanism, leaving the program's logic entirely untouched. This alignment with the fundamental intent of the mentioned statements serves as a compelling rationale for their categorization as pure replacements within the context of this thesis.

Modification of Printing, Logging, and Exception Messages: PurityChecker's discerning evaluation of code replacements extends beyond statements to encompass replacements within the messages logged or printed by a program. We argue that alterations within these messages should be classified as pure. This perspective is underpinned by a set of compelling reasons that highlight that these changes do not affect the program's core behavior.

- **Semantic Consistency**: In the realm of logging and printing messages, the primary objective is to communicate information clearly and effectively. When we replace one message with another, it can be viewed as a pure modification. For example, changing a log message from *File not found* to *File does not exist* preserves the underlying issue being reported, which is the absence of a file. PurityChecker recognizes that these changes, while altering the wording, do not alter the fundamental behavior of the program.

- **Clarity and Readability**: Code maintainability and readability are paramount in software development. Occasionally, replacements within messages are driven by a desire for improved clarity or adherence to coding standards. For instance, substituting an abbreviation like "IO" with "Input/Output" enhances the comprehensibility of the message. These changes enhance the human understanding of the code without affecting the program's functionality. PurityChecker assesses these alterations as pure, as they contribute positively to code quality without introducing behavioral shifts.

- **Message Enrichment**: One of the common practices in software development is to enhance the context and detail of messages by incorporating explanatory variables. This involves replacing static or hard-coded message parts with placeholders that can be dynamically

filled with relevant information during runtime. For instance, substituting a generic message like `log.warn(``Error occurred in assignment")` with a message like `log.warn(` ``Error occurred in assignment of variable %d", x)` provides a more informative and context-rich log. PurityChecker recognizes that these changes improve the comprehensibility of messages without altering the program's fundamental behavior. They enhance the ability to diagnose issues and facilitate troubleshooting, making them valid candidates for pure replacements.

Supplier-get Pattern Extraction Modification: During the manual validation of our oracle, we encountered instances where developers replaced direct method invocations with a `Supplier` functional interface wrapping the invocation, followed by invoking `.get()` on the `Supplier` variable. This particular type of replacement caught our attention and prompted us to conduct a more thorough analysis. Such replacements can serve several purposes and often relates to control over when and how the method is executed. Here are some common reasons why developers might choose to make this kind of transformation:

- **Lazy Evaluation**: One of the primary motivations is to enable lazy evaluation of a method. When you directly call a method, it executes immediately. However, by using a `Supplier`, you can defer the execution of the method until it is actually needed. This can be beneficial for expensive or time-consuming operations that you want to delay until the result is genuinely required. It is a way to optimize performance by avoiding unnecessary computations.

- **Memoization**: Memoization is a technique where the results of a method call are cached for future use. By using a `Supplier`, you can implement memoization more easily. The method is only executed when necessary, and its result can be cached for subsequent calls. This is particularly helpful for functions that have expensive computations or frequently used results.

- **Conditional Execution**: Using a `Supplier` and `.get()` allows you to conditionally execute the method. You can wrap the method call in a condition, and the method will only

execute if the condition is met. This provides fine-grained control over when the operation takes place, which can be useful for scenarios where you want to execute the method under specific conditions or criteria. It is important to note that PurityChecker evaluates the rationale for adding conditions separately, which could potentially introduce impurity to the code transformation.

- **Concurrency**: In multi-threaded environments, using a `Supplier` can help manage thread safety. Since the `Supplier` is evaluated only when you call `.get()`, you can control access to potentially shared resources or mutable state more effectively, reducing the likelihood of race conditions and other concurrency issues.

It is important to note that, according to our analysis, when this replacement pattern is applied in isolation, without additional impure code changes, it qualifies as a behavior-preserving, pure replacement. In essence, it effectively maintains the program's original functionality. However, it is crucial to consider that if other impure code modifications are layered on top of this pattern, it can compromise the purity of the replacement. For instance, if the method being invoked within the `Supplier` is itself subject to impure changes, this specific replacement would be impure.

To illustrate this specific type of replacement and our rationale for categorizing it as pure, we have selected a real-world commit[5] from the infinispan[6] project, which is illustrated in Figure 3.4. In this commit, the `pOpReAwareOperation` method was extracted from the `remoteIterator` method. While most statement mappings remained unchanged, there was a seemingly substantial alteration in one particular mapping, labeled as (7) in our illustration. At first glance, the modification in variable assignment might raise concerns of impurity due to its apparent difference between the parent and child versions.

However, upon closer inspection and partial source code analysis using PurityChecker, after substituting the parameter-argument pairs, we found that the change in the statement mapping boiled down to a transformation from `ch.getPrimarySegmentsForOwner(localAddress)` to `() -> ch.getPrimarySegmentsForOwner(localAddress).get()`. PurityChecker

[5] Visit the GitHub commit link
[6] https://github.com/infinispan/infinispan

Figure 3.4: Extract Method with a Pure Replacement Caused by Supplier Pattern Extraction

identify this as a pure code modification through a meticulous string analysis process. This classification aligns with the Supplier pattern extraction, illustrating how even seemingly intricate changes can, in fact, preserve the program's behavior when scrutinized comprehensively.

Stage 1.3: Justifying Replacements Regarding the Changes Caused by Overlapping Refactoring Operations

During our manual validation of numerous commits within numerous projects, we learnt that overlapping refactorings can indeed lead to pure changes in code. This occurs when the combined effect of multiple refactorings maintains the original behavior of the code while improving its structure, readability, or maintainability. The key to achieving pure changes through overlapping refactorings lies in careful planning and understanding of the interactions between the refactorings involved, and the mechanics of each refactoring type.

Before we dive into the specifics of this stage, it is essential to provide further clarification regarding our rationale. We assert that if a code modification occurring within a method-related refactoring can be directly attributed to another refactoring within the same code commit, we categorize that change as a pure code modification. This classification remains consistent regardless of whether the overlapped refactoring is deemed pure or impure.

For instance, consider an EXTRACT METHOD refactoring. If a code change within the refactoring is directly triggered by a MOVE METHOD refactoring within the same commit, we categorize

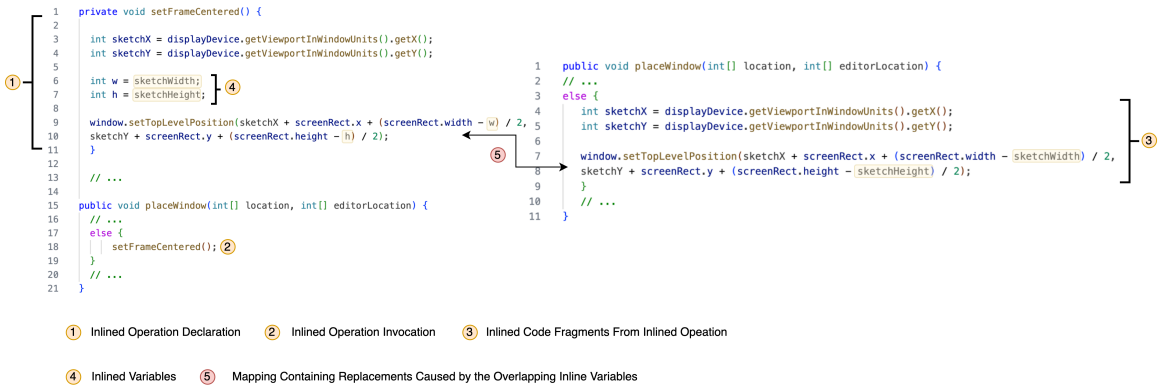

Figure 3.5: Inline Method with Pure Replacements Caused by Two Overlapping Inline Variable Refactorings

this code modification as pure, without regard to whether the MOVE METHOD refactoring itself is pure or impure.

This principle stems from our approach to evaluating the purity of method-related refactorings. As we discussed in Chapter 1, we focus exclusively on the portion of the code that is directly impacted by the particular refactoring in question. To put it simply, when we assess the purity of a refactoring, we consider only the elements of the code that are directly influenced by that specific refactoring operation. Therefore, in the scenario mentioned earlier, the purity classification of the MOVE METHOD refactoring does not affect how we evaluate the purity of the EXTRACT METHOD refactoring.

In the following, we will explore various situations where overlapping refactoring operations can lead to instances of pure replacements. Real-world project examples will be presented to illustrate these scenarios. It is crucial to emphasize that the rules and explanations presented here apply to all ten method related refactorings supported by PurityChecker. Towards the end of this section, we will also explore particular details that are unique to particular types of refactorings.

Inline Variable as an Overlapping Refactoring:

In the context of method-related refactoring operations, the INLINE VARIABLE refactoring has the potential to yield pure replacements, when it is applied as an overlapping refactoring on top of method-related refactorings. To shed light on this concept, this occurs when, within a statement mapping, the substitution of a variable's initializer with the variable itself takes place, which is the definition of INLINE VARIABLE refactoring. Despite the change being an replacement within the

statement mapping, it maintains the code's behavior, rendering it a pure and behavior-preserving replacement.

As an example, if a developer initially performed an INLINE METHOD refactoring to replace a method call with its body and then subsequently applied an INLINE VARIABLE to remove a variable that was introduced during the INLINE METHOD refactoring, this would be a pure replacement. It essentially reverses the effect of the earlier INLINE METHOD operation, restoring the code to its previous state without changing the program's behavior.

To offer a more concrete demonstration of the methodology employed by PurityChecker to justify the replacements arising from overlapping INLINE VARIABLE refactorings, we present a real-world case from the processing project[7]. The actual example is accessible via the provided GitHub commit link[8], visualized in Figure 3.5. Please note that we made minor adjustments to the code to facilitate its presentation as a demonstrative example.

As illustrated in the figure, the `setFrameCentered` method in the parent version has been inlined into the `placeWindow` method in the child version. On top of this code transformation, which is an instance of INLINE METHOD refactoring operation, the two variables `w` and `h` has been inlined as well. The impact of these two INLINE VARIABLE refactorings becomes evident through the replacements within the statement mapping marked as (5) in the figure. According to the output from RefactoringMiner 2.0, within this specific statement mapping, the `w` and `h` variables in the parent version has been replaced with `sketchWidth` and `sketchHeight`, respectively.

To classify such replacements as pure, PurityChecker leverages the RefactoringMiner 2.0 output, cross-referencing it with the initializers of the inlined variables, which is provided within the output of RefactoringMiner 2.0. This cross-reference ensures that the change indeed results from the INLINE VARIABLE refactoring and that no alterations in functionality are introduced. This methodology provides a robust means of confirming the purity of these replacements.

It is crucial to emphasize that categorizing an INLINE METHOD refactoring as a pure refactoring instance demands a more comprehensive analysis. This involves examining non-mapped statements, a topic we delve into in Section 3.4. In this section, our focus has been on elucidating scenarios

[7] https://github.com/processing/processing
[8] Visit the GitHub commit link

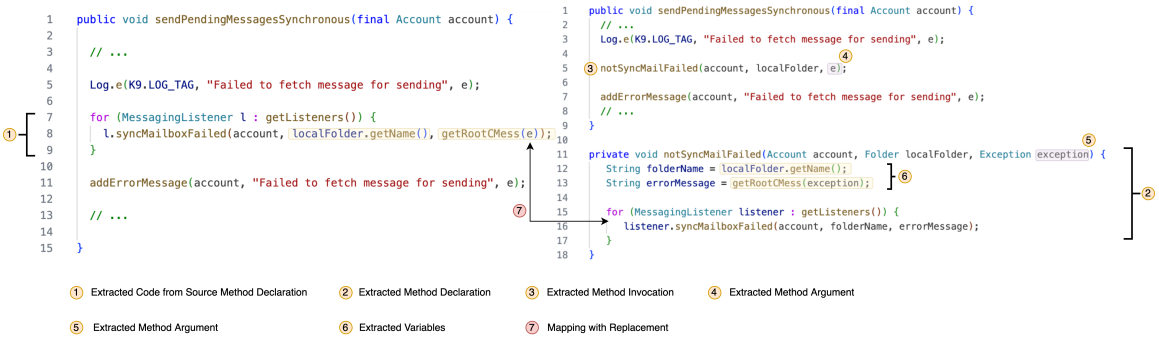

Figure 3.6: Extract Method with Pure Replacements Caused by Two Overlapping Extract Variable Refactorings

where we classify specific replacements as either pure or impure.

Extract Variable as an Overlapping Refactoring:

When the EXTRACT VARIABLE refactoring is employed in conjunction with method-level refactoring operations, it has the potential to yield pure, behavior-preserving code modifications. This process bears similarities to the one observed with overlapping INLINE VARIABLE refactoring, albeit in an inverse manner. In essence, if, within a statement mapping, an expression in the parent version is substituted with an extracted variable that retains the expression without further alterations, this results in a pure code modification.

To clarify, consider a scenario where a developer first performs a method-level refactoring, such as the EXTRACT METHOD operation, to encapsulate a block of code within a distinct method. Subsequently, if the EXTRACT VARIABLE refactoring is applied to generate a variable containing a segment of the previously extracted code, this typically constitutes a pure replacement, and the code's original behavior remains intact. However, it is crucial to note that if the expression being extracted to a variable undergoes impure modifications or if disparities emerge between the extracted variable's initializer and the expression itself, the replacement would then be deemed impure.

