Fine-Grained Source Code Tracking and Visualization in Commit History

Mohammed Tayeeb Hasan

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

September 2023

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Mohammed Tayeeb Hasan**

Entitled: **Fine-Grained Source Code Tracking and Visualization in Commit History**

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

Master of Computer 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. Diego Elias Costa

_____ Examiner
Dr. Peter C. Rigby

_____ Supervisor
Dr. Nikolaos Tsantalis

Approved by _____
Dr. Leila Kosseim, Graduate Program Director

_____ 2023

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

Abstract

Fine-Grained Source Code Tracking and Visualization in Commit History

Mohammed Tayeeb Hasan

CodeTracker is the current state-of-the-art program element change history generator with a reported precision and recall of 99.9% in method and variable tracking [1]. In this thesis, we extend the granularity of CodeTracker to support the tracking of control-flow blocks (e.g., `for`, `while`, `if`, `try`, `catch`, etc.) with a precision and recall of 98.12% and 97.62% respectively, providing researchers and developers with finer-grained information about the evolution of source code. We accompany this extension with a manually validated oracle, which includes the change histories of 1280 code blocks. These code blocks are contained within 200 methods from 20 open-source Java projects (10 methods from each project) comprising the method change history oracle created by Grund et al. [2]. We also present a code change history visualization and navigation tool for CodeTracker, named CodeTracker Visualizer, that overlays the GitHub user interface with change history information enabling users to track code elements directly from the commit page by simply selecting the desired code element. Finally, we compare CodeTracker's block tracking precision and recall using two different tools that provide statement mappings, namely RefactoringMiner [3, 4], the current state-of-the-art refactoring detection tool, and GumTree [5], the current state-of-the-art Abstract Syntax Tree (AST) Diff tool. The enhanced version of CodeTracker along with the extended oracle are made publicly available to facilitate reproducibility and future research on code element tracking techniques [6].

Acknowledgments

I would like to express my gratitude to several individuals who have played a significant role in the completion of this thesis:

First and foremost, I extend my heartfelt thanks to my research advisor, Dr. Nikolaos Tsantalis. Your guidance, expertise, encouragement, and constant support have been invaluable throughout this journey. Your mentorship has helped shape my research skills and academic perspective.

I would like to express my heartfelt gratitude to my esteemed committee members, Dr. Diego Elias Costa and Dr. Peter C. Rigby. Your valuable critiques, constructive feedback, and encouraging words have greatly enriched the quality of this thesis. Your insights have been instrumental in refining the research and enhancing its overall impact.

To my parents, I offer my sincerest appreciation for your unwavering belief in me and your continuous encouragement. Your support has been a driving force behind my accomplishments.

I want to acknowledge my brothers, Tayseer and Tawseef, for their constant presence and encouragement. Your support has provided me with the strength to overcome challenges and pursue my goals.

I am grateful to my colleagues for their collaborative efforts, insightful discussions, and constructive feedback. Your contributions have enhanced the quality and depth of this research.

Once again, thank you to everyone who has played a part in this journey. Your support and contributions have been pivotal in the successful completion of this thesis.

With immense gratitude,

Mohammed Tayeeb Hasan

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A code block, simply referred to as a block, is defined as a lexical structure of source code that is grouped together [15]. In modern structured programming languages, code blocks are extensively used. In a survey conducted by Stack Overflow in 2022 [16], over 70,000 software developers were asked questions about their programming language usage, preferred tooling, and development environments to understand the current dynamic of software engineering. This survey shows that Git and GitHub were the top Version Control Systems (VCS) and VCS hosting platforms of choice. Developers store incremental code changes in their VCS to maintain a log of changes occurring on the code base. While VCS is adequate for its primary purpose of storing and retrieving source code revisions, developers often need to browse the commit history of a project to understand the evolution a code element has undergone [17, 18] and explore common changes being made in a code repository [19], especially in the context of code reviews [20].

A study conducted by Negara et al. has shown that the information stored in a VCS is imprecise and incomplete for analysis [21], which opens up the need for external enhancements and analysis tools to help understand VCS information. There have been attempts made to enhance VCS [22] to provide better storage and understanding of data. Freese attempted to enhance source code management systems by making them refactoring-aware [23]. This shows that there is a need to enable better ways to understand data stored in a VCS.

Grund et al. [2] surveyed professional developers and discovered that method-level granularity is desired for source code history generation, apart from existing file-level implementations. Among

these developers, around 75% responded positively about obtaining change histories at a block-level granularity. Grund et al. [2] also points out that developers are interested in examining histories at program element granularities (i.e., method, field) other than just the file or textual range level as is supported by most tools.

Grund et al. [2] also show that other research projects exist, which propose history queries on code elements down to the block level (e.g., APFEL [24]). In the paper presenting APFEL [24], Zimmermann directly mentions the intention to implement block-level tracking.

Yoon et al. [25] propose that users may be interested in investigating how an `if` block was originally written, or how an expression within a single line has evolved. They developed a tool named Azurite that tracks edits being made on a class and enables selective undo on any desired code fragment. They show that there are other systems that also aim to provide history search, or *history slicing* as they define it [25].

Servant and Jones [26] present a history-slicing tool named CHRONOS, which uses version control snapshots to trace back to find which commits affected a certain area of code. The search scope of CHRONOS can be as small as a single line. Although both these tools (Azurite [25] and CHRONOS) are not refactoring-aware, and thus may lack in accuracy, they show the need for finer-grained source code tracking. [24, 26, 25, 1, 27, 28, 29, 30, 31, 32]

LaToza and Myers [33] surveyed professional developers at Microsoft and found out that they wanted to know the source code history of a block of code (e.g., `for` loop), and would like to go all the way to its introduction rather than its most recent change. They report that developers were also interested in code change history at the level of a code snippet (e.g., a single statement). This corroborates identical findings made by Holmes et al. [34], who also surveyed professional developers in their study.

Fritz and Murphy [35] surveyed eleven professional software developers to recognize questions that developers ask but have no resources that can help answer. Out of the 78 recorded questions, 20 code-specific questions were highlighted in the paper. Among the questions asked, developers were interested in knowing who originally wrote a piece of code (e.g., `if` statement) and who modified it last. These questions show that there is a need for the kind of information that CodeTracker provides, and tracking support at the code snippet level is highly desired.

Ko et al. [36] surveyed seventeen software developers and logged their activities minute by minute. Among the sort of information that developers sought, they found that developers wanted to know more about the implementation evolution-history of a piece of code, i.e., to understand how the code has been implemented over time ("d3. Why was this code implemented this way?" [36]), so as to derive historical reasoning for its current implementation. Another interesting point made in this paper is that during bug fixes, developers need specific code change history to analyze whether the error was anticipated by the designer and explicitly ignored or whether it was overlooked. Thus, having block-level source code history can speed up bug-fixing efforts in cases where bugs are known to be present in a specific block rather than the entire method.

In another study, LaToza et al. [19] conducted multiple surveys and interviews with professional software developers and found that the majority of developers agreed that understanding the history of a specific piece of code was a serious problem for them. This may be due to having a large number of commits to comb through. Having access to a granular stack of changes to review when only a specific code block or statement is of concern could help solve this problem. Confirming the findings presented by Fritz et al. [35], this paper indicates that developers are interested in knowing why and by whom a piece of code was introduced to a codebase. Having block-level history makes such information more accessible, as having only method/file-level history would require additional manual efforts to discover such information.

Thus, it is evident that there is a need for block-level code element tracking that we resolve with this work.

CodeTracker was developed by Jodavi and Tsantalis [1]. It had the ability to track classes, methods, variables, and attributes for Java projects with 99.9% precision and recall. In this thesis, we enhance CodeTracker to possess the ability to track code blocks as our major contribution. We also include a visualization tool for code change history navigation.

Why code blocks are challenging to track:

A significant challenge that occurs when tracking code blocks is the fact that they have an unnamed structure and no signature. This is unlike other code elements (e.g. methods or type declarations), which have a standard well-defined structure and signature. Another challenge is that many different code blocks in a class may have similar code, so running a string similarity check will

spawn a lot of potential matches, and accurately zeroing down to the correct block is challenging. A block can also change between node types (e.g. `if-else-if` to `switch`) or transition to and from a pipeline (e.g. `for-loop` to `forEach()`). In fact, there is no limit to the number and type of changes possible in the case of a code block change. These characteristics of a code block cause an uptick in the effort required to accurately and efficiently track code blocks, and this is the reason why none of the other tools mentioned in Section 2.1 recognize or track a code block. In Section 4.1 we discuss our approach to constructing signatures for a code block to tackle the issue of code blocks lacking a definite signature.

The main novelty of our solution to the problem is that our tool (CodeTracker) is the first tool in the literature that can track the change history of code blocks in a fully refactoring-aware fashion, meaning that it can continue tracking a code block even if it has been moved to another method (e.g., EXTRACT METHOD, INLINE METHOD, MERGE METHOD, SPLIT METHOD refactoring), or moved to another file (e.g., MOVE METHOD, EXTRACT CLASS refactoring). Other tools suffer from disruptions in the change history of the tracked elements in the presence of refactorings, due to *untracked changes* [32]. According to Hora et al. [32], 25% of classes and methods have at least one untracked change (i.e., move, rename, extract, inline refactoring) in their histories.

Our complete contributions presented in this thesis are as follows:

(1) We enhance the granularity of tracking for CodeTracker to track code blocks with a precision and recall of 98.12% and 97.62%, respectively.

(2) We fix inaccuracies present in CodeTracker's method, variable, and attribute oracles [1], enhance the complete oracle with additional change history information, and add 65 new variables and their change histories to the variable oracle.

(3) We extend the oracle presented by Jodavi and Tsantalis [1] to include 1280 new instances of blocks and their change histories.

(4) We present a novel tool, CodeTracker Visualizer, that enables users to browse the code change history of code elements directly within the GitHub user interface, significantly boosting the efficiency of CodeTracker's usage.

(5) We extended CodeTracker Visualizer to be used as an oracle validation tool that significantly cuts down the time required to manually verify change histories when creating our oracle.

(6) We present two versions CodeTracker's BlockTracker API, one integrated with RefactoringMiner [3, 4], and the other with GumTree [5], both of which can be used to generate statement mappings between two commits.

(7) We compare the precision and recall of the aforementioned BlockTracker API versions, and show that the version integrated with RefactoringMiner has superior accuracy over the one integrated with GumTree (+9.3% precision and +18.2% recall).

(8) We developed a REST API that allows for CodeTracker to be used as a web service.

The rest of this thesis is structured as follows: Chapter 2 reviews relevant literature and toolings and their limitations. Chapter 3 provides background information pertaining to the rest of the chapters and helps understand the ideas conveyed in this thesis. Chapter 4 delves into the proposed research approach, and Chapter 5 deals with the implementation details of the tooling contributions made in this thesis. The evaluation is performed in Chapter 6, and the threats to validity are discussed in Chapter 7. Finally, we conclude this thesis in Chapter 8 with some potential use cases and future work.

Chapter 2

Related Work

This chapter of the thesis reviews existing tools developed to perform code-change history tracking, detailing both fine-grained and coarse-grained implementations. We then discuss the shortcomings of these approaches.

2.1 Existing Tooling

Several tools have been developed over the years that aim to track code change history and understand the evolution of codebases as a whole. In this section, we shall discuss a few of the approaches that have been implemented as tools.

Grund et al. [2] developed the tool CodeShovel, which helps unearth code change histories at the method level and can detect histories for 90% of methods and 97% of all method changes. Along with the tool, they present a web application that a user can access to interact with CodeShovel's APIs. Moreover, they contribute an oracle of code change histories for 200 methods from 20 open-source repositories. CodeShovel is specifically designed to operate on-demand, without the need for any upfront processing, and it can effectively handle most of the latest code transformations that occur in modern Java projects. Its method-matching algorithm utilizes text similarity and metrics to detect changes in methods across different versions of files. Extensive evaluations demonstrate that CodeShovel surpasses other prominent tools in both accuracy and runtime performance, making it a valuable asset for various industrial development tasks. This tool is, by far, the closest related to

CodeTracker in terms of the problem it tries to solve.

Zimmermann [24] developed a plugin for Eclipse named APFEL, which aims to compute fine-grained changes from version archives in a database. It is built upon the Eclipse infrastructure and uses CVS as the SCM system and Java as the programming language. APFEL complements previous tools by capturing more detailed changes in software artifacts. It uses token sets to represent syntactic content, allowing efficient comparison of revisions. The database stores token changes, offering insights into software evolution. The limitations of APFEL include its inability to handle renaming directly and the approximation of method signatures based on the argument count. They perform case studies that showcase the benefits of APFEL in identifying crosscutting concerns, pairs of frequently inserted variable names, and variable renaming evidence.

Yoon et al. [25] developed a tool named Azurite that tracks edits made on a class and enables selective undo on any desired code fragment. Azurite is available to use as an Eclipse plugin. The tool provides two views for visualization of code change history: the Timeline View and the Code History Difference View. These views allow users to interactively explore code changes. They show that fine-grained code change histories are valuable for answering developer queries and facilitating backtracking tasks. They highlight their plans for future work, such as extending support to team development environments, introducing advanced history search features, and providing additional editor commands for past operations.