To illustrate the methodology employed by PurityChecker in identifying pure replacements resulting from overlapping EXTRACT VARIABLE refactorings, we provide a real-world example extracted from the K-9 Mail project[9]. The actual illustration is accessible through the provided GitHub commit link[10], which is depicted in Figure 3.6. It is worth noting that we made slight modifications

[9] https://github.com/thundernest/k-9

[10] Visit the GitHub commit link

to the code to enhance its suitability for demonstration purposes.

In the depicted example, the `notSyncMailFailed` method from the child version of the code extracted from the `sendPendingMessagesSynchronous` method in the parent version. Furthermore, two variables, `folderName` and `errorMessage`, are also extracted, as denoted by (6) in the figure. Notably, the initializers of these extracted variables are identical to the expressions from which they were extracted. Within the statement mapping designated as (7) in the figure, the developer opts to replace these expressions with the extracted variables. It is important to highlight that there is another pure replacement, where `e` is substituted with `exception`. This alteration stems from the mechanics of the EXTRACT METHOD refactoring. We have specifically discussed this process earlier in Section 3.3.

To assess these replacements stemming from overlapping EXTRACT VARIABLE refactorings, PurityChecker relies on the output from RefactoringMiner 2.0, analyzing statement mappings and replacements. The justification for these replacements is grounded in the fact that the initializer of the extracted variables in the child version is purely identical to the expressions in the parent version. It is worth noting that in the case of the `errorMessage` extracted variable, the initializer of the extracted variable is not exact similar to its corresponding expression on the parent version (`getRootCMess(e)` to `getRootCMess(exception)`). However, there is no functionality change involved due to the fact that the modification introduced by the EXTRACT METHOD refactoring mechanics. As a result, PurityChecker eliminates this refactoring mechanic-induced replacement as the replacements went thorough stage 3.3. Afterwards, the remaining replacements are justifiable according to our rule about identifying replacements caused by overlapping EXTRACT VARIABLE refactoring.

This specific example vividly demonstrates how various replacements can layer on top of each other while preserving the original behavior of the code.

Rename Variable as an Overlapping Refactoring:

RENAME VARIABLE and similar refactorings related to variable-like renaming are frequently employed in conjunction with method-level refactoring operations. These renaming refactorings, including RENAME PARAMETER, RENAME ATTRIBUTE, and their counterparts, often result in pure replacements when applied alongside or on top of method-level refactorings like EXTRACT METHOD

or INLINE METHOD refactoring.

Renaming a variable, parameter, or attribute entails a modification solely in the identifier used to reference a specific element within the code. Notably, this alteration does not introduce changes to the fundamental behavior or functionality of the code. Instead, it pertains to the syntactic aspects of code representation. Renaming a variable or parameter, often utilized to enhance code clarity and consistency. The renamed element retains its designated role and purpose within the method or class.

RENAME VARIABLE and similar refactorings primarily focus on improving code readability and maintainability by changing how elements are named, while method-level refactorings deal with the structure of code. As long as the behavior of the code remains unaltered after renaming, these changes are considered pure replacements within statement mappings, and this alignment is typically maintained when combined with method-level refactorings.

It is worth noting that several rename refactorings share common characteristics in terms of the replacements they introduce when applied as overlapping refactorings with method-level refactorings. For example, PARAMETERIZE ATTRIBUTE refactoring can result in similar replacements as RENAME VARIABLE when applied on top of a PULL UP METHOD refactoring. Therefore, PurityChecker treats RENAME VARIABLE, RENAME PARAMETER, PARAMETERIZE ATTRIBUTE, PARAMETERIZE VARIABLE, REPLACE VARIABLE WITH ATTRIBUTE, REPLACE ATTRIBUTE WITH VARIABLE, LOCALIZE PARAMETER, and RENAME ATTRIBUTE refactoring operations as largely equivalent when assessing the purity of introduced replacements when applied on top of method-level refactorings. This categorization is based on the logic used by RefactoringMiner 2.0, where these refactorings, except for RENAME ATTRIBUTE, are grouped under the same `RenameVariable-leRefactoring` object.

To illustrate how PurityChecker justifies replacements resulting from overlapping rename-related refactorings, we offer a practical example sourced from the IntelliJ IDEA Community Edition project[11]. The actual code can be accessed via the provided GitHub link[12], as visualized in Figure 3.7. It is worth mentioning that we have made minor modifications to the code to ensure its

[11] https://github.com/JetBrains/intellij-community
[12] Visit the GitHub commit link

Figure 3.7: Extract Method with Pure Replacements Caused by Overlapping Parameterize Attribute Refactoring

suitability for illustrative purposes.

As shown in the Figure, the `fillWithScopeExpansion` method in the child version, which is labeled as (2), has been extracted from the `computeInReadAction` method in the parent version. On top of this refactoring, a decision is made to pass one of the class's attributes, `myPattern`, as an argument to the newly created `fillWithScopeExpansion` method. This particular transformation aligns with the concept of PARAMETERIZE ATTRIBUTE refactoring, a technique that resembles RENAME VARIABLE or RENAME PARAMETER refactoring when it comes to evaluating the replacements generated by overlapping refactorings.

Within the statement mapping instance which is labeled as (6) in Figure 3.7, the reported replacement from RefactoringMiner 2.0 is the replacement of `myPattern` attribute with `pattern` variable. Considering the overlapped PARAMETERIZE ATTRIBUTE refactoring, PurityChecker categorizes this replacement as a pure code modification. It is essential to emphasize that the replacements resulting from all the aforementioned rename-related refactoring operations share a common nature and impact on the program. As an example, consider the scenario depicted in Figure 3.7, where, hypothetically, a developer chose to perform a RENAME ATTRIBUTE refactoring instead of the PARAMETERIZE ATTRIBUTE refactoring from `myPattern` to `pattern`. Interestingly, despite the difference in the refactoring operation applied, the resulting replacements would remain identical.

It is important to acknowledge that within the EXTRACT METHOD refactoring scenario presented here, there exist multiple non-mapped statements. The intricacies of analyzing non-mapped statements across different refactoring operations are discussed in a subsequent section.

Add and Remove Parameter as Overlapping Refactorings:

ADD PARAMETER and REMOVE PARAMETER refactoring operations have the potential to induce pure replacements within statement mappings when they are executed in conjunction with or on top of method-level refactorings. Specifically, these refactorings may result in a change in the number of arguments passed within a statement mapping that includes a method call. Such alterations can be attributed to the ADD PARAMETER or REMOVE PARAMETER refactoring operations occurring simultaneously with method-level refactorings.

As mentioned previously, when evaluating the purity of refactorings, our focus is primarily on the portion of code directly impacted by the specific refactoring operation, as well as the list of other refactorings performed within a commit. In this context, consider a scenario where an ADD PARAMETER refactoring is applied to method `A`, causing a modification in the number of arguments within one of the statement mappings of EXTRACT METHOD refactoring applied on method `B`. In such cases, our assessment of the purity of EXTRACT METHOD applied on method `B` is not influenced by the purity of ADD PARAMETER applied in method `A`. In essence, when analyzing the purity of the EXTRACT METHOD, while investigating statement mappings with replacements, PurityChecker exclusively references the list of other refactorings that occurred in the commit to justify replacements, irrespective of their individual purity states.

A pure replacement caused by an ADD PARAMETER or REMOVE PARAMETER refactoring has certain characteristics. First, the method name should remain the same within the code transformation, except for the cases of RENAME METHOD refactorings involved. Second, the sequence of the passed arguments should remain intact, with respect to the added or removed parameter.

ADD PARAMETER and REMOVE PARAMETER refactorings are not exclusive to regular methods; they can also be applied to constructors in Java. In an ADD PARAMETER refactoring within a constructor, a new parameter is introduced to the constructor's signature. This parameter serves as an additional piece of information required when creating an instance of the class.

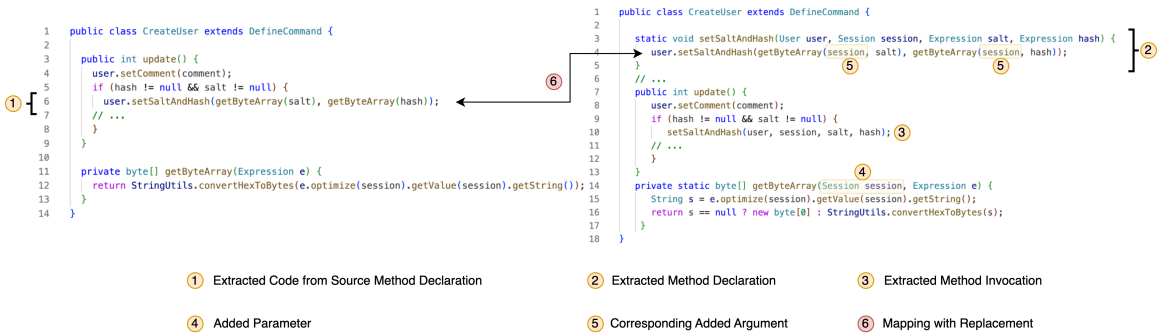

Figure 3.8: Extract Method with Pure Replacements Caused by Overlapping Add Parameter Refactoring

It is worth noting that we have designed PurityChecker to consider scenarios where ADD PARAMETER or REMOVE PARAMETER is applied to constructors as well. While the replacement type reported by RefactoringMiner 2.0 may differ in such cases, the fundamental mechanics underlying the application of Add Parameter or Remove Parameter as overlapping refactorings remain consistent.

Moreover, when PARAMETERIZE VARIABLE and PARAMETERIZE ATTRIBUTE refactorings are employed as overlapping refactorings, they can yield a similar impact in terms of replacements justification as that observed with ADD PARAMETER or REMOVE PARAMETER refactorings.

For an illustration of how PurityChecker justifies replacements arising from overlapping ADD PARAMETER or REMOVE PARAMETER refactoring operations, we present a real-world example[13] from the Lealone project[14].

In Figure 3.8, we observe a code transformation where the `setSaltAndHash` method is extracted from the `update` method. In the course of this transformation, the statement mapping identified as (6) within the figure experiences a replacement: an argument is added to the `getByteArray` method call. Meanwhile, RefactoringMiner 2.0 reports an ADD PARAMETER refactoring in the `getByteArray` method, accounting for the addition of the mentioned added argument in the statement mapping. Consequently, it is reasonable to justify this particular replacement as an outcome of the concurrent Add Parameter refactoring.

Encapsulate Attribute as an Overlapping Refactoring:

[13] Visit the GitHub commit link

[14] https://github.com/lealone/Lealone

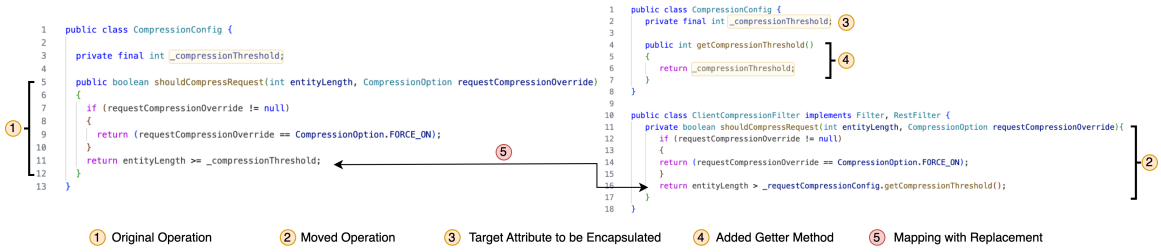

Figure 3.9: Extract Method with Pure Replacements Caused by Overlapping Encapsulate Attribute Refactoring

ENCAPSULATE ATTRIBUTE refactoring, when applied on top of or alongside method-level refactorings such as EXTRACT METHOD, can lead to pure replacements within the code. The ENCAPSULATE ATTRIBUTE refactoring involves wrapping an attribute within getter and setter methods. The attribute is typically made private, ensuring that it is not directly accessible from external code.

When the ENCAPSULATE ATTRIBUTE refactoring applied alongside method-level refactorings, such as Extract Method, it does not inherently change the behavior of the code. The pure replacements occur when the developer replaces direct access to the attribute with calls to the newly created getter and setter methods.

These replacements are typically pure because the encapsulation maintains the original behavior of the code. The encapsulation ensures that the attribute's value is accessed or modified through controlled methods, just as it was before the encapsulation.

To illustrate how PurityChecker justifies replacements caused by an overlapping ENCAPSULATE ATTRIBUTE refactoring, we present a real-world example from the rest.li project[15]. The actual code can be accessed via the provided GitHub link[16], as visualized in Figure 3.9. It is important to note that we made slight adjustments to the code to ensure its appropriateness for the purpose of demonstration.

In the depicted scenario, the `shouldCompressRequest` method has been relocated from the `CompressionConfig` class to the `ClientCompressionFilter` class. An additional change occurred within the `CompressionConfig` class, where the `_compressionThreshold` attribute was encapsulated into the `getCompressionThreshold` getter method, identified as

[15] https://github.com/linkedin/rest.li
[16] Visit the GitHub commit link

an ENCAPSULATE ATTRIBUTE refactoring by RefactoringMiner 2.0. Simultaneously, in the statement mapping marked as (5) in the figure, there is a replacement that transforms the direct access to `_compressionThreshold` into a method call to `getCompressionThreshold`. Essentially, this replacement swaps attribute access with its corresponding getter method. As previously mentioned, PurityChecker, armed with the knowledge of the associated Encapsulate Attribute refactoring reported by RefactoringMiner 2.0, labeled this replacement as a pure one resulting from the overlapped application of Encapsulate Attribute and Move Method refactorings.

Replace Accessor Call with Direct Field Access as an Overlapping Code Transformation (Refactoring):

When ENCAPSULATE ATTRIBUTE refactoring is applied on top of method-level refactorings, the standard outcome is that direct attribute access is replaced by calls to the getter method. This is done to enhance code maintainability and encapsulation.

On the flip side, in some cases, developers may decide to revert this change. They switch from using the getter method back to directly accessing the attribute. This reversal, too, can lead to pure replacements.

So, when ENCAPSULATE ATTRIBUTE is applied, it is common to see direct access replaced by the getter method. Conversely, when a developer reverts the change, pure replacements can also occur as direct access replaces calls to the getter method. These scenarios highlight the flexibility and context-specific nature of refactorings.

Since the aforementioned process is not classified as a refactoring operation by RefactoringMiner 2.0 yet, PurityChecker identifies these pure replacements through static string analysis, and accurately reports them as pure replacements. This analysis along with analysis of overlapping ENCAPSULATE ATTRIBUTE refactoring helps ensure that the tool recognizes and appropriately categorizes changes related to getter methods and attribute access in method-level refactorings.

Move Attribute as an Overlapping Refactoring:

MOVE ATTRIBUTE refactoring, when applied on top of or alongside method-level refactorings, can lead to pure replacements, primarily concerning various ways of accessing the moved attribute. Similar to the mentioned mechanic-induced code modification mentioned in Section 3.3, MOVE

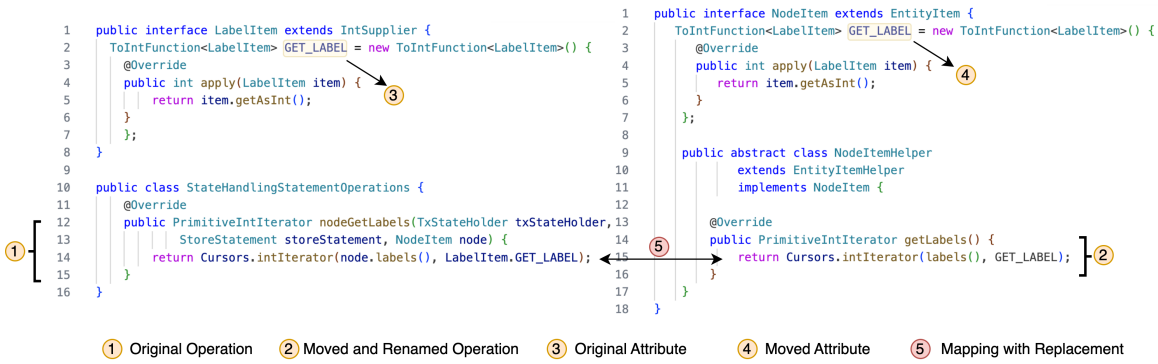

Figure 3.10: Move and Rename Method with Pure Replacements Caused by Overlapping Move Attribute Refactoring

ATTRIBUTE refactoring can cause pure code modifications regarding how an attribute can be accessed.

MOVE ATTRIBUTE refactoring entails transferring an attribute from one class to another. This action redefines the attribute's location within the codebase. When MOVE ATTRIBUTE is applied alongside or on top of method-level refactorings like EXTRACT METHOD or MOVE METHOD, it can alter how the attribute is accessed from different classes. For instance, if attribute `a` in class `B` is moved to class `C`, any attempt to access attribute `a` in class `D` will now involve calling `C.a` instead of `B.a`.

The replacements that arise due to this change in attribute access are typically pure. They do not modify the underlying behavior of the code; they merely establish new pathways for accessing the attribute. The primary purpose is to define how to access the attribute in its new location. This ensures that the code continues to function as intended, albeit with a change in access patterns.

To demonstrate the functionality of PurityChecker in the context of handling replacements resulting from the overlapping MOVE ATTRIBUTE refactoring, we have included a real-world example[17] extracted from the neo4j project[18], which is illustrated in Figure 3.10. It is important to note that we have made slight adjustments to the code to ensure it aligns with our specific objectives while preserving the essence of the original example.

As depicted in Figure 3.10, there has been a code modification involving the moving and renaming of method `nodeGetLabels`. This method was originally part of the `StateHandlingStat-`

[17] Visit the GitHub commit link
[18] https://github.com/neo4j/neo4j

-ementOperations class, but has been moved to the NodeItem.NodeItemHelper class, where it is now called getLabels. This specific change has been recognized as a Move and Rename Method refactoring by RefactoringMiner 2.0.

In addition to this alteration, there has been another adjustment in this commit. The GET_LABEL attribute, previously residing in the LabelItem class, has been moved to the NodeItem class, detected as a MOVE ATTRIBUTE refactoring by RefactoringMiner 2.0. One of the effects of this code change is evident in the statement mapping marked as (5) in the figure, where LabelItem.GET-_LABEL has been replaced with GET_LABEL.

Before this change, if a developer wanted to access the GET_LABEL attribute from any other class except for LabelItem, they had to specify the class of the attribute followed by the attribute name. However, post-transformation, if a developer needs to access the GET_LABEL attribute within the specific class to which it has been relocated, which is the NodeItem and NodeItemHelper classes, they can omit specifying the class name. This is consistent with Java's behavior, where attributes can be accessed from within their own class without mentioning the class name. It is worth noting that the GET_LABEL attribute can also be accessed from the NodeItemHelper class as an inner class without mentioning the class name.

Considering the details of the MOVE ATTRIBUTE refactoring and the adjustments within the statement mapping, PurityChecker utilizes a meticulous string analysis to deduce that the replacement in the mentioned statement mapping is purely a change in the way the attribute is accessed. As a result, PurityChecker allows for the elimination of the LabelItem in the statement mapping and designates the replacement as a pure code modification.

Merge Variable and Split Variable as Overlapping Refactoring:

MERGE VARIABLE and MERGE ATTRIBUTE refactorings are typically performed when developers realize that several variables or attributes serve similar purposes or can be logically grouped into one. They simplify the code by reducing redundancy. Both of these refactorings involve consolidating multiple variables (in the case of MERGE VARIABLE) or attributes (in the case of MERGE ATTRIBUTE) into a single variable or attribute.

In contrast to the MERGE VARIABLE and MERGE ATTRIBUTE refactorings, SPLIT VARIABLE and SPLIT ATTRIBUTE refactorings are performed when developers encounter a single variable or

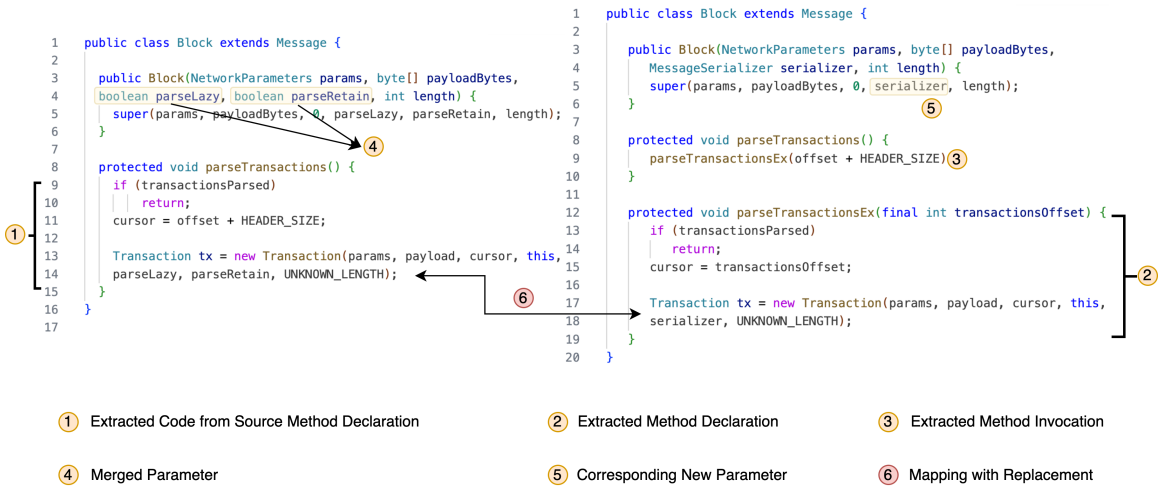

Figure 3.11: Extract Method with Pure Replacements Caused by Overlapping Merge Variable Refactoring

attribute that encompasses multiple, distinct purposes or characteristics. These refactorings aim to enhance code clarity and maintainability by segmenting a complex, multifaceted variable (in the case of SPLIT VARIABLE) or attribute (in the case of SPLIT ATTRIBUTE) into separate, well-defined components.

These refactorings can result in pure replacements when applied alongside or on top of method-level refactorings. The key is that the consolidation or separation of variables or attributes does not change the fundamental behavior of the code. It merely alters how the existing behavior is implemented. These refactorings often introduce cosmetic changes in the code. The behavior of the code, in terms of input-output relationships, remains unaltered. The changes are limited to how variables or attributes are accessed or structured.

We have provided an illustrative example to elucidate the scenarios in which a MERGE VARIABLE refactoring, when applied on top of method-level refactorings, results in pure replacements. Additionally, we have highlighted how PurityChecker responds when encountering these situations. This real-world example[19], as shown in Figure 3.11, is sourced from the bitcoinj project[20].

In the presented example, two significant code changes occurred. Firstly, the `parseTransac--tionsEx` method was extracted from the `parseTransactions` method. Secondly, the two

[19] Visit the GitHub commit link

[20] https://github.com/bitcoinj/bitcoinj

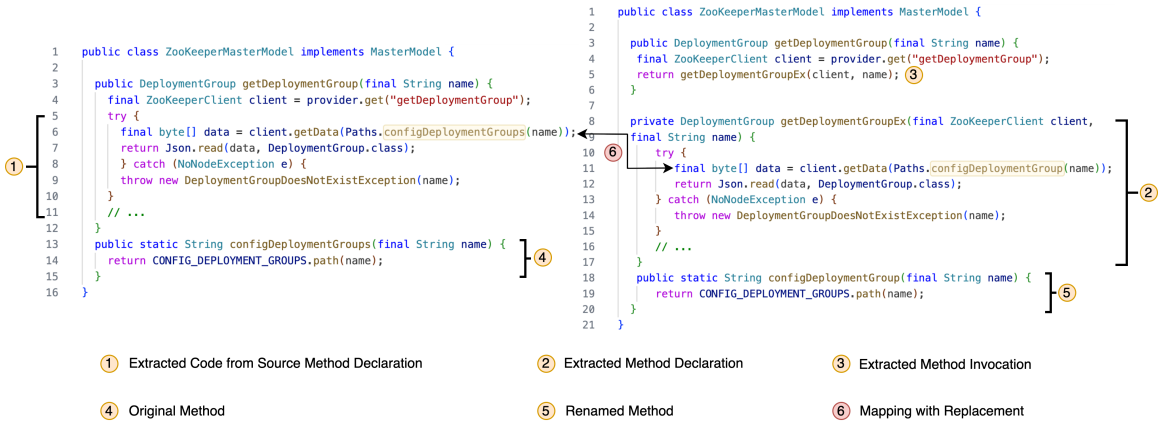

Figure 3.12: Extract Method with Pure Replacements Caused by Overlapping Rename Method Refactoring

parameters, `parseLazy` and `parseRetain`, were merged into a single parameter called `serializer`, a refactoring identified as MERGE PARAMETER by RefactoringMiner 2.0. It is important to note that additional contextual information in the commit reinforces the correctness of this MERGE PARAMETER refactoring report.

In the statement mapping denoted as (6) in Figure 3.11, the replacement `(parseLazy, parseRetain)` with `serializer` occurs. By considering the information about the MERGE PARAMETER refactoring and the replacement within the aforementioned statement mapping, PurityChecker classifies this replacement as a pure code modification.

Rename Method as an Overlapping Refactoring:

RENAME METHOD refactoring is a common practice in software development and aims to enhance code clarity and adherence to naming conventions. When a developer undertakes this refactoring, the core objective is to change a method's name, often to a more descriptive or convention-compliant identifier. This change focuses solely on the method's naming, leaving its underlying functionality and behavior intact. Therefore, this refactoring can result in pure replacements within the statement mapping when applied on top of alongside method-level refactoring.

The renaming process is relatively straightforward. Suppose we have a method, say `methodA`, and the developer decides to rename it to `renamedMethodA`. Consequently, every instance where `methodA` is called within the codebase gets replaced with `renamedMethodA`. These replacements occur within statement mappings that contain calls to `methodA`. The crucial aspect here is

that this replacement is purely a substitution of one method name for another. Consequently, such replacements will be considered as pure code modification by PurityChecker.

To exemplify situations where the application of RENAME METHOD refactoring on top of method-level refactorings yields pure replacements, we present a real-world code snippet[21] sourced from the Helios project[22], which is depicted in Figure 3.12. It is worth noting that we made slight adjustments to the original code sample to ensure its suitability for illustrative purposes.

As depicted in the figure, the extraction of the `getDeploymentGroupEx` method from the `getDeploymentGroup` method is evident. Additionally, a renaming of the `configDeployme-ntGroups` method to `configDeploymentGroup` has occurred. Notably, within the statement mapping denoted as (6) in the figure, a call to the originally named method has been replaced with the new name. PurityChecker combines data pertaining to the RENAME METHOD refactoring and the replacement within the specified statement mapping. It consequently determines that the replacement is entirely justifiable, being a direct outcome of the RENAME METHOD refactoring within the commit.