Servant and Jones [26] developed CHRONOS, a history-slicing Eclipse plugin that traces back in code history using version control snapshots to find commits that affected a specific section of code. It builds a history graph, selects slicing criteria, and computes and visualizes history slices for developers to explore and analyze code changes over time. The history graph is a multipartite graph representing the evolution of each file. Each node in the graph represents a line of code in a specific revision, and each node is linked to nodes in the previous and/or following revisions. The search scope of CHRONOS can range from multiple lines down to a single line, which means that developers can select a set of lines of interest (possibly a code block) from a specific revision as a slicing criterion. The design of CHRONOS is developed based on utilizing coarse-grained version control history (i.e., file level granularity and line numbers).

Looking at some fine-grained alternatives, Historage was developed by Hata et al. [28] that aims

to track method moves within Java projects. To be able to do so, Historage creates a fine-grained Git repository and adds each method from the project to a single file. Doing so allows it to utilize the tracking mechanisms provided by Git, and it can successfully track methods across commit history. Historage stores entire histories of fine-grained entities, including renaming changes, which is beneficial for in-depth software evolution analysis. The paper provides an evaluation of Historage on five open-source software projects and demonstrates its ability to accurately identify renaming changes. By performing their evaluations, they found that the tool achieves high precision and recall for rename identification, with more than 97% correct matches when similarity is above 30%.

Another fine-grained approach presented as a tool named FinerGit by Higo et al. [27], aims to overcome the shortcomings of Historage [28]. Historage's tracking accuracy is limited for small methods that undergo a rename or move method refactoring, leading to the proposal of a new technique called FinerGit to enhance method trackability. Their approach is to put a single token of Java methods per line, i.e., each line contains only one token. They present two heuristics to handle false tracking cases as well. They ran their tool on 1,768K methods in 182 open-source repositories to test their accuracy. The paper concludes by discussing the broader applications of FinerGit in various research domains and the potential impact on experimental results.

Below, we shall explore some related work done concerning code review simplification and streamlining, which CodeTracker Visualizer aims to do.

Huang et al. [37] developed CLDiff, which proposed a new methodology for code differencing. They included a web application that allows researchers and practitioners to visualize the code difference generated by CLDiff. CLDiff takes source code files before and after changes as inputs and performs three steps: pre-processing the code, generating concise code differences by grouping fine-grained changes, and linking related differences based on pre-defined links. The research evaluates CLDiff through experiments with 12 Java projects and a human study with 10 participants, demonstrating its accuracy, conciseness, performance, and usefulness in understanding code changes.

Lee et al. [38] developed Tempura, an eclipse plugin that aims to add a *temporal dimension* to IDEs. By doing this, they propose the extension of code completion and navigation abilities of IDEs by utilizing code history. This approach adds information in the code completion feature

about the code elements that were removed and provides navigation to them as well. Temporal code navigation allows developers to search for and open any type from any revision of the project, even deleted types, using the Open Type in History dialog. A controlled user study was conducted to compare Tempura with EGit, and the results showed that Tempura helps developers learn about code history more accurately and efficiently.

Ge et al. [39] present a refactoring-aware code review tool, built upon the Eclipse development environment, aimed to simplify the pull request review process by muting out refactored code, so that reviewers can focus on code that is prone to introducing bugs. The tool automatically determines which changes in a code set are refactorings, helping developers differentiate between refactoring and non-refactoring changes. Similar to CodeTracker Visualizer, this tool aims to simplify and streamline the code review procedure.

Alves et al. [40] developed RefDistiller, a refactoring-aware code review tool that can help developers detect potential errors in manual refactoring edits by pointing out incomplete refactorings and extra edits made within pure refactorings. This tool, available as an Eclipse plugin, aims to tackle the tedious code review process that almost all software developers perform. RefDistiller utilizes a tool named RefFinder to compare two different versions of a code element and then detects the types and locations of potential refactorings made by the author. Two key techniques used by RefDistiller are RefChecker and RefSeparator. RefChecker detects missing edits by checking required code modifications and preservation of method or field reference bindings. RefSeparator detects extra edits by comparing a manual refactoring version with an equivalent pure refactoring version generated using Eclipse's automated refactoring API.

Brito and Valente [41] developed RAID, a refactoring-aware code differencing tool, which is available as a Chrome extension, and integrates with the GitHub user interface to provide a list of refactorings performed in the commit being reviewed. It also provides a pop-up that properly aligns the code element from both the current and parent versions and reduces the cognitive effort required for reviewing and detecting refactorings. They conducted a field experiment involving eight professional developers using RAID for three months and showed significant results. RAID effectively reduced the number of lines needed for code review during refactorings.

2.2 Limitations of Existing Tooling

Although several tools tackle the problem of code change history, their primary focus is on the code base as a whole and not specific code elements, with CodeShovel [2] being the only exception to this. Below, we shall discuss the shortcomings that these tools have and that CodeTracker overcomes.

(1) **Lack of Code Change History Tracking:** Although the tools mentioned above perform code differencing for two versions of a class, they inherently do not follow code elements through the commit history of a project. These tools have a limited scope, as they only consider one version and its parent at a time, and they do not track changes continuously through the commit history until the introduction. Tools that have this shortcoming are Azurite [25], RAID [41], CHRONOS [26], Tempura [38], CLDiff [37], and RefDistiller [40].

Azurite [25] is designed to selectively reverse code changes made in a single class. Although it can perform a multi-file undo to track a code element when it has been moved between multiple files, the user is expected to bear the responsibility of ensuring that all the files are reverted to the correct point in time and are in a working condition.

RAID [41] detects refactorings between two versions but does not deal with tracking code elements throughout the commit history. Although tedious, one can manually track the code element through the commit history one commit at a time in a more effective manner than using `git blame` due to its refactoring-aware nature.

CHRONOS [26] provides coarse-grained searching, and hence it is not accurate when tracking a large number of small edits [25]. Although it can target a specific area of statements, and one could set a block as the scope of searching, it does not guarantee to track the block in the presence of refactorings that may change the location of the block by moving it to another method or even another file. Such limitations do not exist with CodeTracker [1] as it groups statements as the respective code elements and is completely refactoring aware.

Although Tempura [38] allows for navigation to code elements that were present in the code change history of a project, they are presented as a collection and are not mapped to code

elements that are currently present. Thus, an evolution chain cannot be constructed. The information is simply used to present more details in the auto-complete feature of the Eclipse IDE.

(2) **Lack of Block Tracking or Differencing:** Existing tools perform code differencing between two versions for code elements but do not specifically focus on code blocks. Tools like CodeShovel [2], Historage [28], FinerGit [27], APFEL [24], Tempura [38], RAID [41], and RefDistiller [40] can track methods, while CHRONOS [26] does not track based on code elements but based on line numbers in a file. Our extension of CodeTracker [1] recognizes and tracks code blocks as distinct program elements, which results in the generation of accurate change history.

(3) **Lack of On-Demand Computation:** Unlike any of the other tools mentioned above, Azurite [25] requires the installation of an Eclipse plugin before edits are made to a file since it tracks code changes on its own and does not rely on an external VCS like Git. Hence, changes made before the tool is installed are not traceable using this approach. Both Historage [28] and FinerGit [27] require processing the entire repository and all of its commits. They preprocess the project repository to create another finer-grained repository that tracks changes in methods. Grund et al. [2] reported that FinerGit [27] could not finish processing the four largest repositories in their validation data set within 15 mins and in some cases ended up running out of memory, rendering the tool unusable on large projects with an extensive history. Hence, all three of these tools do not have the ability to compute change histories on the fly, while CodeTracker can do so on an average time of under 7 seconds [1].

(4) **Lack of Support for New Java Features:** Some projects mentioned above have been developed almost 10 years ago, with APFEL (2004) being developed almost 20 years ago, with no major updates being made to their implementations. Some examples of projects in this category are Historage (2011), CHRONOS (2012), Azurite (2013), Ge et al.'s code review tool (2014) [39], RefDistiller (2014), and Tempura (2015). Since most of these were developed before or close to the release year of Java 8 (2014), projects containing newer Java

features, such as the Java Stream API, default and static methods in interfaces, functional interfaces, and lambda expressions may break the implementation of these tools. CodeTracker can process all aforementioned features, including mapping code blocks to Stream API implementations where applicable. Moreover, CodeTracker can match code blocks, such as `for` loops and `if` conditionals migrated to the Java Stream API, and thus continue tracking changes back in history even before their migration.

(5) **Lack of Git Support:** Git is the most popular Version Control System (VCS) according to a 2022 developer survey performed by Stack Overflow [16], and tools that do not support tracking the commit history of a Git repository are at a major disadvantage. As discussed in Chapter 1, developers who utilize source code management software often require external assistance to understand the data stored in it. APFEL uses CVS instead of Git and Azurite tracks changes in its internal representation and does not rely on any VCS to extract change history information. CodeTracker reads the git repository directory to parse code changes from commits without having to *checkout* each version. This makes it applicable in all code bases using Git as their VCS.

(6) **Lack of GitHub User Interface Integration:** According to the 2022 Stack Overflow developer survey [16], GitHub is the most popular version control hosting platform for both personal and professional use. Having a tool integrated with the user interface of GitHub would mean that users would be able to instantly access the tool and its functionalities right away during code reviews on GitHub. CodeShovel has a standalone web application that can be used to track code change history, however, it does not directly integrate with GitHub. Tools like APFEL, CLDiff, Tempura, and RefDistiller provide a GUI built into an IDE, so a user performing a code review on a pull request would have to locally clone the repository and perform the tracking operation. CodeTracker Visualizer integrates directly with the GitHub user interface and provides code change history for any selected code element directly on the GitHub commit page, and the user's workflow is not disrupted by having to inspect information displayed on an IDE or a standalone web application.

(7) **Lack of IDE Independence:** Most tools detailed above are implemented as Eclipse plug-ins, and since the time of publishing these papers, Eclipse has steadily fallen in popularity, with only 12.57% of developers using Eclipse as their IDE according to a developer survey conducted by Stack Overflow in 2022 [16]. These tools are Azurite, APFEL, Tempura, RefDistiller, Ge et al.'s code review tool, and CHRONOS. CodeTracker is provided as a Java API and can be used in any Java application. CodeTracker also provides a REST API that can be used to build web applications. This is discussed in more detail in Chapter 5.

(8) **Lack of Multi-browser support:** RAID is the only tool in this list that presents its implementation in the form of a browser extension [41], the same as CodeTracker Visualizer. RAID provides a browser extension with Google Chrome as its only supported browser. As discussed in Section 5.5, CodeTracker Visualizer provides multi-browser support, supporting Chrome, Firefox, and Opera. Having a wide variety of supported browsers increases the flexibility provided to the user.

Chapter 3

Background

In this chapter, we shall define key terms that occur frequently throughout the thesis. Understanding the details and definitions of these terms shall aid in the thorough understanding of the ideas conveyed throughout the rest of the chapters.

(1) **Code block:** Code block, simply referred to as a block, is a lexical structure of source code that is grouped together [15]. It can also be defined as being the fundamental structure used to group multiple declarations and statements within its body. Although Java is primarily an object-oriented programming language and not a block-structured programming language, it permits the use of blocks. Code blocks are denoted by enclosing a group of statements within curly braces '{}', and they follow a sequential order of execution. For this thesis, we refer to control flow statements that use blocks as code blocks. All types of code blocks present in Java are supported for tracking by CodeTracker. These are: `for` statement, `enhanced for` statement, `while` statement, `if` statement, `do` statement, `switch` statement, `synchronized` statement, `try` statement, `catch` clause, and `finally` block. Some examples of control-flow code blocks are shown in Listing 1.

(2) **Change history:** The change history of a code element refers to the historical record of modifications made to that particular element (e.g., method, variable, attribute, block) in the source code over time. It includes details about when changes were made, who made the changes, and what specific changes were implemented. To keep track of the changes being

Listing 1 Example `for statement` and `if statement` control flow code blocks.

```java
public class Example {
    public static void main(String[] args) {
        int count = 5;

        // For loop code block
        for (int i = 0; i < count; i++) {
            System.out.println("Iteration: " + i);
        }
        // For loop code block ends

        // If-else code block
        if (count > 0) {
            System.out.println("The count is positive.");
        } else {
            System.out.println("The count is zero or negative.");
        }
        // If-else code block ends
    }
}
```

performed on the codebase as a whole, a version control system like Git or Mercurial is utilized. By examining the change history of a code element, developers can understand how the code has evolved over time, identify the reasons behind certain modifications, and track the progression of bug fixes or feature enhancements. The change history of an element is stored in the form of a JSON file to allow for easy parsing in any programming language. An example of code change history for a code block is shown in Listing 2.

(3) **Change-history oracle:** Change-history oracle, simply referred to as the oracle, is a collection of change histories of code elements grouped by the type of code element. In our change history oracle, we have a total of 3370 different change history files of code elements found throughout 20 open-source repositories. The oracle is further divided into sub-directories by code element type, referred to as the method/attribute/variable/block oracle in this thesis, and these oracles contain change histories for only the type of code element mentioned in its name. Table 3.1 contains the breakdown of the oracle code element types and the number of instances present for each type.