Move Method as an Overlapping Refactoring:

Similar to the scenario involving MOVE ATTRIBUTE refactoring, MOVE METHOD refactoring can also generate pure replacements within statement mappings when applied atop method-level refactorings. This phenomenon occurs when a method is relocated from one class to another, causing changes in how the method is accessed across different classes. As a result, calls to the moved method from other classes need to be adjusted to reflect the new location.

For instance, in case of a static method `m`, if `m` originally resided in class `A` but is moved to class `B`, references to `m` within class `C` must change from `A.m` to `B.m`. These changes primarily concern the mechanics of method access, rather than modifying the core behavior of the code.

It is important to note that these replacements within statement mappings are typically regarded as pure, given that they pertain to how the method is invoked in various parts of the code, and the fundamental behavior of the code remains unaltered. PurityChecker's analysis considers these changes in access patterns and categorizes them as pure replacements.

[21] Visit the GitHub commit link
[22] https://github.com/spotify/helios

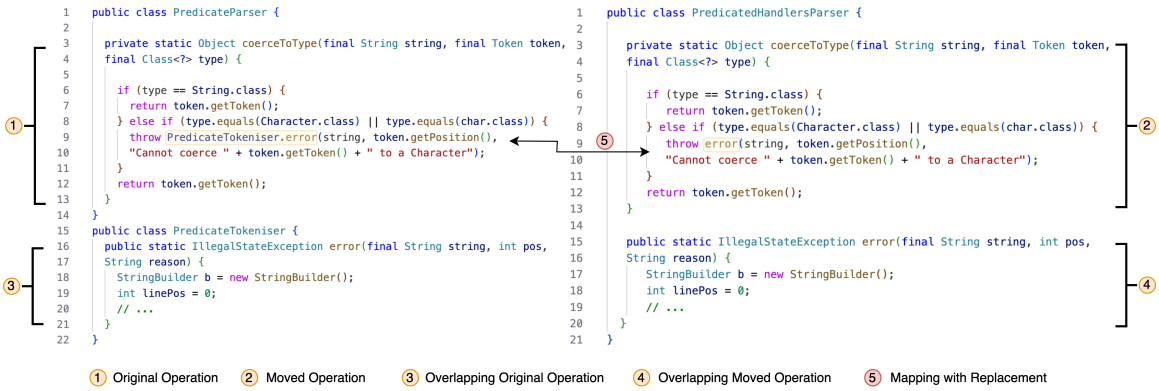

Figure 3.13: Move Method with Pure Replacements Caused by Overlapping Move Method Refactoring

To elucidate the concept of MOVE METHOD refactoring giving rise to pure replacements when performed overlapping with method-level refactorings, we have included a real-world example[23] from the undertow project[24], depicted in Figure 3.13. It is important to mention that minor adjustments were made to the code to optimize it for our illustrative purposes.

In the provided illustration, the `coerceToType` method has been relocated from the `PredicateParser` class to the `PredicatedHandlersParser` class. Simultaneously, the `error` static method was moved from the `PredicateTokeniser` class to the `PredicatedHandlersParser` class. When scrutinizing the statement mapping denoted as (5) in the figure, we can observe a replacement from `PredicateTokeniser.error(...)` to `error(...)`. PurityChecker justifies this replacement as pure. It argues that this modification results directly from the transfer of the static method `error` from the `PredicateTokeniser` class to its new home, `PredicatedHandlersParser` class. Hence, the removal of the class prefix in the replacement is considered pure, attributing this code transformation to the overlapping MOVE METHOD refactoring.

Pull Up and Push Down Method as Overlapping Refactorings:

PULL UP METHOD and PUSH DOWN METHOD refactorings can lead to pure replacements when applied on top of method-level refactorings, similar to MOVE METHOD refactoring, as their underlying mechanics are similar. These refactorings often change how methods are accessed in different

[23] Visit the GitHub commit link

[24] https://github.com/undertow-io/undertow

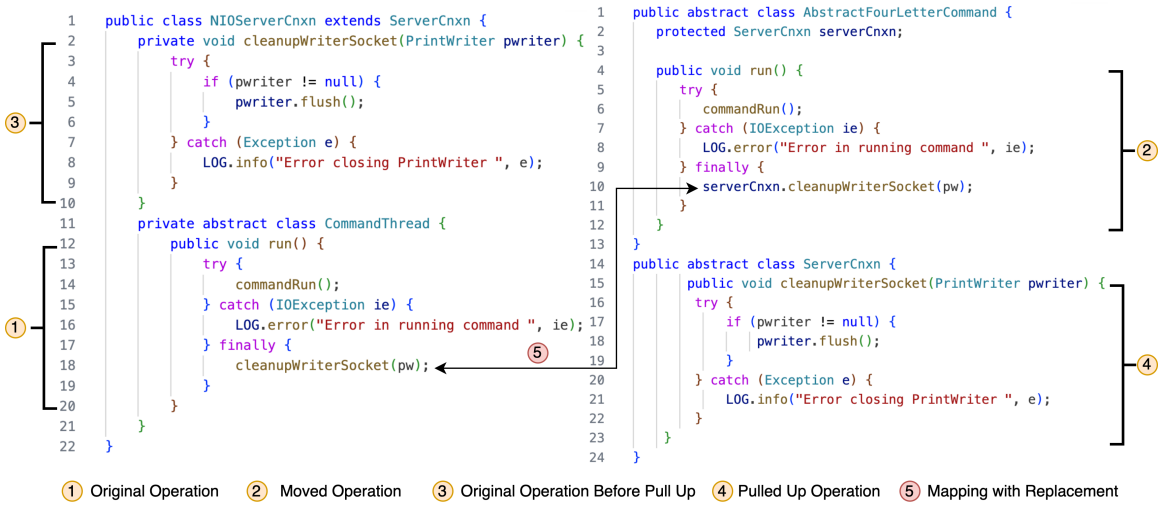

Figure 3.14: Move Method with Pure Replacements Caused by Overlapping Pull Up Method Refactoring

classes, thus impacting statement mappings which involve calls to the pulled up and pushed down methods.

When a developer performs a PULL UP METHOD refactoring, the method is moved from a subclass to a superclass. In the superclass, this method becomes accessible to all the subclasses as well. Conversely, with PUSH DOWN METHOD, a developer is moving a method from a superclass to one or more subclasses. This allows the subclasses to access the method directly. In the subclass that gains the method, there may be replacements needed to accommodate the new method call.

These refactorings are similar to MOVE METHOD refactoring in terms of introducing probable pure replacements, because they mainly affect how methods are accessed. PurityChecker identifies such changes as behavior-preserving, resulting in pure replacements.

To better illustrate how pure replacements can arise from applying PULL UP METHOD and PUSH DOWN METHOD on top of method-level refactoring, we have provided a real-world example[25] sourcing from the zookeeper project[26], which is illustrated in Figure 3.14. Please note that we made slight code adjustments to enhance clarity and suitability for illustration.

In the provided illustration, the `run` method was moved from the `CommandThread` class to the `AbstractFourLetterCommand` class. Additionally, the `cleanupWriterSocket`

[25] Visit the GitHub commit link

[26] https://github.com/apache/zookeeper

method was moved from the `NIOServerCnxn` class to its superclass, `ServerCnxn`, and this action was detected as a PULL UP METHOD refactoring by RefactoringMiner 2.0. Within the statement mapping denoted as (5), we observe a replacement from `cleanupWriterSocket(pw)` to `serverCnxn.cleanupWriterSocket(pw)`. This change signifies the shift from directly accessing the `cleanupWriterSocket` method to using the `serverCnxn` field to access it after the PULL UP METHOD refactoring.

PurityChecker analyzes this replacement by considering the type of the class field `serverCnxn` with the type `ServerCnxn`, the replacement itself, and the PULL UP METHOD refactoring. Based on these factors, it concludes that this replacement is indeed pure. It is crucial to note that the mechanics behind introducing pure replacements when applying MOVE METHOD, PULL UP METHOD, and PUSH DOWN METHOD refactorings as overlapping operations are essentially the same.

Rename Class and Move Class as Overlapping Refactorings:

Renaming or moving a class can result in pure replacements when applied on top of method-level refactorings. By applying a RENAME CLASS refactoring, all class instantiations of the original class name within statement mappings are substituted with the new class name, essentially constituting pure replacements. If the renaming also involves moving the class to a different package, any references to the previous package name are likewise updated within the statement mappings to align with the new package structure.

Furthermore, when the class name is referenced using its fully qualified name, encompassing both the class name and the package name, these references undergo modifications to mirror the new class name and its location within the package structure.

In this context, PurityChecker leverages information about RENAME CLASS and MOVE CLASS refactorings to assess the replacements occurring within statement mappings. These replacements often manifest in changes to parameter and variable types. It is important to emphasize that these alterations essentially constitute syntactic adjustments aimed at harmonizing the code with the new class name or location. Consequently, these changes within statement mappings can typically be classified as pure replacements, as they maintain the code's fundamental logic intact.

3.4 Step 2: Non-mapped Statement Analysis

Non-mapped statements within the body of method-level refactorings are portions of code that do not have a direct, one-to-one correspondence between the original version and the refactored version when a method-level refactoring is applied. In other words, these are statements that are either added, removed, or significantly altered during the refactoring process, so they can not be precisely mapped from the original code to the refactored code.

For instance, when you perform a method-level refactoring like EXTRACT METHOD, some statements from the original method might be moved to the new extracted method, while others remain in the original method. The ones that are moved to the new method are considered mapped because they have a clear correspondence in the new code. However, statements left behind in the original method that are no longer present in the new extracted method are considered non-mapped. The context can be inverted as well. In certain situations, non-mapped statements may also be observed in the refactored method instead of the original method.

Analyzing the non-mapped statements is essential in assessing the purity of method-level refactorings because these statements can impact the code's behavior or introduce impurities, and they are in the focus of PurityChecker for ensuring that refactorings maintain code's behavior and assessing the refactorings' purity.

Typically, when non-mapped statements are present within method-level refactorings, the purity of the refactoring is compromised, leading to its classification as impure. Nonetheless, there are specific scenarios in which non-mapped statements can co-exist with method-level refactorings without violating their purity criteria. We will now explore these scenarios to provide a more comprehensive understanding of this concept.

Stage 2.1: Tolerable Statement Addition or Deletion

We can identify a few scenarios in which statement additions or deletions, categorized as non-mapped statements, do not introduce any impact on the program's behavior or the purity of the refactoring operation. Additionally, certain types of refactorings have inherent mechanics that make them tolerant to non-mapped statements while preserving the functionality of the refactored code.

These scenarios can be broadly classified into two main categories, which we will explore in detail.

Non-Mapped Statements with No Impact on Behavior:

PurityChecker employs a rule that disregards certain statement types when assessing the purity of a refactoring. This rule was established through extensive analysis, wherein numerous refactorings were rigorously evaluated for their purity.

The primary categories of statements exempt from consideration encompass those that do not carry any behavioral information. These determinations are primarily derived from our comprehensive examination of refactorings' purity within the process of our oracle validation.

Statements related to printing or logging functions fall into this category. Typically, these statements serve debugging purposes. Therefore, when assessing method-level refactorings, any additions or removals of such statements are overlooked. For example, consider a MOVE METHOD refactoring, where a method relocates from class `A` to class `B`. In this context, the introduction of new logging statements within the moved method is inconsequential to the refactoring's purity, as long as the other purity criteria remain satisfied. Additionally, debugging assertions, like those involving the `assert` statement, are omitted from purity assessments, as they are primarily meant for debugging.

Furthermore, in situations where variable declarations have no substantial impact on code behavior, they will be exempt from scrutiny, particularly if the variables have no relevance elsewhere in the refactoring-affected code.

It is crucial to note that the specific list of statement types to disregard can be refined based on the particular requirements of a code analysis tool or the context of the refactoring under evaluation. The categories provided above represent common categories happened within our dataset, but the exact criteria for determining purity can be tailored to the needs of a given project or analysis tool.

Non-mapped Statements Resulting from Certain Refactoring Mechanics:

In the context of the EXTRACT METHOD refactoring, PurityChecker allows for the presence of non-mapped leaves within the source operation. This means that any non-mapped statements or code that appear within the source operation before the extraction of the method do not impact the refactoring's purity assessment. In essence, PurityChecker focuses exclusively on the portion of the code affected by the actual extraction process.

This approach aligns with the mechanics of the EXTRACT METHOD refactoring itself. The goal of an EXTRACT METHOD operation is to isolate a specific block of code into a separate method. Therefore, any code that exists outside this selected block is irrelevant to the extraction process. As a result, non-mapped leaves within the source operation are disregarded in the assessment of the EXTRACT METHOD refactoring's purity, as they do not affect the refactoring's intended behavior.

Additionally, it is important to note that EXTRACT METHOD refactorings can introduce return statements within the newly created method. These additions are also considered pure changes, and they are permitted as non-mapped leaves in the refactored code.

On the other hand, in the context of the INLINE METHOD refactoring, PurityChecker extends a degree of tolerance for non-mapped leaves within the target operation (child version). The criteria for assessing purity of INLINE METHOD differ because this refactoring is essentially the opposite of the EXTRACT METHOD.

During an INLINE METHOD operation, the goal is to replace a method call with the actual method body. In this case, non-mapped leaves or statements found within the target operation (the child version) before the inlining are disregarded in the purity assessment of the INLINE METHOD refactoring. These statements do not affect the intended behavior of the refactoring and are permitted as they are part of the refactoring's mechanics.