Table 3.1: Number of instances per code element type included in the oracle. The Block oracle is contributed in this thesis, while the other ones are contributed in [1].

Oracle Type	**Number of Instances**
Method	200
Attribute	480
Variable	1410
Block	1280
Total	3370

(4) **Version:** The term version is used in this thesis to refer to the SHA-1 Git commit ID in which the particular change occurred on the code element. It is denoted as V_e, where e is the code element upon which the change occurred. The current version refers to the commit being processed at the given point in time, and the parent version refers to the parent commit of the current version.

(5) **Statement mappings:** Statement mappings, or mappings for short, refer to the relationship or association between specific statements in the source code of one version and their corresponding counterparts in another version of the same codebase. These mappings can be utilized to analyze the modifications that occurred in the code during the transition between the two versions and are valuable for tasks like code review, impact analysis, and visualizing differences between versions. We obtain statement mappings for the tracking process from RefactoringMiner. These statement mappings help CodeTracker link code elements from one version to another and continue the tracking process with the parent version code element.

Figure 3.1 shows the statement mappings generated between two commits in the project Checkstyle. The black lines indicate a mapping between statements, and the statements inside the red rectangle are indicated as being introduced. Note that in line `L297`, the method invocation is mapped to the corresponding invocation in the current commit on line

`R302`, even though the variable name of one of the parameters changed from `version` to `checkstyleVersion`, since the statement mappings generated by RefactoringMiner are refactoring and syntax aware.

(6) **Evolution chain:** When tracking the change history of a code block from a certain commit to its introduction, we sequentially go through each commit where a change on this code element has occurred. The *evolution chain* of a code element refers to the sequence of changes that occurred in its change history over time. It represents how the code element has evolved and been modified from its initial creation to its current state. Each change in the history contributes to the code element's development and may include bug fixes, feature enhancements, code refactoring, and other modifications. By examining the evolution chain, one can trace the history of the code element and understand how it has been modified over time.

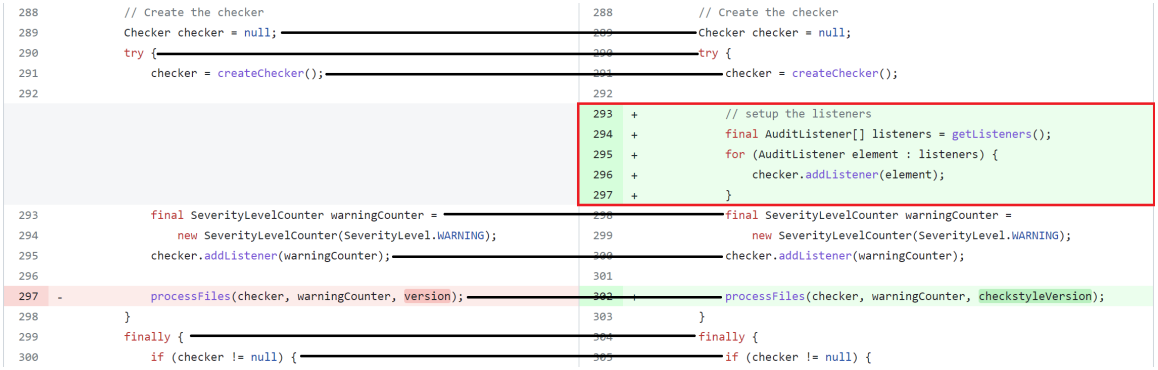


Figure 3.1: Statement mappings generated using RefactoringMiner in project Checkstyle [7].

Listing 2 Example change-history of a control flow code block from project Commons-Lang [42].


```
{
  "repositoryName": "commons-lang",
  "repositoryWebURL": "https://github.com/apache/commons-lang.git",
  "filePath": "src/main/java/org/apache/commons/lang3/LocaleUtils.java",
  "functionName": "toLocale",
  "functionKey": "src/main/java/org.apache.commons.lang3.LocaleUtils
  #toLocale(String)",
  "functionStartLine": 60,
  "blockType": "IF_STATEMENT",
  "blockKey": "src/main/java/org.apache.commons.lang3.LocaleUtils
  #toLocale(String)$if(114-116)",
  "blockStartLine": 114,
  "blockEndLine": 116,
  "startCommitId": "a36c903d4f1065bc59f5e6d2bb0f9d92a5e71d83",
  "expectedChanges": [
    {
      "parentCommitId": "a6d27fd89dc5f8c317637e539bebb3fec14caf39",
      "commitId": "15b80753a6e8f481ea5029bc278e362994cb7bee",
      "commitTime": 1460581055,
      "changeType": "body change",
      "elementFileBefore": "src/main/java/org/apache/commons/lang3
      /LocaleUtils.java",
      "elementNameBefore": "src/main/java/org.apache.commons.lang3
      .LocaleUtils#toLocale(String)$if(113-115)",
      "elementFileAfter": "src/main/java/org/apache/commons/lang3
      /LocaleUtils.java",
      "elementNameAfter": "src/main/java/org.apache.commons.lang3
      .LocaleUtils#toLocale(String)$if(113-115)"
    },
    {
      "parentCommitId": "bc255ccf5c239666ab54e5a31720d3f482ae78eb",
      "commitId": "4d46f014fb8ee44386feb5fec52509f35d0e36ea",
      "commitTime": 1357193992,
      "changeType": "introduced",
      "elementFileBefore": "src/main/java/org/apache/commons/lang3
      /LocaleUtils.java",
      "elementNameBefore": "src/main/java/org.apache.commons.lang3
      .LocaleUtils#toLocale(String)$if(106-108)",
      "elementFileAfter": "src/main/java/org/apache/commons/lang3
      /LocaleUtils.java",
      "elementNameAfter": "src/main/java/org.apache.commons.lang3
      .LocaleUtils#toLocale(String)$if(106-108)",
      "comment": "new block"
    }
  ]
}
```

Chapter 4

Research Approach

This chapter is divided into five sections - Section 4.1 lays down the foundations for the code element identifiers (specifically for code blocks) in CodeTracker, Section 4.2 deals with the block tracking enhancements made to CodeTracker, and Section 4.3 details the concept of evolution hooks. Section 4.4 highlights the methodologies adopted for the development of CodeTracker Visualizer. Finally, Section 4.5 describes the Oracle Validator fork of CodeTracker Visualizer and how it aids in oracle validation research.

4.1 Code Block Identifier

Jodavi and Tsantalis [1] defined each code element e to be uniquely identified in the commit history of a software repository with the following tuple:

$$I_e = (V_e, CON_e, SIG_e) \tag{1}$$

where V_e is the version of e corresponding to the SHA-1 Git commit-ID in which a change took place on code element e, CON_e is the signature of the container to which e belongs, and SIG_e is the signature of e. [1]

Building upon this design, the identifier for a code block b is defined as follows:

$$I_b = (V_b, CON_b, SIG_b) \tag{2}$$

Here, V_b is the version of b that corresponds to the commit ID at which a change on this block exists.

The container of a control-flow block declaration b is the tuple:

$$CON_b = (CON_{M_b}, SIG_{M_b}) \tag{3}$$

where M_b is the method declaration in which b is declared, and CON_{M_b} and SIG_{M_b} are the container and signature of M_b, respectively.

The container of a method declaration M is the tuple:

$$CON_M = (CON_{C_M}, SIG_{C_M}) \tag{4}$$

where C_M is the type declaration to which M belongs, and CON_{C_M} and SIG_{C_M} are the container and signature of C_M, respectively, as defined by Jodavi and Tsantalis [1].

The container of a type declaration C is the tuple:

$$CON_C = (SRC_C, PKG_C) \tag{5}$$

where SRC_C is the source folder path and PKG_C is the package name to which C belongs.

Now, looking at the signature of control-flow blocks:

The signature of a control-flow block declaration b, with a $body$, is the tuple:

$$SIG_b = (T_b, SIG_{p_b}, SIG_{body}) \tag{6}$$

where T_b, is the block type (e.g., `for`, `if`, `try`, `switch`), SIG_{p_b} is the signature of b's parent statement, and SIG_{body} is the signature of b's body, which is the hash value of the code inside b's body.

The signature of the parent statement p_b has a recursive definition as shown in the tuple:

$$SIG_{p_b} = (SIG_{p\prime_p}, T_p, I_{p_b}) \tag{7}$$

where $p\prime$ is the parent of p, T_p is the statement type of p, and I_{p_b} is the index of b in p's list of statements, respectively.

This information is necessary to create a unique identifier for each code block, as there may exist multiple blocks within a method that are textually identical, but have a different location in the method's control and execution flow structure.

4.2 Block Tracking in CodeTracker

CodeTracker integrates an extended version of RefactoringMiner to obtain refactoring information and detect changes taking place on code elements between commits in a refactoring-aware manner. It uses custom heuristics to improve performance alongside employing techniques like partial analysis when applicable [1].

As illustrated in Chapter 1, having block-level granularity for change-history generation is quite desirable. To achieve block-level granularity in CodeTracker, we introduced to the codebase a new `Block` type to support the processing of blocks, along with the development of `BlockTracker`, which is an extension to the multi-tracker builder pattern approach followed by CodeTracker.

Input: `BlockTracker` takes as input the following information: Git repository URL, a starting commit SHA-1 ID (or HEAD by default), the file path containing the code block of interest, the type of the code block (e.g., `for`, `if`, `while`, `try`, `catch`), and the start line number of the code block's declaration.

Output: The output is a graph, where the nodes represent code elements with their unique identifiers, and the list of changes between two nodes is attached to the edge connecting them. The change history is returned in the form of a graph due to the possibility of forks. A fork occurs when two or more different blocks are merged into one. For example, two or more `catch` blocks could be merged into a single `catch` block using the union type feature of Java for the handled exception types (e.g., `catch(ClassNotFoundException | IllegalAccessException ex)`). Another example is the extraction of two or more duplicated code blocks from the same or different methods into a single commonly used method (i.e., EXTRACT METHOD refactoring). The change history of a code block starts from the commit provided as input and goes all the way back to its

introduction commit. Therefore, by traversing the graph from the start node, we are able to visualize the changes that took place in each commit, and since the graph can contain forks, every block that was potentially merged with the current block can also be tracked to its introduction commit. Another major reason to implement a graph-style representation is to facilitate the implementation of *evolution hooks*, a mechanism that allows users to continue the tracking process in case the method containing the tracked code block gets extracted or inlined at some point in the commit history. More details about evolution hooks are given in Section 4.3.

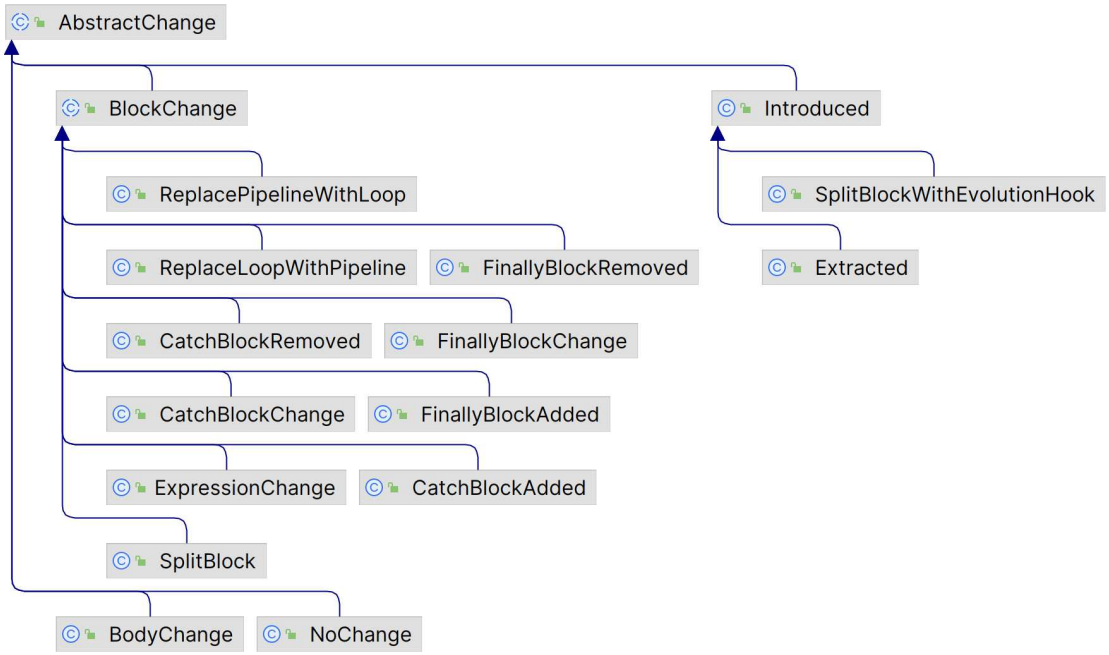


Figure 4.1: Hierarchy of supported change kinds for code blocks.

Figure 4.1 shows the change type hierarchy supported by CodeTracker for code blocks. A BLOCK SPLIT change is used to denote changes where a certain block is split into two or more blocks but maintains similar functionality as the parent commit block. BODY CHANGE and EXPRESSION CHANGE indicate a change in the body and expression of the block, respectively. We have found that BODY CHANGE is the most common type of change performed on blocks, followed by EXPRESSION CHANGE. The REPLACE PIPELINE WITH LOOP change is used to capture instances where a loop with nested conditional logic is replaced with the Java Stream API. This is a fairly common practice as developers tend to migrate their codebase to new language features [43, 44].

The reverse is also detected and reported as a valid change (i.e., REPLACE LOOP WITH PIPELINE). The addition and removal of a `catch` or `finally` block in `try` statements are captured by the CATCH BLOCK ADDED/FINALLY BLOCK ADDED and CATCH BLOCK REMOVED/FINALLY BLOCK REMOVED changes respectively. This case is also fairly common as developers tend to add and remove `catch` and `finally` blocks to handle errors as the `try` statement block evolves with new code. Finally, any changes made inside the body of a `catch` or `finally` block are indicated with the CATCH BLOCK CHANGE and FINALLY BLOCK CHANGE, respectively.

One interesting feature to point out is that the block tracking process also supports the transformation of blocks from one type to another. For example, in one case that we found, a `switch` statement was used to replace a rather cumbersome `if-else-if` ladder and then add a few extra cases [45]. We support continuous tracking in such instances, as the `switch` cases are mapped to the corresponding `if` conditionals, and the evolution chain continues.

The complete list of such transformations includes (the inverse transformation of all these cases is also supported):

(1) `if-else-if` to `switch`

(2) `if` statement to `while` loop (a developer wants to repeat the body until resolved)

(3) `for` loop to `while` loop

(4) `for` loop to `forEach` pipeline

(5) `for` loop to `if` block

(6) `try` block to `synchronized` block

(7) `catch` block to `finally` block

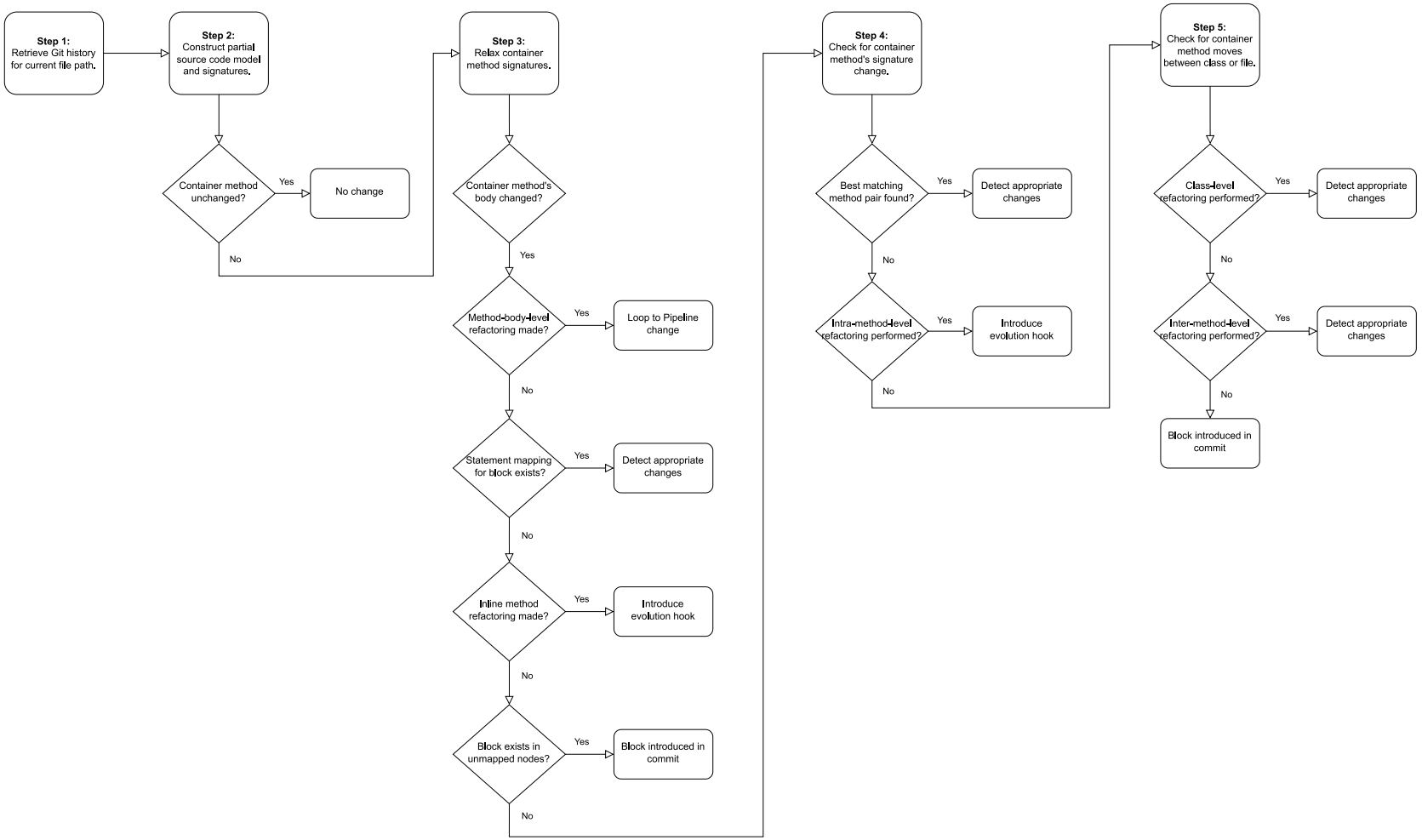


Figure 4.2: Steps 2-5 are run for each commit in the list of commits generated in Step 1, stopping execution of the remaining steps when a step identifies the next code block.

The tracking process for blocks (illustrated in figure 4.2), as implemented in `BlockTracker` is detailed below:

STEP 1: Retrieve Git history for the current file path

As a starting point, we identify the code block that is of interest (denoted by b) and obtain the file path containing the block as input from the user. We then retrieve the git history of the given repository and obtain all the commits in which the file had undergone a change. We collate these commits and move them onto the next step, ignoring the rest of the commits that had no change on the parent file. This step cuts down the need to iteratively process each and every commit in the repository. The command used for this process is `git log --follow filePath`, and by using the `follow` flag, we obtain the commit history in case `filePath` is moved or renamed as well.

STEP 2: Check if the container method is unchanged

After obtaining the set of commits where a change may have taken place on the concerned block, we iterate through each of these commits and construct a partial source code model for the file containing the block in the current commit r and the same file in the parent commit p, respectively. On the first commit we process, we obtain the code block b and its container method M in the commit r using the block type and line number. Alongside the partial source code model, we also construct the signatures (SIG_{M_r}, SIG_{b_r}) and containers (CON_{M_r}, CON_{b_r}) of the method containing the block and the block itself, as illustrated in Section 4.1. We then look into p's model and search for a method with the same signature as SIG_{M_r}. If we do find a match, it would mean that the container method had remained unchanged, and thus so had the code block contained inside its body. We then map the type declaration containers (CON_{M_p}, CON_{M_r}), method declarations (M_p, M_r) and blocks (b_p, b_r) to each other and continue the evolution chain. If a match is not found, we move on to **STEP 3**.

STEP 3: Check if the container method's body changed.

Reaching this step would indicate that the container method M_r has undergone some change in its parent commit p. Therefore, we check to see if the method remains in the same file in the parent commit and has only undergone a change in its body, that caused a change in its identifier. We do this by relaxing the method identifier when searching for a match. More specifically, by omitting

the method body's string representation hash value, SIG_{body}, we can now compare to see if there is a method that matches this identifier in commit p. If we do find a match, this would indicate that the body of the method has changed. In turn, the block contained inside the body of this method could potentially have also changed.

At this point, we execute RefactoringMiner on the partial source code models for commits r and p constructed in **STEP 2**. We then check to see if the block b_r is involved in any *method-body-level* refactorings, such as REPLACE LOOP WITH PIPELINE, REPLACE PIPELINE WITH LOOP, SPLIT CONDITIONAL, MERGE CONDITIONAL. If that is the case, we report the appropriate refactoring as a change on the block and continue the evolution chain.

In the case that none of these refactorings were performed involving b_r, we obtain the statement mappings returned by RefactoringMiner and check if b_r has been mapped to a statement b_p in the parent commit p. Upon finding a match, we construct the unique identifiers of b_r and b_p and link the two code element nodes in the graph. We then check if the contents of the block have remained the same. If the expression and/or body have changed, we report an EXPRESSION CHANGE and BODY CHANGE, respectively. If the block is a `try` block, we separate the `catch` blocks and perform a similar string representation construction and equality check. We also make note of `catch` blocks that do not have a mapping in the parent/child commit and report a CATCH BLOCK ADDED/REMOVED CHANGE, respectively. This approach is also adopted for `finally` blocks that may be present within the `try` statement.

If we don't find a match for b_r in the above process, we can suspect that b_r may have been introduced in method M_r. There are two possible scenarios. Either b_r corresponds to new functionality added in method M_r, or b_r has been moved to M_r by inlining a method originally called by M_p. To verify the latter scenario, we check if RefactoringMiner reported an INLINE METHOD with M_r as the target, and b_r has been matched with a block b_p from the inlined method. In that case, we introduce an evolution hook, so that the user can attach on demand the evolution sub-graph of b_p starting from commit p. To verify the first scenario, we check through RefactoringMiner's list of unmapped blocks present in M_r and then check if any of those correspond to b_r. If that is the case, we can safely say that the block had been *Introduced* in commit r as part of the newly added functionality of a bug fix. If b_r is not found within this list, we need to move on to **STEP**

4, where we explore the possibility of b_r belonging to a method whose signature (i.e., parameter type list, method name, return type) changed, or method M_r is introduced as the outcome of a local *intra-method-level* refactoring, such as EXTRACT METHOD, MERGE METHOD, SPLIT METHOD.

STEP 4: Check if the container method's signature changed

At this point, we utilize the information extracted by RefactoringMiner in **STEP 3** after its execution on the partial source code models for commits r and p. RefactoringMiner initially matches the method pairs within type declaration containers (CON_{M_p}, CON_{M_r}) with identical signatures (i.e., method name and parameter type list), and then compares all combinations of the remaining unmatched methods from CON_{M_r} with the remaining unmatched methods from CON_{M_p} to find the best matching method pairs with changes in their signatures. If M_r is found in a matching method pair (M_p, M_r) with method signature changes, we obtain the statement mappings returned by RefactoringMiner and check if b_r has been mapped to a statement b_p in the parent commit p. If indeed a statement mapping is found for b_r, we construct the unique identifiers of b_r and b_p, and link the two code element nodes in the graph. Otherwise, we utilize again the information extracted by RefactoringMiner to examine if any of the remaining unmatched methods from CON_{M_r} has been extracted from a pair of matched method pairs and finally check if any subset of the remaining unmatched methods from CON_{M_p} have been merged to a single remaining unmatched method from CON_{M_r} (i.e., MERGE METHOD refactoring), as well as the reverse scenario (i.e., SPLIT METHOD refactoring). If M_r is found being involved in any of the aforementioned refactoring scenarios, we obtain again the statement mappings included in the corresponding refactoring and check if b_r has been mapped to a statement b_p in the parent commit p. If a statement mapping is found for b_r, we construct the unique identifiers of b_r and b_p, link the two code element nodes in the graph, and introduce an evolution hook, so that the user can attach on demand the evolution sub-graph of b_p starting from commit p, as M_r is essentially a newly introduced method in CON_{M_r} type declaration container.

If no matches are found in this step, then we move on to **STEP 5**, where we check if b_p is located in a file other than `filePath` in which b_r is located, since there is a possibility that the container method M_r has been moved to `filePath` from another file, or the type declaration CON_{M_r} containing M_r has been renamed or moved to another package.

STEP 5: Check if the container method was moved to another class or file

This step is the most computationally expensive step of the tracking process, as we keep the partial source code model for commit r as is, but add all modified and removed files in commit p to p's source code model (i.e., we create the complete source code model for commit p) to enable the detection of *inter-method-level* refactorings, such as PULL UP METHOD, PUSH DOWN METHOD, MOVE METHOD, as well as *class-level* refactorings, such as MOVE CLASS, RENAME CLASS, EXTRACT CLASS, MERGE CLASS, SPLIT CLASS. To avoid the unnecessary processing of files and speed-up the tracking process, we exclude from p's source code model all files with identical contents, and files with only trivial changes in comments (e.g., license headers) and import declarations. Moreover, we support two scenarios in which additional files need to be included in r's source code model to correctly track b_r, which are explained in detail in [1]:

(1) b_r is copied into a new file: In this scenario, developers copy methods they want to deprecate into a new file, and then declare the original methods or their container class as `@deprecated`.

(2) b_r is extracted to a new file: In this scenario, developers move some members of an existing class into a new class, and instantiate the new class into the origin class in order to access the moved functionality (i.e., EXTRACT CLASS refactoring), or extend the origin class in order to inherit the non-moved functionality (i.e., EXTRACT SUBCLASS refactoring).