This approach aligns with the mechanics of the INLINE METHOD refactoring, where the target method (the one being inlined) is absorbed into the calling method. The context of the target method beyond the inlined portion is not considered when evaluating the INLINE METHOD refactoring's purity.

Regarding return statements, the allowances for non-mapped leaves in the context of the INLINE METHOD refactoring are switched from the allowance of addition of return statements to the allowance of deletion of return statements in the inlined operation (child version). These adjustments align with the specific mechanics of the INLINE METHOD refactoring and its purity criteria.

To illustrate this phase more clearly, we present a real-world example[27] sourced from the facebook-android-sdk project[28]. The example is visualized in Figure 3.15. Please note that we

[27] Visit the GitHub commit link

[28] https://github.com/facebook/facebook-android-sdk

Figure 3.15: Pure Extract Method Refactoring In Presence of Non-mapped Statements

have made slight adjustments to the code to ensure its suitability for demonstration purposes.

In the depicted scenario, we have an EXTRACT METHOD refactoring where the `getErrorMes--sage` method is being extracted from the `handleResultOk` method. The source operation before the extraction, labeled as (1) in the figure, contains non-mapped statements labeled as (3). These non-mapped statements, although present in the parent version, do not affect the purity of the extracted method, labeled as (4) in the figure. Additionally, a `return` statement, marked as (5) in the figure, has been added to the body of the extracted code portion, which aligns with the Extract Method refactoring mechanics.

Given these considerations, PurityChecker classifies this EXTRACT METHOD refactoring as pure, since it satisfies all the purity criteria related to replacements and non-mapped statements.

Stage 2.2: Justifying Non-mapped Statements Caused by Overlapping Refactoring Operations

Several refactoring operations have the potential to introduce non-mapped statements when applied alongside method-level refactorings. Since the inclusion or removal of these non-mapped statements directly results from the application of overlapping refactorings, PurityChecker tolerates these changes. These non-mapped statements are accepted because we argue that they do not alter the program's behavior.

Extract Variable and Inline Variable as Overlapping Refactorings:

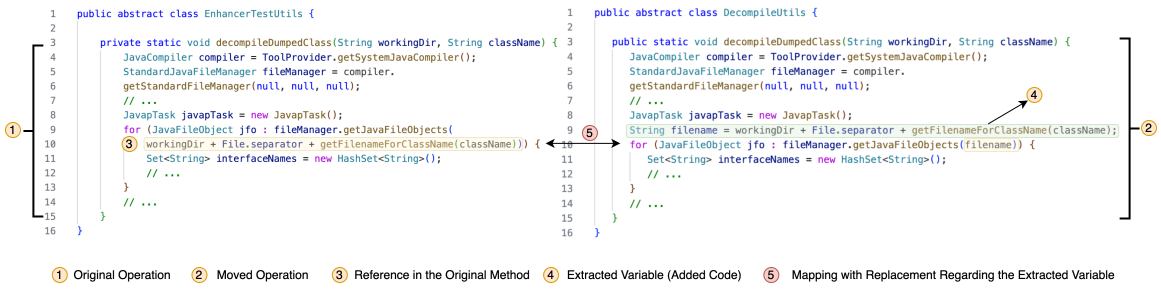

Figure 3.16: Pure Move Method Refactoring In Presence of Non-mapped Statements Caused by Overlapping Extract Variable

The Extract Variable refactoring can lead to the pure addition of non-mapped statements when applied alongside or on top of a method-level refactoring, such as MOVE METHOD. When EXTRACT VARIABLE is applied on top of method-level refactorings, it often involves the creation of new variable declarations within the refactored code. These new variables are essentially non-mapped statements because they do not exist in the original version of the method. The addition of these non-mapped statements, which mainly consists of the extracted variable declarations, is a direct result of the EXTRACT VARIABLE refactoring's mechanic.

In terms of purity, these non-mapped statement additions can be deemed pure. They serve to enhance code understandability without altering the fundamental behavior of the method. Therefore, PurityChecker tolerates these additions, considering them compatible with the code's core functionality.

In summary, when EXTRACT VARIABLE is applied on top of method-level refactorings, it may result in the pure addition of non-mapped statements, primarily in the form of newly declared variables within the refactored method, contributing to code clarity while preserving its behavior.

Similarly, when the INLINE VARIABLE refactoring is employed in conjunction with method-level refactorings, it can result in the reporting of non-mapped statements in the original method, particularly the deleted variable declarations. This occurrence primarily transpires within the original method, contrasting with the non-mapped statement additions associated with the EXTRACT VARIABLE refactoring.

To illustrate how EXTRACT VARIABLE and INLINE VARIABLE refactorings can lead to the pure

addition or removal of non-mapped statements, we have presented a real-world example[29] derived from the hibernate-orm project[30], which is depicted in Figure 3.16. Please note that we have made minor modifications to the code to ensure its appropriateness for this illustrative purpose.

In the presented scenario, the `decomplileDumpedClass` method was moved from the `EnhancerTestUtils` class to the `DecompileUtils` class. Simultaneously, the developer carried out an EXTRACT VARIABLE refactoring, as indicated by labels ③ and ④ in the figure, resulting in the extraction of the `fileName` variable.

The figure highlights a particular statement labeled as ④, distinguished in green, which does not have a direct counterpart in the original method. RefactoringMiner 2.0 identifies this as a non-mapped statement. However, PurityChecker takes a more nuanced approach by comparing the initializer of the extracted variable with the 'before' version of the replacement within the statement mapping marked as ⑤ in the figure. As the replacement aligns perfectly with the initializer of the extracted variable, PurityChecker deems the addition of this statement, corresponding to the declaration of the extracted variable, as acceptable and pure. In the previously presented example, we encounter a combination of pure replacement and pure statement addition (non-mapped statement) due to the overlapped use of the EXTRACT VARIABLE refactoring.

Localize Parameter and Parameterize Variable as Overlapping Refactorings:

LOCALIZE PARAMETER refactoring is a process in which a parameter, which was originally accessible as a variable in the method or operation, is transformed into a local variable within the method. In essence, it shifts a parameter's scope from being a method's input to being a locally scoped variable within the method.

When the LOCALIZE PARAMETER refactoring is applied on top of method-level refactorings, it might introduce non-mapped statements within the refactored code. This happens because when a parameter is localized, a new local variable declaration is introduced to capture the parameter's value at the beginning of the method. This newly added local variable declaration qualifies as a non-mapped statement, as it is introduced solely due to the LOCALIZE PARAMETER refactoring and does not have a direct counterpart in the original code.

[29] Visit the GitHub commit link

[30] https://github.com/hibernate/hibernate-orm

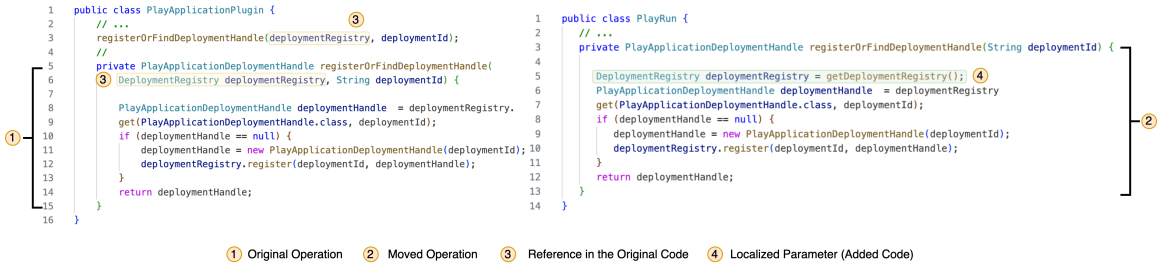

Figure 3.17: Pure Move Method Refactoring In Presence of Non-mapped Statements Caused by Overlapping Localize Parameter Refactoring

PARAMETERIZE VARIABLE is somewhat related to the concept of LOCALIZE PARAMETER but involves a different process. In PARAMETERIZE VARIABLE, a local variable within a method is elevated to become a parameter of that method. When PARAMETERIZE VARIABLE is applied on top of method-level refactorings, it can result in non-mapped statements in the original code. This occurs because, in PARAMETERIZE VARIABLE, the original local variable is transformed into a parameter, which result in the deletion of the variable declaration in the original method.

The mechanisms behind how LOCALIZE PARAMETER and PARAMETERIZE VARIABLE refactorings introduce non-mapped statements are quite akin to the process seen in EXTRACT VARIABLE and INLINE VARIABLE refactorings. In all these cases, we witness the addition or deletion of non-mapped statements as a direct consequence of the respective refactoring operations.

To better elaborate on this concept, we have presented a real-world example of applying LOCALIZE PARAMETER refactoring on top of a method-level refactoring. This real-world examples[31] sourced from the gradle project[32], which is depicted in Figure 3.17. It is worth mentioning that we have made minor code adjustments to facilitate illustration.

In this scenario, the `registerOrFindDeploymentHandle` method has been moved from the `PlayApplicationPlugin` class to the `PlayRun` class. Additionally, a parameter called `deploymentRegistry`, as indicated by (3) in the figure, has undergone the LOCALIZE PARAMETER refactoring. This refactoring introduced the `deploymentRegistry` local variable and added it to the moved method.

[31] Visit the GitHub commit link
[32] https://github.com/gradle/gradle

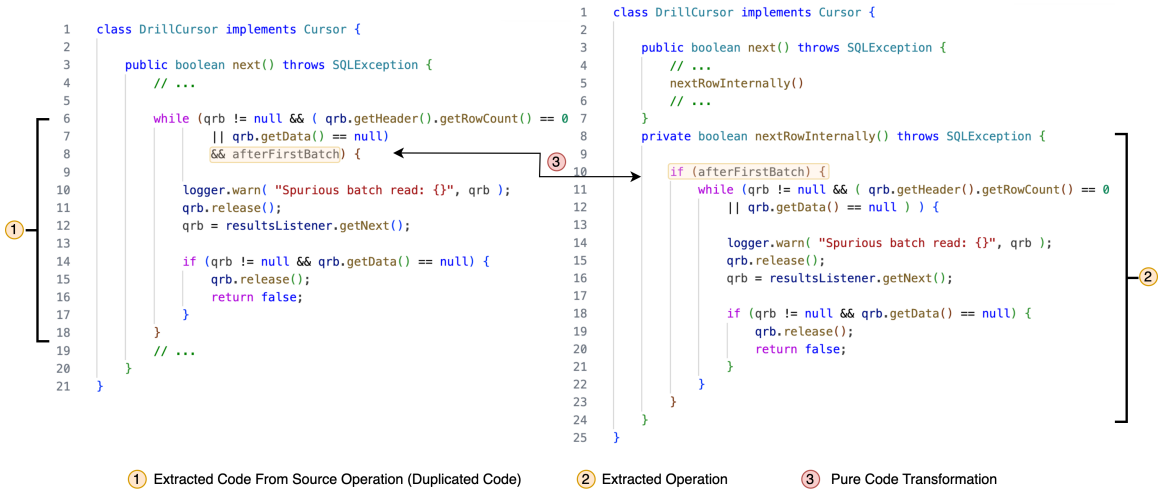

Figure 3.18: Pure EXTRACT METHOD Refactoring In Presence of Non-mapped Statement Caused by Overlapping SPLIT CONDITIONAL Refactoring

PurityChecker permits this addition of a statement for two primary reasons. Firstly, the statement is directly related to the concurrent application of the LOCALIZE PARAMETER refactoring. Secondly, as the new variable's initialization is inherently linked to the argument passed as the corresponding parameter, PurityChecker deems this statement addition a direct outcome of the overlapped LOCALIZE PARAMETER refactoring.

As previously mentioned, PurityChecker also tolerates replacements involving the direct access of fields with their associated getter methods, as observed in overlapping ENCAPSULATE ATTRIBUTE refactoring, which is mentioned in section 3.3.

Merge and Split Conditional as Overlapping Refactorings:

MERGE CONDITIONAL is a refactoring technique often employed when developers identify multiple conditional statements within a code block that can be simplified and grouped into a single, more concise conditional expression. By doing so, they reduce code redundancy and enhance code clarity. This refactoring can yield pure code modifications, particularly when merging conditionals that perform distinct and non-overlapping checks. When these merged conditionals are behaviorally equivalent to the individual conditions, the refactoring results in a pure change, preserving the functionality of the code while making it more streamlined.

SPLIT CONDITIONAL, on the other hand, is used when developers encounter a complex, compound conditional statement that can be logically separated into multiple smaller, more manageable

conditions. This refactoring simplifies code readability and helps prevent nested or deeply nested conditionals. Split Conditional can also lead to pure code modifications when a complex condition is divided into separate, distinct conditions that individually serve a specific purpose. If these newly created conditions, which are based on the original compound condition, maintain the same behavior, the Split Conditional refactoring produces pure changes.

The changes causing from the overlapping application of these two refactorings appear as statement deletion (in case of MERGE CONDITIONAL) or addition (in case of SPLIT CONDITIONAL) within the body of method-level refactorings. To provide an example showing the effect of applying these refactorings on top of method-level refactorings, we have included a real-world case[33] from the drill project[34], which is depicted in Figure 3.18.

Within this example, the `nextRowInternally` method has been extracted from the `next` method. Within this EXTRACT METHOD refactoring, the developer decided to split the while conditions into an if statement containing one of the expressions within the while condition, and another while statement containing the rest conditions, which is reported as a SPLIT CONDITIONAL refactroing by RefactoringMiner 2.0. Moreover, the newly added if statement is reported as a non-mapped statement by RefactoringMiner 2.0. This specific code transformation, marked as (3) in the figure, qualifies as a pure code alteration, as it is justifiable by the overlapped SPLIT CONDITIONAL refactoring.

Rename Method as an Overlapping Refactoring:

Interestingly, when RENAME METHOD refactoring is applied on top of method-level refactorings, it can lead to the introduction of non-mapped statements in both the original and refactored methods. This peculiarity arises from the way PurityChecker analyzes data provided by RefactoringMiner 2.0, including replacements, statement mappings, and refactoring operations.

In specific situations, when a RENAME METHOD refactoring significantly changes a method name, particularly if argument names are also altered, RefactoringMiner 2.0 may not recognize the statements influenced by overlapping RENAME METHOD, as mapped statements, even though they inherently should be.

[33] Visit the GitHub commit link
[34] https://github.com/gradle/gradle

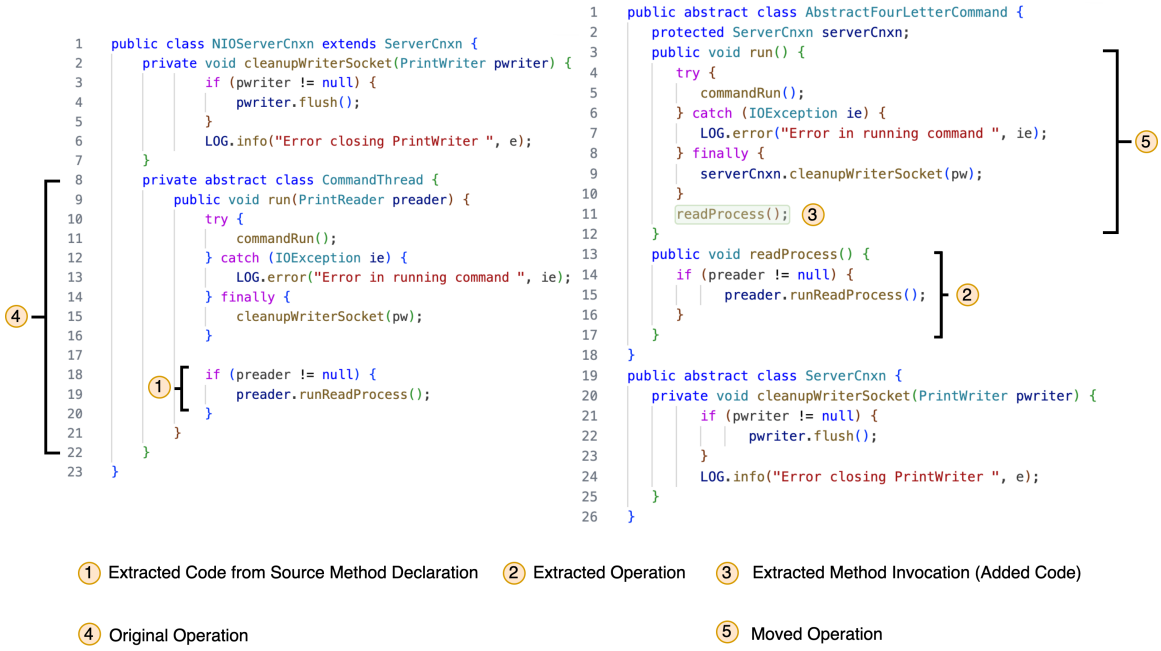

Figure 3.19: Pure Move Method Refactoring In Presence of Non-mapped Statements Caused by Overlapping Extract Method Refactoring

To address this, PurityChecker meticulously inspects non-mapped statement pairs that involve method calls. If it identifies that RENAME METHOD refactorings are responsible for these non-mapped statements, it tolerates these statements as direct consequences of overlapping RENAME METHOD refactorings.

Extract Method and Inline Method as Overlapping Refactorings:

When EXTRACT METHOD and INLINE METHOD refactorings overlap with method-level refactorings, they can lead to the introduction of non-mapped statements. This occurs due to the inherent nature of these refactorings.

During EXTRACT METHOD, a method is extracted, and this is naturally followed by a call to the newly extracted method. While the statements within the extracted operation might find their counterparts in the original method, the new method call in the refactored method remains non-mapped, as there is no equivalent in the original method.

Likewise, in the case of an overlapped INLINE METHOD refactoring, a similar situation unfolds in the original method. When a method is inlined, the statement containing the call to the inlined method is deleted, and this non-mapped change is permitted.

PurityChecker recognizes and allows the addition of non-mapped statements that represent the calling of the extracted method in the child version and the deletion of statements responsible for the call to the inlined method in the parent version. It is important to note that these non-mapped statements do not compromise the purity criteria of the method-level refactorings.

To provide a more detailed illustration of the scenario described earlier, we have made slight adjustments to one of the real-world examples previously discussed in Section 3.3. The altered case is visually presented in Figure 3.19.

As shown in the figure, as stated earlier, the `run` method was moved from the `CommandThread` class to the `AbstractFourLetterCommand` class. Additionally, the `cleanupWriterSocket` method was moved from the `NIOServerCnxn` class to its superclass, `ServerCnxn`, and this action is detected as a PULL UP METHOD refactoring by RefactoringMiner 2.0. Additionally, the `readProcess` method is also extracted from the run method, as indicated by the label ② in the figure. Therefore, there are two overlapping refactorings atop the MOVE METHOD refactoring. We would not reiterate how PurityChecker tags the replacement caused by the overlapping PULL UP METHOD refactoring as pure, as we have already discussed this in the previous section.

Concerning the overlapping EXTRACT METHOD refactoring, the call to the extracted method is marked as ③ in the figure. Since there are no corresponding statements in the original method, RefactoringMiner 2.0 rightly identifies this statement as a non-mapped statement, which is entirely reasonable. Looking at the figure, it is clear that adding this call as a non-mapped statement did not disrupt the purity criteria for the MOVE METHOD refactoring. Instead, it maintains the purity standards for the MOVE METHOD refactoring. This non-mapped statement is a call to a nested EXTRACT METHOD refactoring, and since their statements have been mapped by RefactoringMiner 2.0, PurityChecker accepts this statement addition and labels the MOVE METHOD as pure.

3.5 PurityChecker Structure and Functionality

PurityChecker, as an extension of RefactoringMiner 2.0, is intended to become an integral part of RefactoringMiner. Currently, PurityChecker is a separate public fork from RefactoringMiner, yet to be fully integrated.

In terms of PurityChecker's project structure, it features a crucial method called `isPure`. This method takes two main arguments: `umlModelDiff`, which contains information about the parent and child classes along with all the changes in a commit, and `refactorings`, which includes a list of refactorings performed in the given commit. These arguments are obtained by invoking RefactoringMiner on a specific commit (commit URL).

PurityChecker integrates with all RefactoringMiner APIs. RefactoringMiner offers multiple APIs for detecting refactorings, allowing users to work with locally cloned git repositories, directories containing Java source code, file contents as strings, and directly through the GitHub API. PurityChecker's compatibility lies in its ability to utilize the same arguments as specified earlier, ensuring that it functions in tandem with all RefactoringMiner API options.

To enhance the usability of PurityChecker, we have developed an API method within the "API.java" file. This API method simplifies the process by only requiring the commit URL as input and providing the purity output as a result.

The output of PurityChecker indicates whether the refactorings are pure or not, along with an automatically generated comment that explains why the changes are behavior-preserving or not. For example, in the case of an EXTRACT METHOD refactoring detected as pure due to the application of an overlapping INLINE VARIABLE refactoring, PurityChecker generates a comment like this: "Overlapped refactoring - can be identical by undoing the overlapped refactoring - Inline Variable," and assigns a purity value of true.

Chapter 4

Evaluation and Experimental Results

In this chapter, we present the outcomes of our extensive investigation into the purity of method-level refactoring operations. Our research encompasses a comprehensive empirical analysis of roughly 2,400 method-level refactoring operations found across over 600 commits, with a primary emphasis on determining their purity. To this end, we established two oracles that serve as our primary sources of ground truth to evaluate our tool: one is used for training our purity criteria rules and the other for testing them. Each method-level refactoring supported by PurityChecker underwent meticulous manual validation during our evaluation process, a significant contribution that characterizes our thesis. In the subsequent sections, we will provide a detailed account of our evaluation procedure.

Our evaluation is designed to address the following research questions:

RQ1. To what extent does PurityChecker accurately assess pure and impure refactorings?

RQ2. What are the key factors that render inaccuracies for PurityChecker?

RQ3. What is the distribution of pure and impure refactorings? Are there any discernible trends or patterns in this distribution?

RQ4. Among refactorings that are not identical in their bodies, how many were correctly identified as pure due to overlapping refactorings? What are the most popular overlapping refactoring types?

4.1 Oracle Creation

4.1.1 Dataset and Commit Selection

Our primary training dataset is based on RefactoringMiner 2.0's dataset (Tsantalis et al., 2020). This choice is rooted in two fundamental reasons. Firstly, PurityChecker operates as an extension of RefactoringMiner 2.0 and is highly dependent on it. Therefore, working with the same dataset utilized by this foundational tool offers several advantages. In the validation process, we maintained open lines of communication with the authors to address any potential issues or discrepancies within the original tool. Secondly, the refactorings within this dataset have been verified by a team of developers, including the authors of the paper. This meticulous validation process instills a higher level of confidence in our own dataset, as it has already been validated by domain experts.

In the case of our testing dataset, we conducted a thorough examination of various available datasets and opted for the one featured in the study by Pantiuchina et al. (Pantiuchina et al., 2020). The authors of this study provided an extensive dataset encompassing a wide spectrum of refactoring operations. Notably, their refactoring dataset was generated using the RefactoringMiner 2.0 tool for detecting refactorings. It is essential to highlight that the primary objective of their study was to detect and predict the motivations driving refactoring operations within version histories.

When selecting commits from our datasets, we adhered to specific criteria. As mentioned before, PurityChecker only supports method-level refactorings. Therefore, our first criterion was to choose commits that included at least one method-level refactoring. Additionally, we intentionally sought out commits with a relatively large number of refactorings. This deliberate choice allowed us to thoroughly evaluate the tool's robustness, particularly when dealing with overlapping refactorings, a core aspect of our study's innovation.

Our purity oracles are in form of JSON files, encompassing details about the repository, the commit, detected refactorings by RefactoringMiner, and purity information regarding method-level refactorings. This information includes whether the refactoring is manually validated as pure or impure, PurityChecker's detection of the refactoring as pure or impure, a comment generated by us during the manual validation process, and an auto-generated comment by PurityChecker explaining why the refactoring is identified as pure or impure.

4.1.2 Refactoring Purity Manual Validation

As discussed in Chapter 2, PurityChecker stands out as the first available tool designed for the automated assessment of refactoring purity, specifically identifying whether a method-level refactoring preserves the behavior of the code. In the realm of refactoring research, we have come across two other tools that aimed to tackle this problem, which are the RefDistiller and GhostFactor. Unfortunately, these tools are not publicly available, which made it impossible for us to perform direct comparisons. It is worth noting that while the DIFFCAT tool is accessible online, a direct comparison with PurityChecker was not feasible. This is because DIFFCAT analyzes code modifications in every code change, whereas PurityChecker operates within the scope of refactorings. In simpler terms, DIFFCAT assesses whether a code change is non-essential, while PurityChecker determines whether a refactoring is behavior-preserving, pure or not. Therefore, a comparison between these tools would not yield meaningful insights.

In light of this, we recognized that our evaluation phase needed to be exceptionally rigorous. Given the absence of comparative studies and tools, it was imperative to ensure that our tool's evaluation process was robust, instilling confidence among experts regarding the tool's reliability and trustworthiness.

We made a deliberate decision to manually validate each commit containing at least one method-level refactoring in our training dataset, which is sourced from RefactoringMiner 2.0's oracle. While we acknowledge that manual validation can be a time-consuming endeavor, our aim was to assemble the largest possible training dataset to bolster our tool's reliability. Since our purity rules are derived directly from real-world cases, increasing the number of validated cases provided invaluable insight into developer practices and purity patterns. Consequently, this approach yielded more accurate and dependable purity rules.

For manual validation of method-level refactoring cases, we employed diverse diff tools like GitHub, Visual Studio Code (VSCode), and RefactoringMiner's Chrome extension to visually inspect changes in method bodies. In cases where diffing the changes was impractical, as with certain refactorings occurring in different files, such as MOVE METHOD, we replicated both the original and moved methods into a local diff tool like VSCode. We meticulously examined changes in the

Table 4.1: Number of Validated Refactoring per Refactoring Type

Refactoring Type	**# Validated Refactorings**	
	Training	Test
Extract Method	925	93
Extract and Move Method	94	51
Inline Method	104	93
Move and Inline Method	13	6
Move Method	345	77
Move and Rename Method	107	26
Pull Up Method	282	53
Push Down Method	42	53

bodies to determine if the behavior was maintained.

Given PurityChecker's support for changes related to overlapped refactorings, instances where there were indications of changes due to overlapped refactorings prompted us to scrutinize the list of refactorings in the commit. This scrutiny aimed to determine if the change was behavior-preserving. Throughout manual validation, we documented comments outlining our justifications for the purity or impurity of a refactoring. In instances of complex cases where manual determination of a refactoring's purity was challenging, we engaged in discussions during our meetings with various expert software developers and researchers. This collaborative approach aimed to validate the accuracy of our manual assessments.