After setting up the partial source code models for commits r and p, we execute RefactoringMiner again. First, we check all class-level refactorings (e.g., MOVE CLASS, RENAME CLASS) to find a pair of type declarations (CON_{M_p}, CON_{M_r}) involving CON_{M_r}. If such a pair is found, we obtain the corresponding class-level diff object from RefactoringMiner, which includes all pairs of matched methods. We then check if M_r is included in the matching method pairs. If so, we obtain the statement mappings returned by RefactoringMiner for the (M_p, M_r) method pair, and check if b_r has been mapped to a statement b_p in the parent commit p. If indeed a statement mapping is found for b_r, we construct the unique identifiers of b_r and b_p, and link the two code element nodes in the graph.

If there is still no match found for b_r, this is an indication that either b_r itself or its method

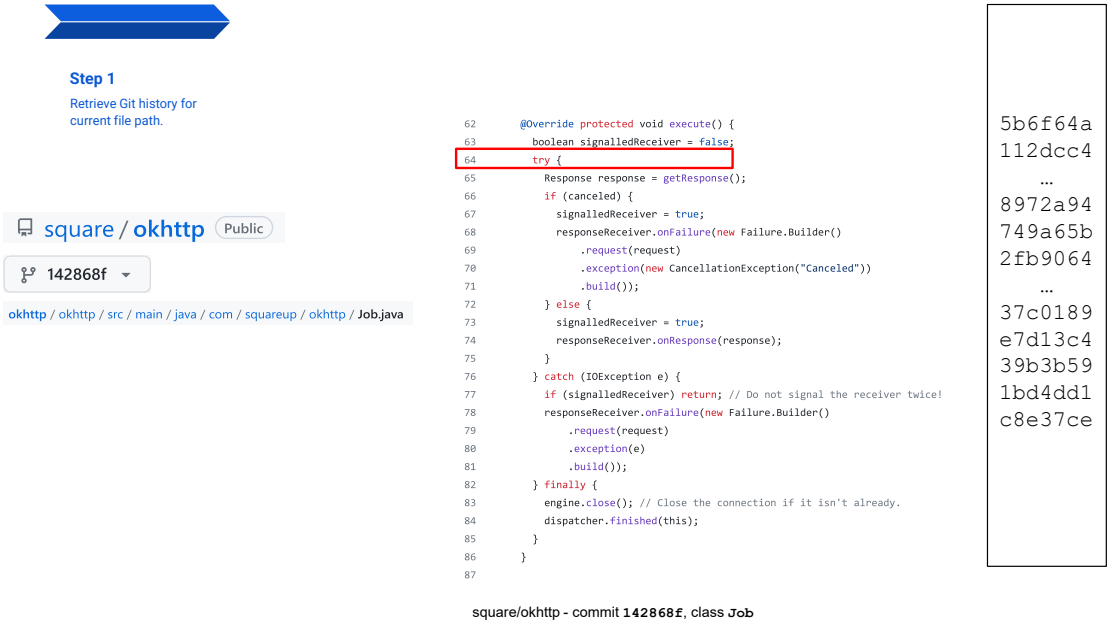


Figure 4.3: Running example: Running step 1 on the block to generate a list of commits shown on the right.

container M_r has been moved to another file through an EXTRACT AND MOVE METHOD or MOVE METHOD refactoring, respectively. Finally, we check in all *inter-method-level* refactorings reported by RefactoringMiner if method container M_r is involved. If so, we obtain the statement mappings included in the corresponding refactoring and check if b_r has been mapped to a statement b_p in the parent commit p. If indeed a statement mapping is found for b_r, we construct the unique identifiers of b_r and b_p, and link the two code element nodes in the graph.

If by the end of **STEP 5** there is still no match found for b_r, we report that b_r has been *Introduced* in commit r as part of a newly added method.

Steps 2-5 are iteratively executed until the tracked block is found as *Introduced*, or we reach the first commit of the project, which means that the tracked block has existed since the beginning of the project.

We shall now run these steps on an example block to understand the flow of execution (Figure 4.3 to Figure 4.13).

Figure 4.4: Running example (commit 1/10): Running step 2 on the block to check if the container method has remained unchanged.

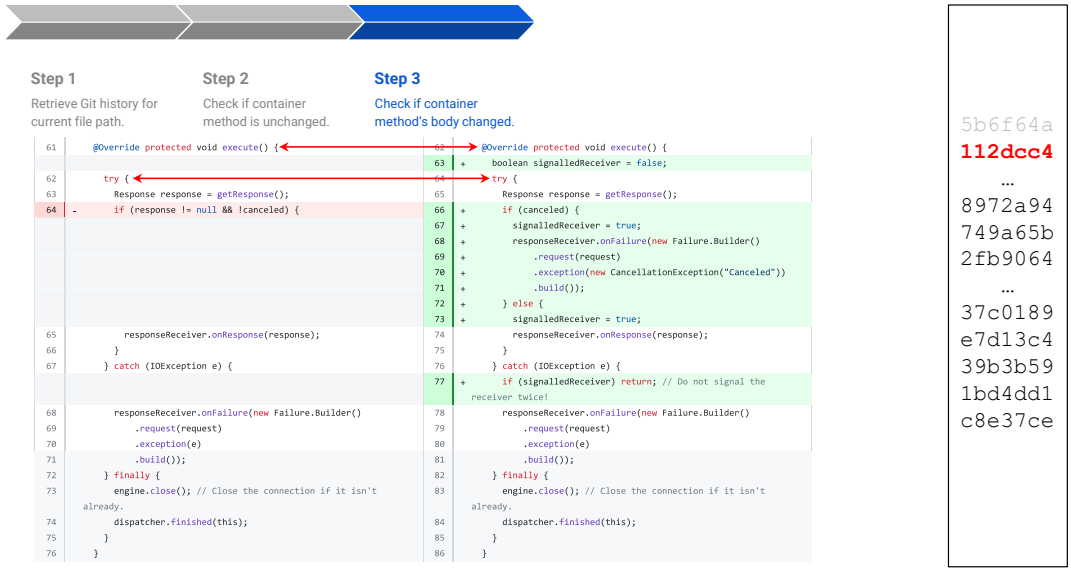


Figure 4.5: Running example (commit 2/10): Running step 3 on the block to check if the container method's body has changed.

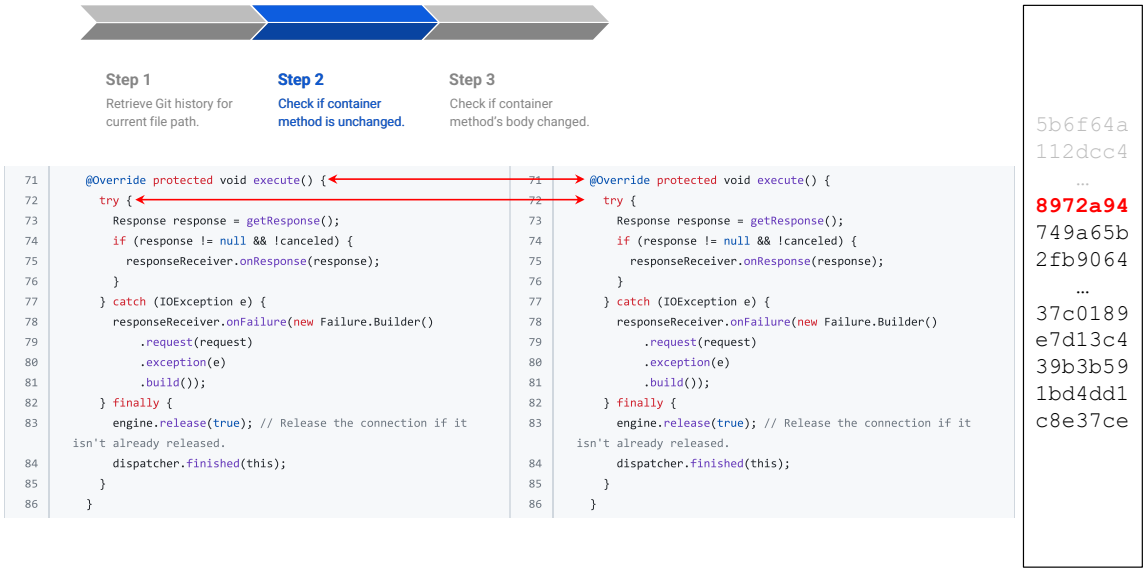


Figure 4.6: Running example (commit 3/10): Running step 2 for this commit is sufficient as the container method has not changed (steps 3-5 are skipped.)

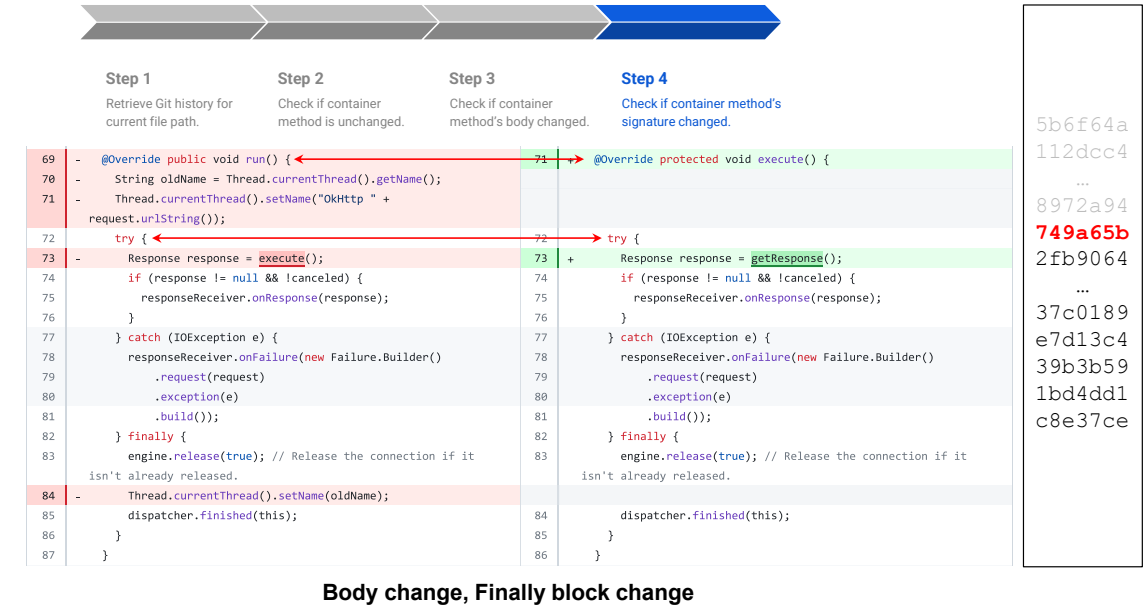


Figure 4.7: Running example (commit 4/10): Running step 4 on the block to check if the container method's signature has changed, which in this case it has.

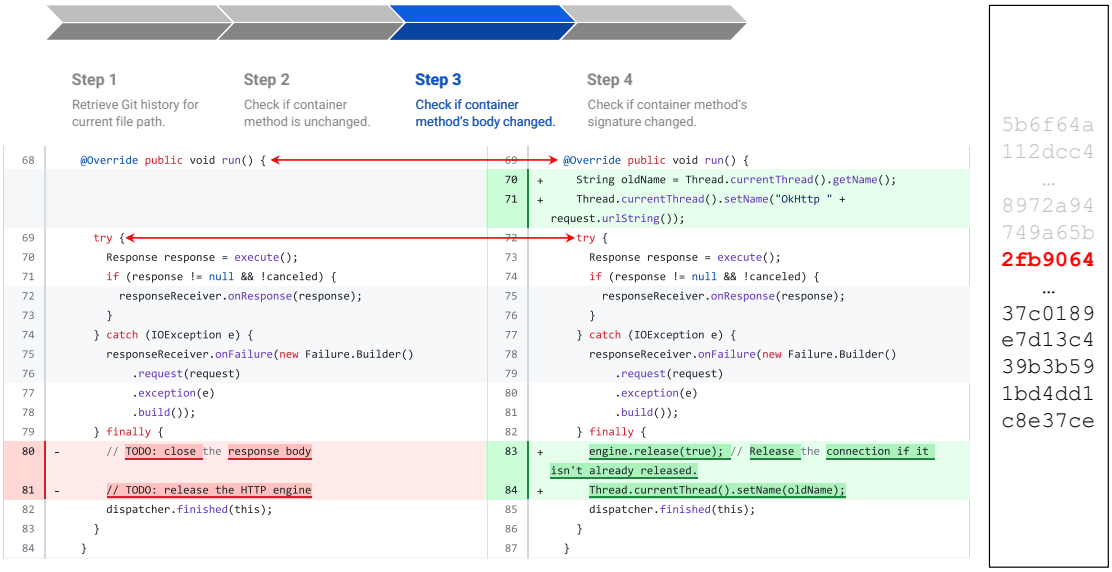


Figure 4.8: Running example (commit 5/10): Running step 3 on the block to check if the container method's body has changed.