Our purity rules and algorithms are primarily influenced by the actual cases validated in our training dataset, leading us to follow a similar thinking process. The list of applied refactorings in a commit, provided by RefactoringMiner, also proved valuable in discerning whether a change resulted from the application of overlapped refactorings.

It is noteworthy to highlight that during the testing phase, PurityChecker underwent evaluation on a thoroughly isolated dataset. This rigorous testing approach was instrumental in ensuring the validity, generality, and comprehensiveness of our purity rules and algorithms.

During our manual validation process, we meticulously assessed over 2,400 method-level refactorings spanning across 600 commits. Interestingly, the number of refactorings we validated was at least double the number of validated method-level refactorings. This is due to our comprehensive

approach that involves examining not only method-level refactorings but also considering overlapping refactorings, which included a broader category of refactoring types. While this approach extended the duration of our validation and evaluation process, it was instrumental in fortifying our purity rules and ensuring the reliability of our metrics.

Table 4.1 displays the number of validated refactoring operations categorized by their types. For our testing oracle, we validated a minimum of 50 refactoring instances for each of the supported refactoring types. Notably, we increased the number of testing validations for EXTRACT METHOD refactorings, examining 93 cases in total. EXTRACT METHOD refactorings are of particular interest in the field of Software Refactoring, which led us to focus more on them. In line with this decision, the number of training cases for EXTRACT METHOD refactorings exceeds that of other refactorings, as sourced from the RefactoringMiner 2.0 oracle. It is worth emphasizing that for the MOVE AND INLINE, and MOVE AND RENAME METHOD refactorings, the instances available in our testing dataset were limited, totaling fewer than 50 cases. We meticulously incorporated all of these instances into our validation process.

Upon completing the implementation of PurityChecker and crafting our purity rules, we initiated the process of validating and evaluating our tool using the testing oracle. It is imperative to note that we strictly isolated the testing oracle from the tool's training data, ensuring that the testing metrics would remain reliable and untainted by any influence from the training dataset.

During our manual validation of the training oracle, we encountered instances where RefactoringMiner 2.0 failed to accurately report essential information required by PurityChecker. This information encompasses the details of refactoring operations within a commit, statement mappings within method-level refactorings, and the replacements that occurred within these mappings. Recognizing the critical significance of these resources in correctly identifying the purity of refactorings, we reported these issues to the authors of RefactoringMiner 2.0. Our aim was to benefit both tools by ensuring the accuracy and completeness of the provided data.

In the realm of Software Engineering research, there are numerous cases where the absence of a valid testing validation negatively impacts the quality and reliability of empirical studies. In this context, it is worth emphasizing that during the evaluation of our testing oracle, we deliberately froze the version of RefactoringMiner tool and based our testing on that specific version. This

Table 4.2: Evaluation Metrics per Supported Refactoring Types for Training and Testing Oracles

Refactoring Type	Metrics					
	Training			Testing		
	Precision	Recall	Specificity	Precision	Recall	Specificity
Extract Method	95.28	88.68	95.31	97.73	81.13	97.5
Extract and Move Method	92.31	73.47	93.34	100	93.02	100
Inline Method	98.28	95	97.73	96.97	86.49	98.21
Move and Inline Method	100	100	100	100	100	100
Move Method	100	92.03	100	100	82.81	100
Move and Rename Method	100	80.65	100	100	75	100
Pull Up Method	100	94.92	100	100	86.36	100
Push Down Method	100	87.18	100	100	100	100

approach guarantees the validity of our testing evaluation.

to

To summarize, one of our key contributions lies in the creation of two meticulously designed purity oracles that stand as valuable resources for future research endeavors.

4.2 Results and Discussion

This section offers an in-depth analysis of our tool's performance, featuring different evaluation metrics. These metrics are derived from our meticulously crafted training and testing oracles, which were developed through thorough manual validation. Furthermore, we will delve into our responses to the research questions outlined earlier in this section.

4.2.1 RQ1: Purity Detection Accuracy

Table 4.2 shows the performance of PurityChecker for both the training and testing phases. During our evaluation, we have mainly used three metrics as follows:

$$Precision = \frac{TP}{TP + FP}$$

Table 4.3: Weighted Average of Evaluation Metrics According to the Number of Each Refactoring Type

Training			**Testing**		
Precision	**Recall**	**Specificity**	**Precision**	**Recall**	**Specificity**
97.07	90.01	97	98.57	87.61	98.34

$$Recall = \frac{TP}{TP + FN}$$

$$Specificity = \frac{TN}{TN + FP}$$

Where True Positive (TP) highlights the number of refactorings that are genuinely pure and correctly identified as pure by PurityChecker, True Negative (TN) shows the number of refactorings that are genuinely impure and correctly identified as impure, False Negative (FN) shows the number of refactorings that are genuinely pure and erroneously categorized as impure by PurityChecker, and False Positive (FP) identifies the number of refactorings that are genuinely impure and erroneously categorized as pure by the tool.

In the realm of Software Engineering empirical studies, many researchers commonly rely on Precision and Recall as their primary evaluation metrics. However, we recognized the importance of a metric that not only considers true predictions of pure refactorings but also accounts for the accurate identification of impure ones. This recognition led us to include the Specificity metric, which gauges the tool's ability to correctly identify impure refactorings among all the actual impure refactorings in the dataset.

Due to the varying quantities of refactorings within each refactoring type, we have provided Table 4.3, offering a weighted average of the evaluation metrics. This table shows the actual overall performance of PurityChecker in both the training and testing phases.

As indicated in the provided tables, the recall rate is consistently lower than the other metrics by 3-15%. This discrepancy can be attributed to the inherent difficulty of minimizing false negatives (FNs) in our study. Specifically, detecting purity in scenarios involving multiple code transformations that affect numerous statement mappings and generate replacements, which are inherently pure, poses a substantial challenge. Consequently, we tend to experience a higher number of FN

cases than false positives (FPs) in most situations, leading to a slightly lower recall rate compared to other metrics.

An interesting observation is that the evaluation metrics within the testing oracle performed slightly better than those in the training oracle. This contrasts with the typical findings in most research studies. The rationale behind this divergence lies in the complexity of the refactoring operations present in our training oracle, which tend to be more intricate than those in our testing oracle. Furthermore, the relatively greater amount of time spent on validating the training samples compared to the testing samples aligns with this observation. We found that it took us approximately 1.4 times more time to validate a training sample compared to a testing sample.

4.2.2 RQ2: PurityChecker Inaccuracies

Our study is undeniably subject to inaccuracies, leading to several occurrences of both false negatives (FNs) and false positives (FPs) in our evaluation. These inaccuracies can be attributed to two main factors.

First, some code transformations are inherently complex, making it challenging to determine their purity solely through static source code analysis. In other words, relying solely on statement mappings and replacements is not sufficient to assess the purity of these refactorings in such scenarios. For instance, in certain situations, we had to evaluate intricate conditions or a small section of code to be able to automatically deduce whether the refactoring is behavior-preserving or not. Having the current resources in PurityChecker, it is not possible to detect the purity of some of these cases.

Secondly, since all required information comes from RefactoringMiner 2.0, and our tool is highly reliant on this tool, we inherit the limitations of RefactoringMiner 2.0. In specific cases, RefactoringMiner 2.0 fails to accurately report statement mappings and especially, the replacements within statement mappings in a given commit. Consequently, PurityChecker's analysis is compromised, resulting in FPs and FNs.

Notably, during our validation process, we encountered a considerably smaller number of false positive (FP) cases compared to false negatives (FN). This phenomenon primarily stems from the fact that identifying impure modifications is a comparatively simpler task when contrasted with

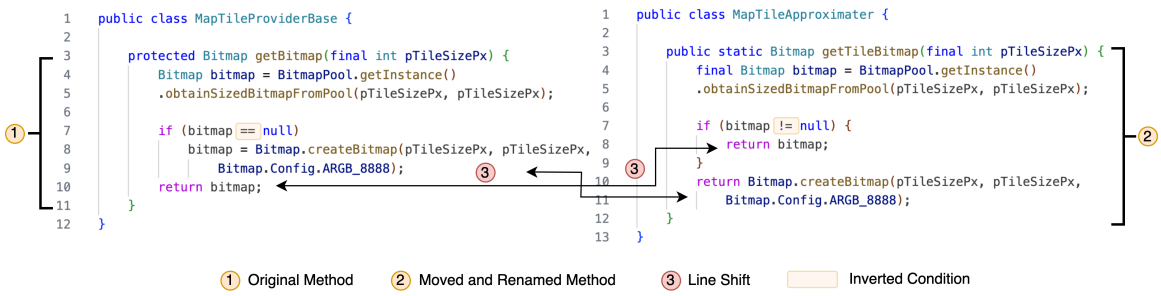

Figure 4.1: Pure MOVE AND RENAME METHOD Refactoring Mistakenly Reported as Impure by PurityChecker

recognizing pure code changes. In simpler terms, any code alteration within the context of a method-level refactoring is initially classified as impure, unless it can be reasonably justified by a limited number of purity rules and considerations. It is worth mentioning that the realm of impure code modifications encompasses a much broader spectrum than that of pure alterations.

To provide examples of such cases which highlight the PurityChecker's limitations, we have included two real-world examples from our training oracle, where PurityChecker produced false negatives.

The first example[1] sourced from the osmdroid project[2], which is depicted in Figure 4.1. In this example, the developer moved the `getBitMap` method from the `MapTileProviderBase` class to the `MapTileApproximater` class, and renamed the method to `getTileBitMap`, which is accurately reported as a MOVE AND RENAME MEHTOD refactoring by RefactoringMiner.

However, in addition to this modification, the developer made an adjustment by inverting the if statement's condition, resulting in a change in line sequence. This alteration led to the switching of two pairs of return statements, labelled as (3) in Figure 4.1. While RefactoringMiner accurately reported the statement mappings, replacements, and the list of refactorings, which built the resources of PurityChecker, there was a limitation in PurityChecker's resources – it did not possess the information about the sequence of lines.

Consequently, PurityChecker could not justify the replacement concerning the inverted condition. To classify this refactoring as pure and label the replacement as a pure code modification, it was essential to consider the order of lines. Justifiably, the replacement could only be considered as

[1] Visit the GitHub commit link
[2] https://github.com/osmdroid/osmdroid

Figure 4.2: Pure EXTRACT AND MOVE METHOD Refactoring Mistakenly Reported as Impure by PurityChecker

pure if it was known that the return statements had indeed been interchanged.

The second example[3] derived from the graphhopper project[4], which is depicted in Figure 4.2. Within this example, the developer extracted the `toDetailsString` method from a method with the same name in the `GraphHopperStorage` class, and moved it to the `BaseGraph` class, as indicated as (1) and (2) in the figure. This code transformation is accurately reported as an EXTRACT AND MOVE METHOD by RefactoringMiner.

Moreover, the developer moved the `nodeCount` attribute from the `GraphHopperStorage` class to the `BaseGraph` class, which is the destination class of the EXTRACT AND MOVE METHOD refactoring mentioned above. On top of this, the developer encapsulated the `nodeCount` attribute into the `getNodes` method, which is labeled as (5) in the figure. The last change, which can be defined as an (Move and) ENCAPSULATE ATTRIBUTE refactoring, have not been reported by RefactoringMiner.

PurityChecker fails to justify the replacement from `nodeCount` to `getNodes()`, which is directly attributed to the missing information about the ENCAPSULATE ATTRIBUTE refactoring. If PurityChecker had access to the information of the above mentioned ENCAPSULATE ATTRIBUTE refactoring, it could definitely justify the replacement, as PurityChecker supports tolerating code transformation resulting from the overlapping application of ENCAPSULATE ATTRIBUTE refactoring.

[3] Visit the GitHub commit link

[4] https://github.com/graphhopper/graphhopper

Table 4.4: Number of Actual Pure and Impure Refactoring Operation Instances

Refactoring Type	# Pure Instances	# Impure Instances
Extract Method	530	488
Extract and Move Method	92	53
Inline Method	97	100
Move and Inline Method	5	14
Move Method	352	70
Move and Rename Method	53	94
Pull Up Method	300	35
Push Down Method	88	7
Total	**1517**	**861**

4.2.3 RQ3: Distribution of Pure and Impure Refactorings

One of our primary research questions aimed to explore the prevalence of two categories of method-level refactorings: pure and impure. As described in Chapter 1, impure refactorings involve modifications that alter the behavior of the refactored code, while pure refactorings are those that transform the code while preserving its original behavior.

Based on our empirical analysis of the distribution of pure and impure refactoring operations, as detailed in Table 4.4, it appears that developers exhibit a preference for behavior-preserving refactorings over those that introduce changes affecting the program's behavior. In total, pure refactoring operations outnumber impure ones by a ratio of two to one.

Several factors contribute to this observation. Firstly, it can be attributed to the widespread usage of automatic refactoring tools. These tools tend to generate pure refactorings as they operate based on predefined mechanics, resulting in zero overlapping changes. Secondly, when developers engage in refactoring activities, they often consciously isolate the refactoring task from other concurrent changes, such as bug fixes. This separation streamlines the reviewing process, making it more straightforward to evaluate the impact of the refactoring itself.

Another intriguing observation from our study is that specific refactoring types, primarily those associated with moving code across different classes (MOVE METHOD, PULL UP METHOD, and PUSH DOWN METHOD), exhibit a significantly higher occurrence of pure refactorings compared

to impure ones. This noteworthy finding provides an intriguing avenue for future research to delve into the underlying reasons behind this phenomenon.

4.2.4 RQ4: Frequency of Overlapped Refactorings Causing Pure Refactoring Operations - Most Popular Overlapping Refactoring Types

As our primary innovation, we have equipped PurityChecker with the capability to identify pure code modifications resulting from the application of overlapping refactorings within method-level refactorings. We have thoroughly explained these scenarios in Sections 3.3 and 3.4, demonstrating how different overlapping refactorings can lead to the introduction of pure code modifications.

In this section, we will present the findings of our empirical study, focusing on the prevalence of overlapping refactorings and their contribution to pure code changes.

Table 4.5: Frequency of Overlapping Refactorings within Method-level Refactorings

Refactoring Type	# Cases (%)
Rename Variable	91 (29.6%)
Add Parameter	66 (21.4%)
Rename Class	51 (16.6%)
Extract Method	20 (6.5%)
Rename Method	13 (4.2%)
Replace Accessor Call with Direct Field Access	13 (4.2%)
Remove Parameter	12 (3.9%)
Extract Variable	11 (3.6%)
Rename Attribute	7 (2.3%)
Inline Variable	6 (2%)
Move Attribute	6 (2%)
Merge Variable	3 (1%)
Split Conditional	2 (0.6%)
Move Method	2 (0.6%)
Pull Up Method	2 (0.6%)
Encapsulate Attribute	1 (0.3%)
Merge Conditional	1 (0.3%)
Inline Method	1 (0.3%)

Within our training oracle, we encountered a total of 1053 TP (True Positive) cases, indicating refactorings correctly identified as pure by PurityChecker. Within these cases, 605 refactorings are identical in their bodies and 448 refactorings involved changes within their bodies. In other words, more than 42% of TP cases in our training oracle exhibited non-identical statements.

Among these 448 cases, 204 had at least one change within their body, which is directly related to the application of overlapping refactorings, marking a substantial portion of cases as we expected. This underscores our decision to include analysis of changes resulting from overlapping refactorings in our tool.

Additionally, we observed 304 cases that contained behavior-preserving changes resulting from the application of overlapping refactorings in both true positive and true negative cases, which form our ground truth. This analysis provides insights into the most common types of overlapping refactorings within our training dataset.

Table 4.5 presents the occurrence rates of different types of overlapping refactorings in our training dataset. The table reveals that Rename Variable is the most frequently observed overlapping refactoring type, followed by Add Parameter, Rename Class, Extract Method, and Rename Method refactorings, which make up the top five in terms of prevalence

Negara et al. addressed a research question in their study (Negara et al., 2013) concerning the most commonly performed refactorings. According to their findings, the most prevalent refactoring is Rename Variable, with Extract Method, Rename Method, Rename Class, and Extract Variable refactorings also ranking among the top 10 most popular refactorings.

(Wang et al., 2018)

Our analysis aligns with the findings of the mentioned study regarding the most frequently used overlapping refactoring types. This suggests that the adoption of overlapping refactorings is intrinsically intertwined with the broader practice of applying refactorings, and interestingly, the popularity of applied refactoring types is more or less the same as the popularity of the applied overlapping refactoring types.

In other words, the relatively frequent use of overlapping refactoring during method-level refactorings, coupled with our findings aligning with the study's results, implies that developers regularly

employ overlapping refactorings alongside other refactorings. This connection indicates that developers, in their refactoring endeavors, frequently encounter scenarios where overlapping refactorings are not only necessary but also advantageous, contributing to their widespread use.

Chapter 5

Threats to Validity

5.1 Internal Validity

Our research is susceptible to internal threats, mainly concerning biases, errors, and limitations originating from the process of building purity rules. Here, we address these potential threats and how they can affect the generalizability and accuracy of the PurityChecker tool.

Building purity rules relied on our own validated oracle, and these rules have been gradually refined over time through extensive validation. This iterative process poses potential biases, including overfitting rules to our specific dataset. Such biases may limit the tool's adaptability to diverse datasets and hinder its generalizability. As we evaluated our tool on a fully separate dataset and the outcoming results are promising, we have managed to address this threat to some extent.

The generation of purity rules inherently involves a manual, judgment-based approach. Human subjectivity and individual perspectives play a significant role in this process. In some cases, determining the purity of refactorings can be subjective, as it requires an understanding of the developer's thought process during the code transformation. Without insight into the developer's exact intentions, making definitive claims about the purity of certain changes becomes challenging.

5.2 Construct Validity

The threats concerning the construct validity of our research mainly focus on the reliance of our tool to RefactoringMiner 2.0 and the accuracy of this tool.

PurityChecker heavily relies on RefactoringMiner 2.0 to provide information about refactorings within a codebase. A threat arises if RefactoringMiner incorrectly identifies or misclassifies refactorings or statement mappings and the replacements among them. For example, if a complex refactoring operation is misunderstood or underreported by RefactoringMiner, PurityChecker may inherit these inaccuracies and produce flawed results. As a result, a discrepancy can emerge between what RefactoringMiner identifies and what actually occurs within the codebase. This issue could lead to false positives, where impure refactorings are incorrectly classified as pure, or false negatives, where pure refactorings are erroneously marked as impure.

5.3 External Validity

One external threat to the validity of your research is the potential impact of shifts in programming practices and coding standards within the software development community. The coding landscape is not static; it evolves with emerging best practices, methodologies, and paradigms. As such, what constitutes a “pure” or “impure” refactoring can change over time. New coding patterns and idioms that become popular may not align with the existing criteria used by PurityChecker.

Moreover, the evolution of Java programming language is another external threat to our study. Java continuously develop and introduce new features, constructs, and paradigms. These changes can significantly impact how code is structured and how refactorings are conducted.

Chapter 6

Conclusion and Future Works

In the realm of software engineering, refactoring is a foundational practice embraced by developers to enhance code quality and maintain codebases effectively. The ability to discern whether a refactoring operation maintains code behavior, in other words, whether it is 'pure' or not, holds substantial value across various domains.

For code reviewers, this distinction streamlines the review process, providing profound insights into code health while significantly reducing review time. In the realm of empirical research within Software Engineering, the concept of refactoring purity opens up a wealth of opportunities, enabling comprehensive large-scale studies leveraging these insights. Furthermore, practical and research tools in software development can harness this knowledge to enrich their capabilities.

In the scope of this research, we introduced a publicly accessible tool known as PurityChecker, designed for the automated assessment of method-level refactoring purity. A comprehensive review of the relevant literature underscores PurityChecker's pioneering status in this specialized domain of Software Engineering tools.

Our extensive evaluations demonstrate PurityChecker's remarkable capabilities, achieving precision and recall rates exceeding 90% across diverse scenarios. This tool's robust performance reaffirms its position as a cutting-edge solution in the field, providing invaluable insights into the purity of refactoring operations.

Another substantial contribution arising from this thesis is the extensive empirical study we conducted on the purity of refactoring operations. This comprehensive investigation yielded two

manually-validated oracles. These oracles, with their carefully verified data, can serve as invaluable resources for future research endeavors within the domains of Software Engineering and Software Refactoring.

In terms of potential future avenues in this field, there are promising directions to explore. Incorporating a variety of tools that provide multiple sources of statement mappings and replacements can enhance the reliability and robustness of our tool. Furthermore, while our current work focused on defining purity for method-level refactorings, there is room for expansion into supporting variable and class-level refactorings in future research efforts.

References

Agnihotri, M., & Chug, A. (2020). A systematic literature survey of software metrics, code smells and refactoring techniques. *Journal of Information Processing Systems*, *16*(4), 915–934.

Agnihotri, M., & Chug, A. (2022). Understanding refactoring tactics and their effect on software quality. In *2022 12th international conference on cloud computing, data science & engineering (confluence)* (pp. 41–46).

Alves, E. L., Song, M., & Kim, M. (2014). Refdistiller: a refactoring aware code review tool for inspecting manual refactoring edits. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering* (pp. 751–754).

Bacchelli, A., & Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In *2013 35th international conference on software engineering (icse)* (pp. 712–721).

Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., & Strollo, O. (2012). When does a refactoring induce bugs? an empirical study. In *2012 ieee 12th international working conference on source code analysis and manipulation* (pp. 104–113).

Black, A. P., & Murphy-Hill, E. (2007). Why don't people use refactoring tools?

Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D., & Garcia, A. (2017). How does refactoring affect internal quality attributes? a multi-project study. In *Proceedings of the xxxi brazilian symposium on software engineering* (pp. 74–83).

Dig, D., Comertoglu, C., Marinov, D., & Johnson, R. (2006). Automated detection of refactorings in evolving components. In *Ecoop 2006–object-oriented programming: 20th european conference, nantes, france, july 3-7, 2006. proceedings 20* (pp. 404–428).

Fluri, B., Wursch, M., PInzger, M., & Gall, H. (2007). Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, *33*(11), 725–743.

Fowler, M. (2018). *Refactoring*. Addison-Wesley Professional.

Henkel, J., & Diwan, A. (2005). Catchup! capturing and replaying refactorings to support api evolution. In *Proceedings of the 27th international conference on software engineering* (pp. 274–283).

Kawrykow, D., & Robillard, M. P. (2011). Non-essential changes in version histories. In *Proceedings of the 33rd international conference on software engineering* (pp. 351–360).

Kaya, M., Conley, S., Othman, Z. S., & Varol, A. (2018). Effective software refactoring process. In *2018 6th international symposium on digital forensic and security (isdfs)* (pp. 1–6).

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

Kim, M., Zimmermann, T., & Nagappan, N. (2012). A field study of refactoring challenges and benefits. In *Proceedings of the acm sigsoft 20th international symposium on the foundations of software engineering* (pp. 1–11).

Kim, M., Zimmermann, T., & Nagappan, N. (2014). An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, *40*(7), 633–649.

Kolb, R., Muthig, D., Patzke, T., & Yamauchi, K. (2005). A case study in refactoring a legacy component for reuse in a product line. In *21st ieee international conference on software maintenance (icsm'05)* (pp. 369–378).

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

Murphy-Hill, E., & Black, A. P. (2007). Why don't people use refactoring tools? In *Proceedings of the 1st workshop on refactoring tools* (pp. 61–62).

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

Murphy-Hill, E., Parnin, C., & Black, A. P. (2011). How we refactor, and how we know it. *IEEE*

Transactions on Software Engineering, *38*(1), 5–18.

Murphy-Hill, X. G. E. (n.d.). Manual refactoring changes with automated refactoring validation.

Negara, S., Chen, N., Vakilian, M., Johnson, R. E., & Dig, D. (2013). A comparative study of manual and automated refactorings. In *Ecoop 2013–object-oriented programming: 27th european conference, montpellier, france, july 1-5, 2013. proceedings 27* (pp. 552–576).

Nouri, P. (2023). *Puritychecker.* (https://github.com/pedramnoori/RefactoringMiner?organization=pedramnoori&organization=pedramnoori)

Opdyke, W. F. (1990). Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proc. of 1990 symposium on object-oriented programming emphasizing practical applications (sooppa).*

Opdyke, W. F. (1992). *Refactoring object-oriented frameworks*. University of Illinois at Urbana-Champaign.

Ouni, A., Kessentini, M., Ó Cinnéide, M., Sahraoui, H., Deb, K., & Inoue, K. (2017). More: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. *Journal of Software: Evolution and Process*, *29*(5), e1843.

Pantiuchina, J., Zampetti, F., Scalabrino, S., Piantadosi, V., Oliveto, R., Bavota, G., & Penta, M. D. (2020). Why developers refactor source code: A mining-based study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *29*(4), 1–30.

Prete, K., Rachatasumrit, N., Sudan, N., & Kim, M. (2010). Template-based reconstruction of complex refactorings. In *2010 ieee international conference on software maintenance* (pp. 1–10).

Ratzinger, J., Fischer, M., & Gall, H. (2005). Improving evolvability through refactoring. In *Proceedings of the 2005 international workshop on mining software repositories* (pp. 1–5).

Sillitti, A., & Succi, G. (n.d.). A case study on the impact of refactoring on quality and productivity in an agile team.

Silva, D., da Silva, J. P., Santos, G., Terra, R., & Valente, M. T. (2020). Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, *47*(12), 2786–2802.

Silva, D., Tsantalis, N., & Valente, M. T. (2016). Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering* (pp. 858–870).

Silva, D., & Valente, M. T. (2017). Refdiff: Detecting refactorings in version histories. In *2017 ieee/acm 14th international conference on mining software repositories (msr)* (pp. 269–279).

Tsantalis, N., Ketkar, A., & Dig, D. (2020). Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, *48*(3), 930–950.

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., & Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th international conference on software engineering* (pp. 483–494).

Vakilian, M., Chen, N., Negara, S., Rajkumar, B. A., Bailey, B. P., & Johnson, R. E. (2012). Use, disuse, and misuse of automated refactorings. In *2012 34th international conference on software engineering (icse)* (pp. 233–243).

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

Wang, K., Zhu, C., Celik, A., Kim, J., Batory, D., & Gligoric, M. (2018). Towards refactoring-aware regression test selection. In *Proceedings of the 40th international conference on software engineering* (pp. 233–244).

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

Xing, Z., & Stroulia, E. (2007). Api-evolution support with diff-catchup. *IEEE Transactions on Software Engineering*, *33*(12), 818–836.