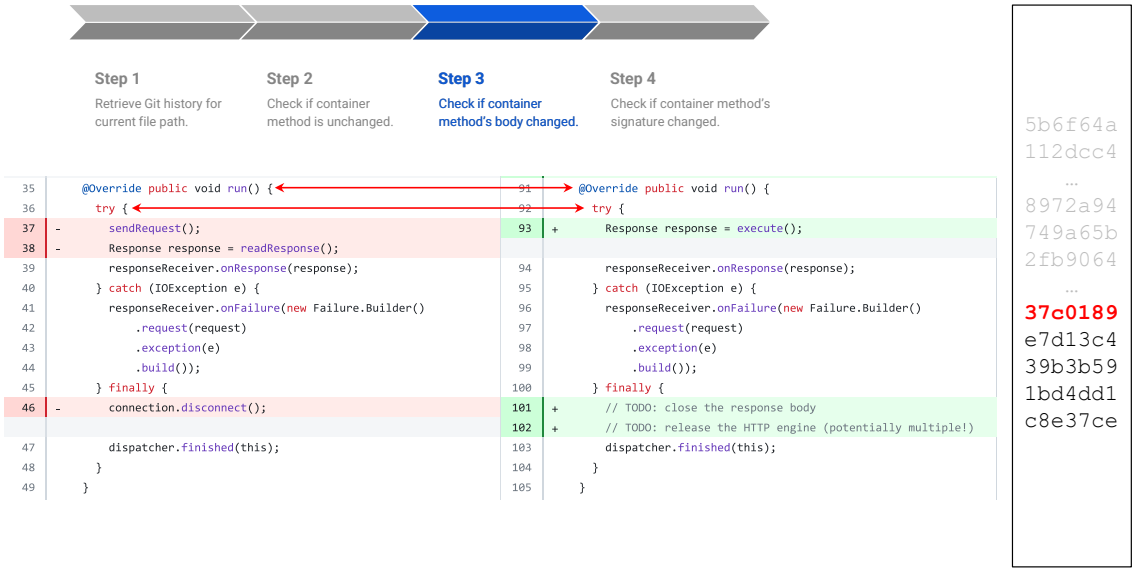


Figure 4.9: Running example (commit 6/10): Running step 3 is sufficient for this commit.

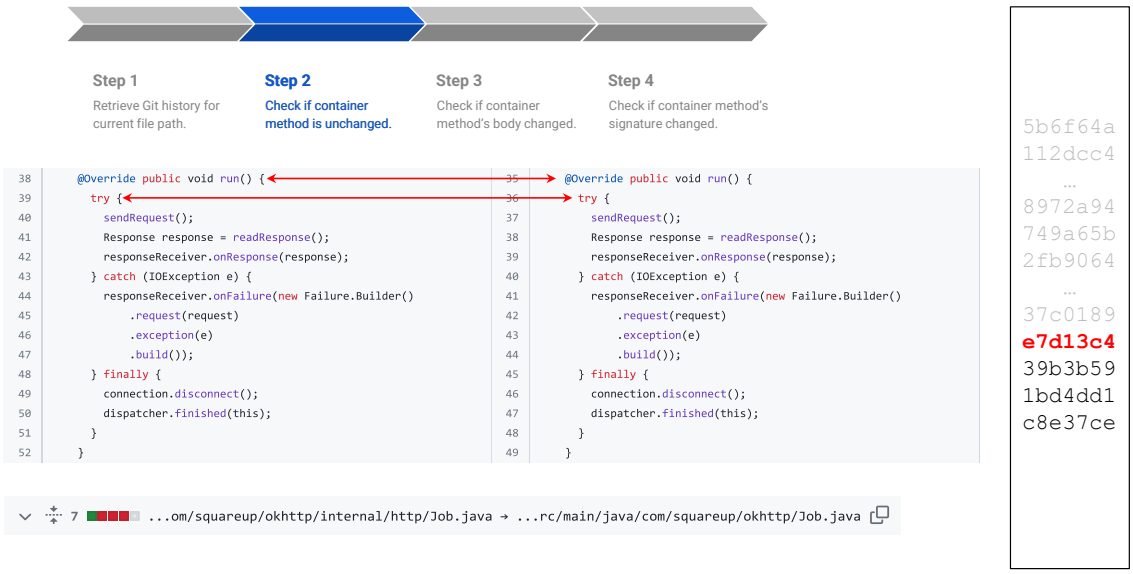


Figure 4.10: Running example (commit 7/10): Running step 2 is sufficient as the container method has not undergone any changes.

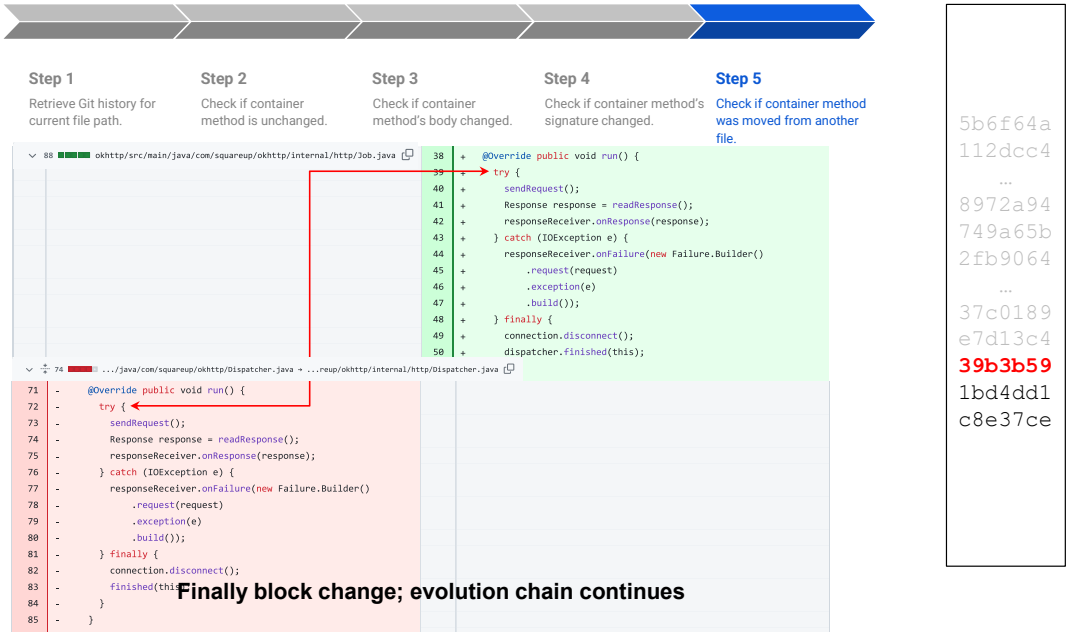


Figure 4.11: Running example (commit 8/10): Running step 5 on the block to detect container method file moves.

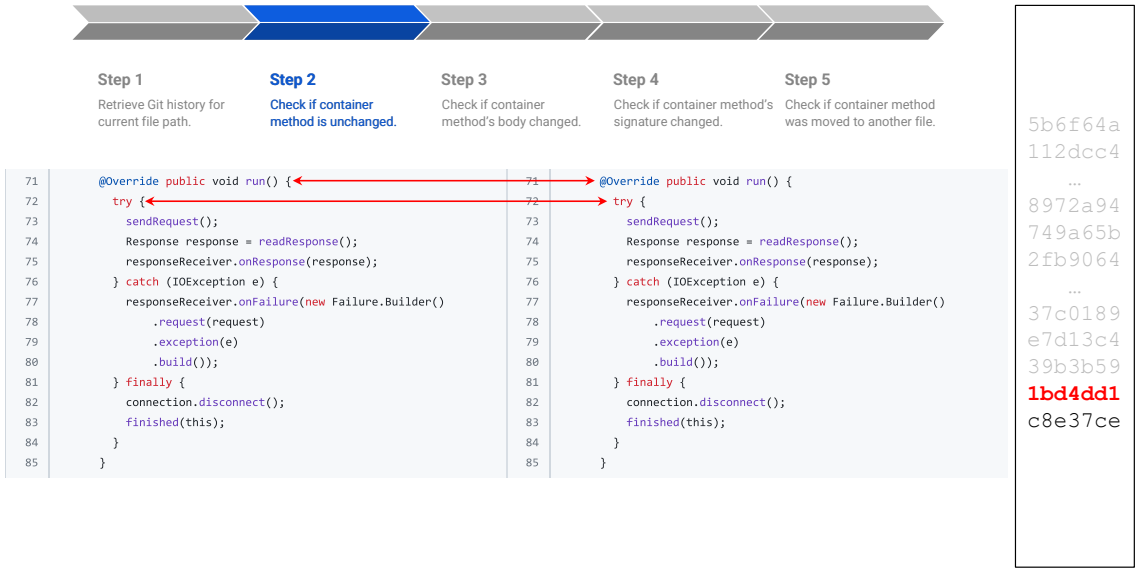


Figure 4.12: Running example (commit 9/10): Running step 2 on the block is sufficient.

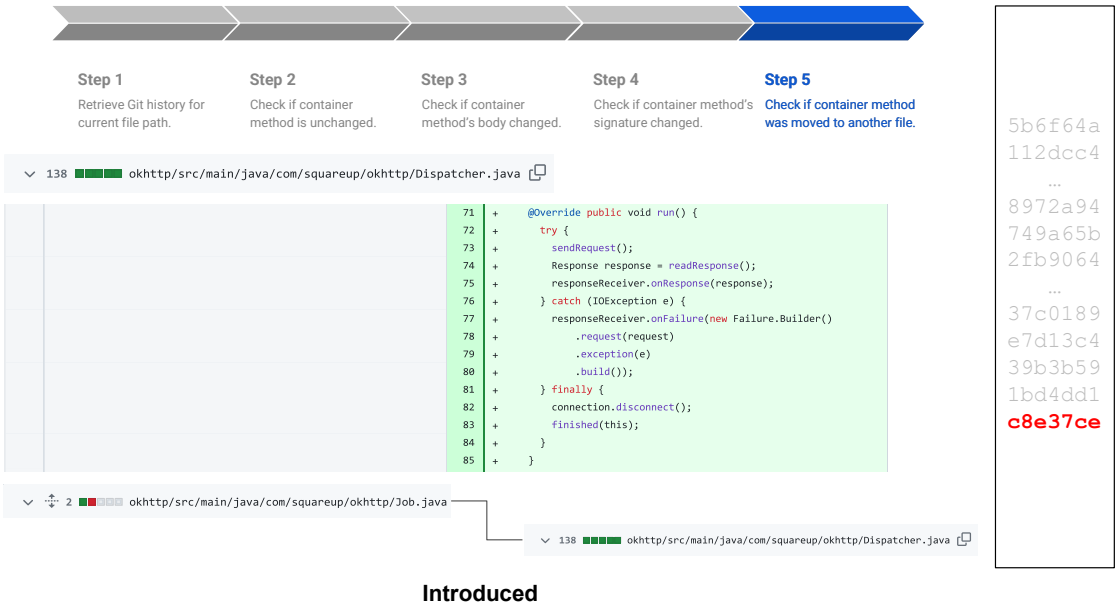


Figure 4.13: Running example (commit 10/10): Running step 5 to check for container method file moves. In this case, the block was not moved and was introduced, concluding the tracking process.

4.3 Change Graph Evolution Hooks

As described in Section 4.2 and introduced by Jodavi and Tsantalis [1], *evolution hooks* are a mechanism that allows users to continue tracking code elements on-demand when the tracked code element is found located in a newly introduced container (e.g., method) after the application of a local *intra-method-level* refactoring, such as EXTRACT METHOD, INLINE METHOD, MERGE METHOD, SPLIT METHOD. This design allows us to avoid computing additional change history which might not desired by the user, but at the same time inform the user about the opportunity to further explore the change history, if and when desired.

With respect to code blocks, when we come across an evolution hook, we automatically continue the tracking process. We did so since a code block can be considered as an independent code element, which can be moved between methods in the same file or even different files. Moreover, this choice allowed us to have a comprehensive oracle of code block changes with advanced evolution scenarios, where code blocks are moved to newly introduced containers (i.e., methods or classes).

CodeTracker Visualizer utilizes the ability to perform on-demand change history generation using evolution hooks for code elements other than code blocks (e.g., method declarations) and is described in detail in Section 4.4.

4.4 CodeTracker Visualizer Approach

Code review is a crucial aspect of contemporary software development. Presently, this practice involves utilizing textual code differencing tools, like the one found on GitHub, to review code changes [5]. Textual diffs are prone to quickly becoming convoluted and difficult to comprehend for more complex changes. When using the textual code difference view provided on GitHub, refactorings are not explicitly represented on the commit page. As a consequence, understanding the changes being made between versions becomes more intricate since reviewers must deduce on their own that a specific group of added and removed lines of code could signify a potential refactoring [41].

The vision for CodeTracker Visualizer was to build a tool that could provide easy access to code

change history and assist software developers during code reviews by helping provide near-instant historical information and navigation for just about any code element in the commit. CodeTracker provides its functionality as a rich Java API [1] which is excellent for developing software that integrates with CodeTracker, however, everyday usage with an API is both tedious and unrealistic. Having to gather information about each code element that one would like to track and manually passing it onto the API was the only option users had, especially when browsing projects on a cloud-based version control application, such as GitHub. CodeTracker Visualizer aims to bridge this gap and provide an intuitive out-of-the-box solution to using CodeTracker straight from within the GitHub user interface. Ease of use was key during the development of CodeTracker Visualizer and remains to be one of its strongest assets.

Since integration with GitHub was a primary goal to achieve, the first step we took was to explore the capabilities of GitHub Apps, an in-house solution supported by GitHub. GitHub Apps directly interact with GitHub as a user would, and can provide deep integrations with the GitHub API. However, we quickly realized that a major limitation that comes with this approach is the inability to interact with the browser DOM (Document-Object-Model), which meant we could not provide the user with a fully automated experience and would have the user jumping through hoops by clicking links. Another potential approach we explored was to utilize the WebHooks infrastructure that GitHub provides. This approach would allow us to set up listeners that can await commits and pull requests made onto a repository and then allow us to process these commits with CodeTracker, however, this was not exactly what we wanted to achieve from CodeTracker Visualizer. We needed a solution that could provide efficient and on-the-fly code change history without having the need to install GitHub apps or set up webhooks. We also built a React web application that would communicate with the GitHub REST API and CodeTracker using a Java web server to display the code change history of a selected code element, and this application was linked to the user via a comment left by a GitHub App bot. This solution had access to the DOM and the code change history was computed on the fly, however, it needed the user to leave the GitHub web application and go onto a completely different web app, and this was not the seamless experience we were looking for.

To eliminate the need for a bot leaving comments with links throughout commits, we replaced

the GitHub app with a browser extension since all the app was doing was redirecting users and a browser extension could just as easily do that while not polluting the repository with comments. This route ended up being pivotal to the design of CodeTracker Visualizer, as we know it today. With the implementation of the browser extension, we realized that instead of having it redirect the user to another app, we could modify the GitHub app to integrate the functionalities we wanted. This ended up being the final path we chose for CodeTracker Visualizer as it met all the goals we had laid out - easy to set up and use (no configuration required), integrates directly with the GitHub user interface, and can control the DOM to allow for task automation.

Below, we shall look into what CodeTracker Visualizer is at its core, and discuss the solutions we implemented to achieve a fully automated code change history navigator.

Figure 4.14: CodeTracker Visualizer browser extension with visible history nodes.

At its current state, CodeTracker Visualizer is a browser extension (Figure 4.14) that integrates with the GitHub UI to provide a visual overlay with code change history for any code element present on GitHub. To obtain code change history from CodeTracker Visualizer, a user only needs to double-click the desired code element on the GitHub web app, whether it may be on a commit or file blob page. The extension also allows for the tracking of code elements from private repositories, with the input of a valid API key in the settings pane, as shown in Figure 4.15.

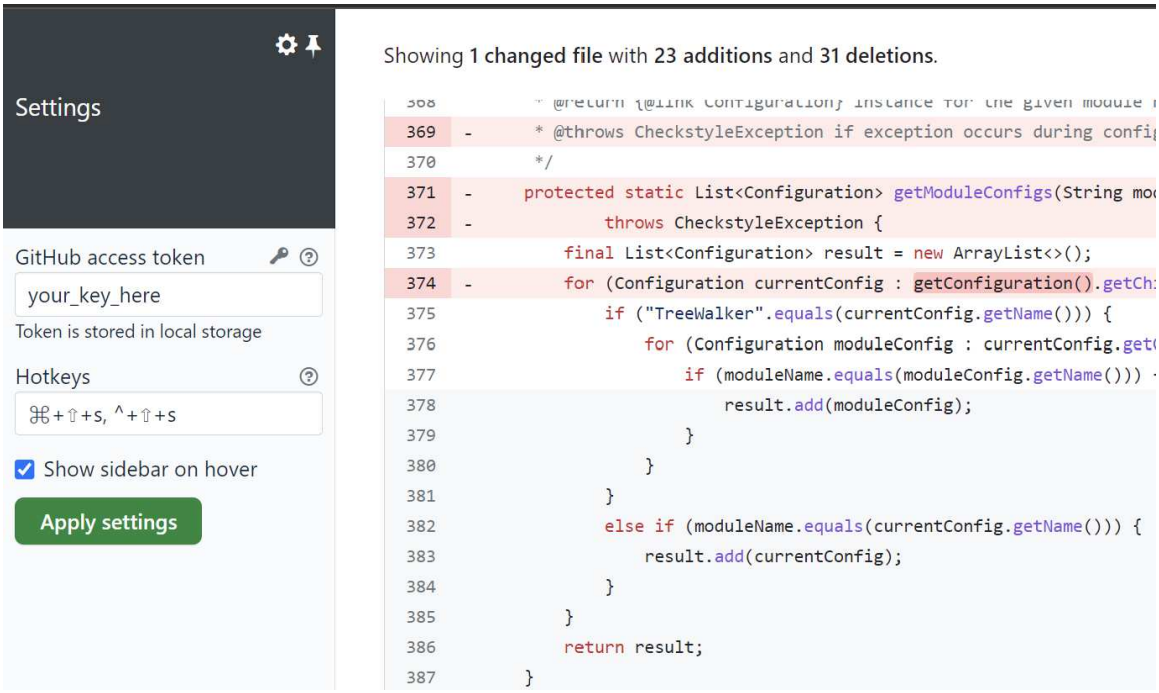


Figure 4.15: Settings panel for CodeTracker Visualizer.

When a user selects a code element, we capture the mouse event and obtain the text selected by the user. This selection is the name of a method, attribute, variable, or code block.

By accessing the DOM, we then pick up the line at which the code element is present and capture the line number. After this, we move up in the DOM until we reach the file container of the line, which contains the file path of the class containing the selected code element. Finally, we capture the commit from the webpage URL, along with the repository name. As seen in Figure 4.16, all this information is then passed to CodeTracker's REST API, which is run on a Java Web Server and can serve CodeTracker's functionalities over the web. Among the various endpoints available, which are covered in detail in section 5.4, we have an endpoint `GET codeElementType`, which takes in the information provided above and returns the type of code element being selected. The type of code element selected is then shown to the user on the side panel along with the name of the code element, as shown in Figure 4.17. This helps provide instant feedback to the user with just a click. When the user makes an invalid selection, for example, an incomplete method name, a keyword, an operator, or multiple code elements at once, we gray out the track button using information obtained from this API endpoint.

When the user has made a valid selection (method, variable, attribute, or block) and would like

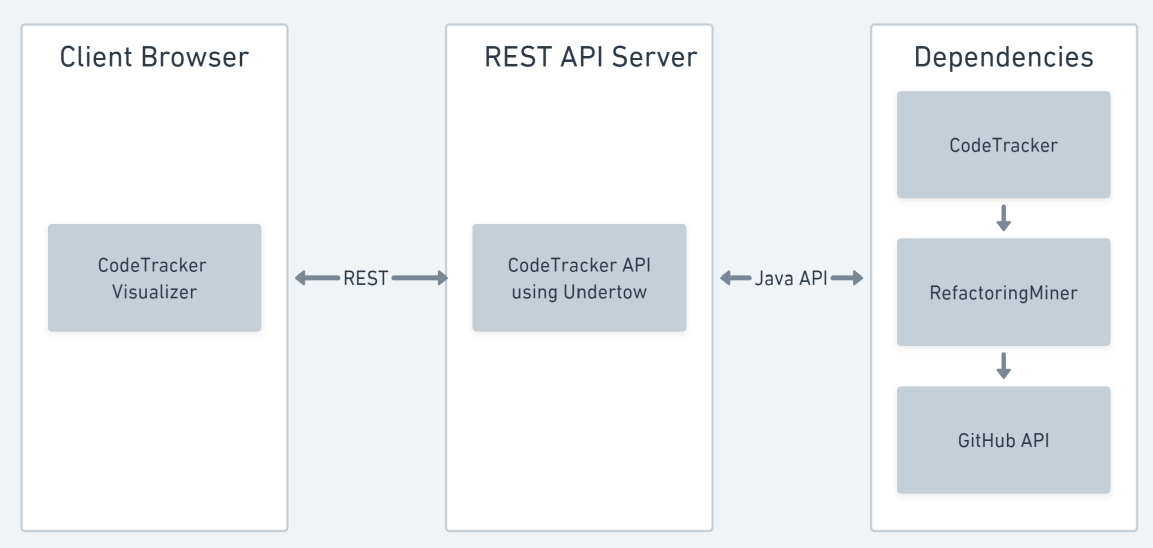


Figure 4.16: CodeTracker Visualizer architecture.

to proceed with tracking its code change history, they click the "Track" button which initiates the tracking process using CodeTracker, via the REST API. As seen in Figure 4.18, once the data is obtained from the API, we model the JSON response into a graph, which is essential to render a visual representation of the code change history evolution. The visual representation of the code change history can be seen in Figure 4.14.

A green node in the code change history indicates a change found in the history of the code element. As seen in Figure 4.20, the user can hover their cursor over a node to see details about the change and the commit that it was made in. A yellow node indicates the presence of an *evolution hook* in the change history of the tracked code element, which can be expanded and computed on the user's demand. As shown in Figure 4.19, when a yellow node is encountered, the change history stops at that point, since there is no further history on this specific code element. Right-clicking a yellow node expands the graph to reveal the change history of the code element in reference to its new container method. This concept is referred to as an *evolution hook* 4.3 and is used to compute additional code change history when required by the user.

A gray dotted border encompasses the complete history of a code element. When an evolution hook is tracked, the history is appended to the current history to present a continuous traceable list. However, this new history is encompassed in its own border, helping differentiate between the end of one code element's history and the start of another. This dotted border can be seen in Figure 4.19.

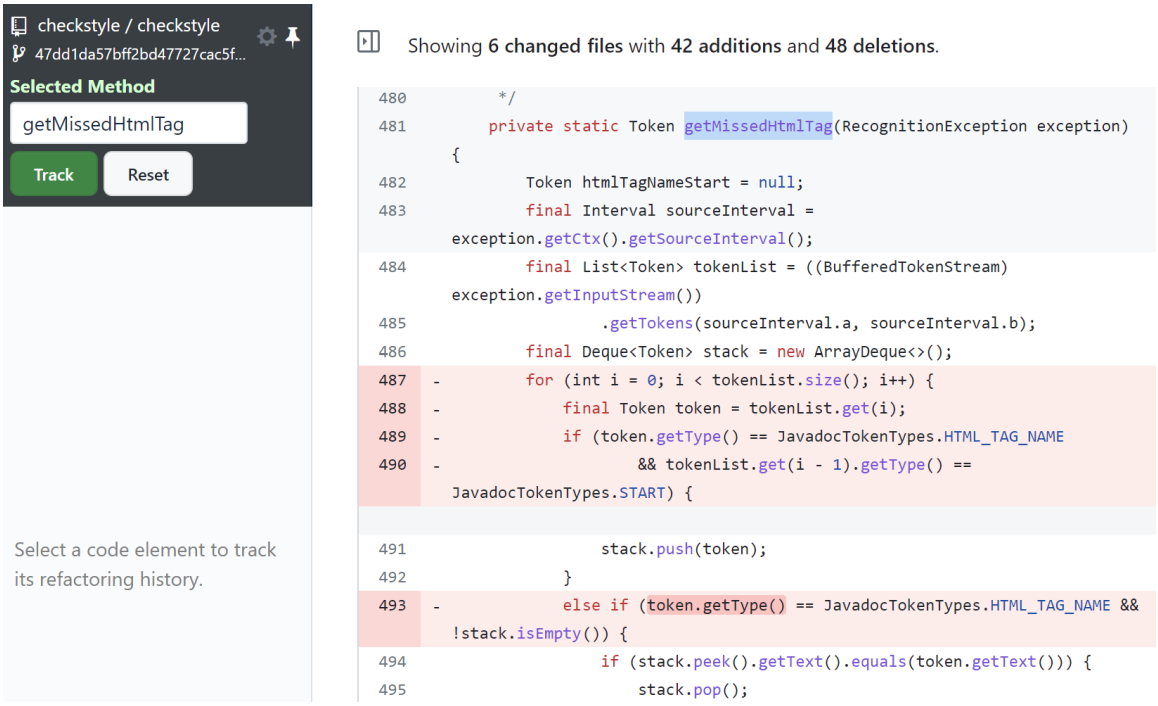


Figure 4.17: The code element type is analyzed and displayed to the user.

Looking at the navigational capabilities of CodeTracker Visualizer, the user may click on any node on the change history graph and they will be appropriately navigated to the exact line the code element is present in, in its specific commit. The complete implementation details of this navigation can be found in detail in Section 5.5, but in short, we use a variety of mechanisms to automate the browser redirects, scrolls, clicks, and network calls required in various scenarios.

As mentioned at the start, ease of use was a key element in mind during the design and development of the CodeTracker Visualizer application, and our approach emphasizes this aspect. A user can obtain the entire code change history for a code element in two clicks, and navigate to the exact code element at any change throughout its commit history with one additional click. This streamlined process aims to help increase efficiency in the usage of CodeTracker as a code change history generator.

4.5 CodeTracker Visualizer: Oracle Validator

A modified version of CodeTracker Visualizer, known as the Oracle Validator version, was developed to help validate the change history oracle that is used to evaluate the precision and recall

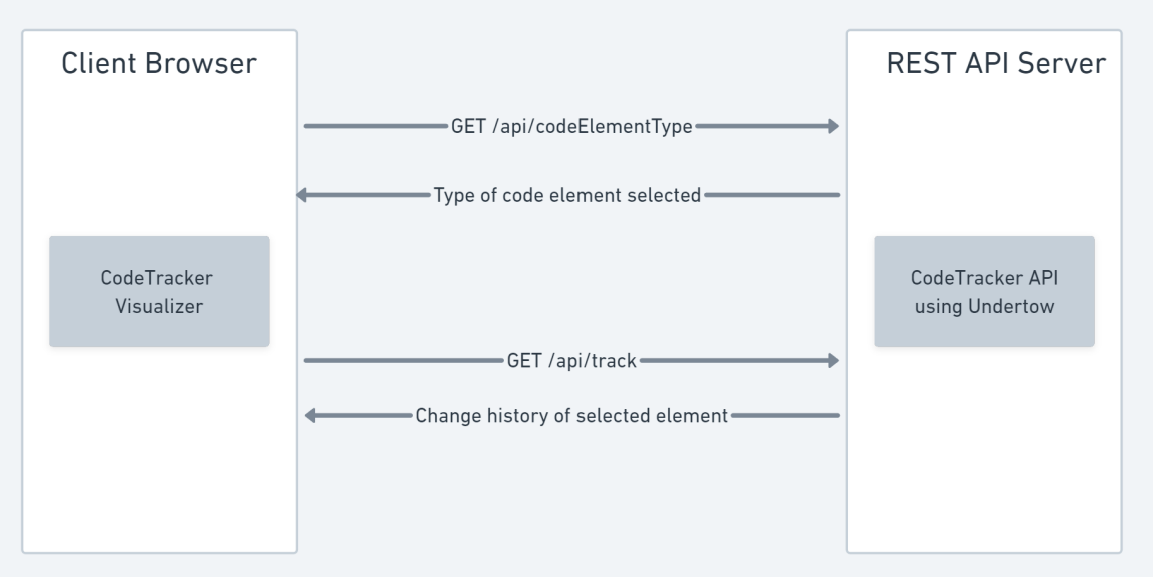


Figure 4.18: API flow of the tracking process.

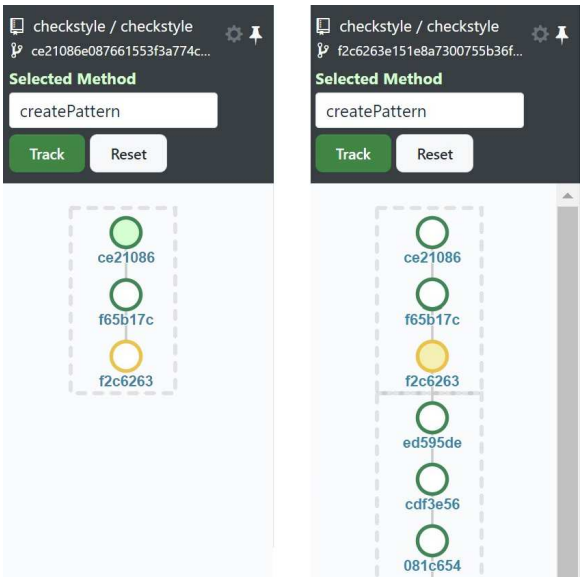


Figure 4.19: A yellow node indicates the presence of an *evolution hook*, which can be expanded on-demand by right-clicking on it.

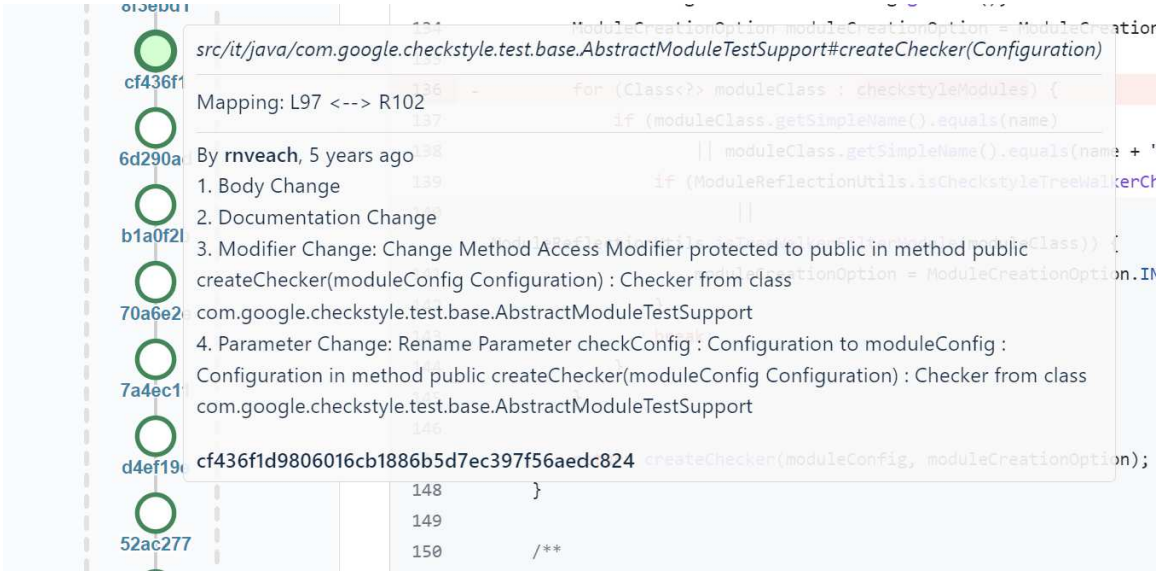


Figure 4.20: Hovering over a node provides more semantic information about the change.

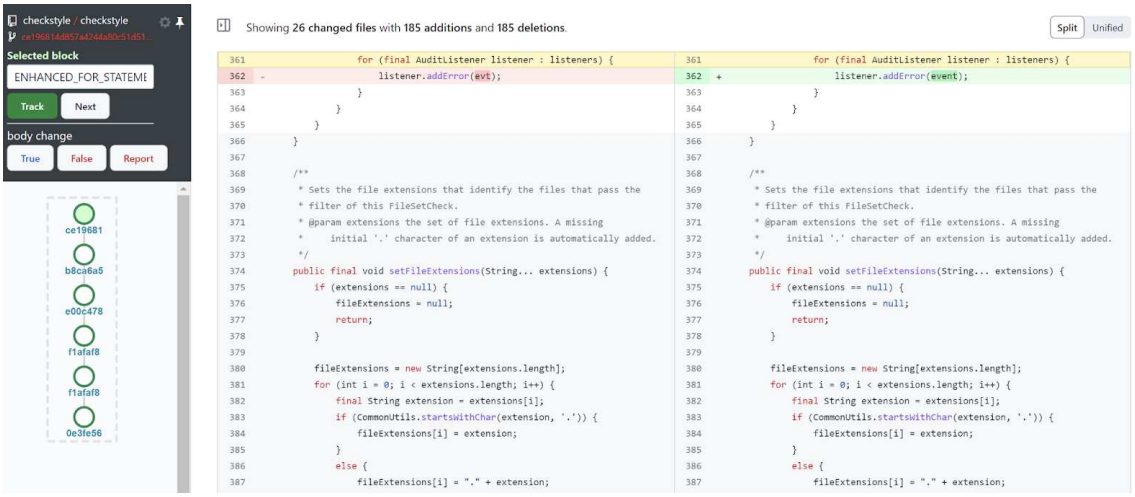


Figure 4.21: CodeTracker Visualizer's oracle validation fork.

of CodeTracker. Oracle Validator automates the navigation between oracle files and their commits and allows users to efficiently report errors discovered within them. Figure 4.21 shows a screenshot of its user interface.

This version of CodeTracker Visualizer utilizes a REST API that can serve oracle files from the file system to the application, which then utilizes its internal navigational capabilities to walk through each commit in the change history and have a human validator verify each change mentioned in the oracle. It integrates with the GitHub API to provide a quick report feature, which can automatically report an invalid change being detected by CodeTracker as a GitHub issue, with all the relevant information required to investigate the error. The issue generation feature cut down significantly on the time required to report an error in the oracle or the code mappings being returned by RefactoringMiner. When an oracle file was fully verified, it would be moved to a *valid* directory by the REST API, which indicated that this oracle was manually verified as being valid. On an error being detected, the error was reported and the file moved to an *invalid* directory. After this, the next file in the oracle would automatically be loaded, and the user would be navigated to the code element in the first commit described in this file.

This semi-automated solution cut down on the time required to validate our oracle, which in turn allowed for multiple rounds of full-oracle validations to be performed, ensuring integrity in the changes detailed by the oracle.

Chapter 5

Implementation

In this chapter, Section 5.1 provides a background of the implementation details of CodeTracker. Section 5.2 explores the implementation details of the `Block` code element type, while Section 5.3 introduces implementation details of the `BlockTracker` class, which is the tracker implementation responsible for the tracking of code blocks. Section 5.4 describes the implementation of CodeTracker's REST API. Next, Section 5.5 entails the design and architecture of CodeTracker Visualizer, followed by its implementation and test details. Finally, Section 5.6 talks about the development of the BlockTracker version integrated with GumTree [5].

5.1 CodeTracker Implementation Background

CodeTracker is written in Java and the build automation tool of choice is Maven. CodeTracker heavily depends on RefactoringMiner to gather refactoring information and statement mappings. Other libraries that it depends on are JGit and Guava. CodeTracker uses JGit to gather the commit history information of a file. Guava is used to represent the change-history generated as a graph.

CodeTracker is primarily exposed as an API for ease of use. The API is designed as a single, core API, housing multiple *Trackers* inside. The various tracker classes present are `ClassTracker`, `MethodTracker`, `VariableTracker`, and `AttributeTracker`. Each of these trackers is modeled around its respective code element class. For example, `MethodTracker` uses a `Method` class to help process methods. This thesis introduces the addition of the `Block` and

`BlockTracker` classes, utilized for tracking code blocks. The following two sections provide more information about the implementation of the `Block` and `BlockTracker` classes.

5.2 `Block` Code Element Type

The various code element classes in CodeTracker are all modeled around the `BaseCodeElement` abstract class, containing essential methods such as identifier generators and code element comparators, and attributes such as the name of the code element as well as its file path, version, and identifier.

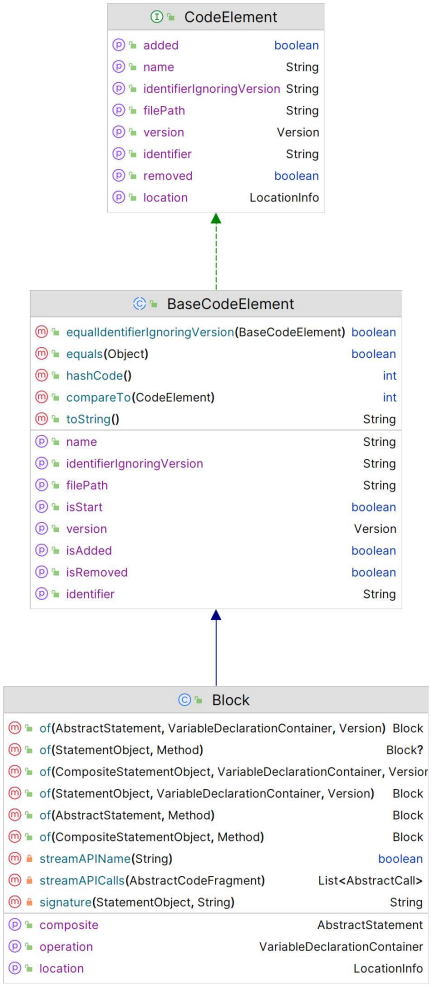


Figure 5.1: Methods and attributes available in the `Block` code element class

To implement block-level tracking, we first need to define what a block is in CodeTracker terms. This meant we had to extend the `BaseCodeElement` abstract class and implement a `Block` class

that included the methods needed to process a code block. Inside the `Block` class, we have one constructor and several static factory methods, following the factory design pattern to provide rich API usage. Figure 5.1 shows the various methods and attributes available in the `Block` class, as well as its superclasses. Below are the various static factory methods made available in the `Block` class for the purpose of instantiating a code block element taking various RefactoringMiner API objects as input parameters:

(1) `public static Block of(CompositeStatementObject composite, Method method)`

(2) `public static Block of(StatementObject statement, VariableDeclarationContainer operation, Version version)`

(3) `public static Block of(StatementObject statement, Method method)`

(4) `public static Block of(AbstractStatement statement, VariableDeclarationContainer operation, Version version)`

(5) `public static Block of(AbstractStatement statement, Method method)`

5.3 BlockTracker Implementation

Similar to the code element classes, the tracker classes are also extended upon an abstract class, `BaseTracker`. As shown in Figure 5.2, `BlockTracker` needs to be instantiated using the `CodeTracker` interface – which returns a `BlockTracker` builder. The fluent [46] builder then collects the repository information and the inputs as described in section 4.2 to identify the code block of interest. Table 5.1 shows the methods that are available in the `BlockTracker` class. Table 5.2 shows all the types of code blocks that are trackable by `BlockTracker`.

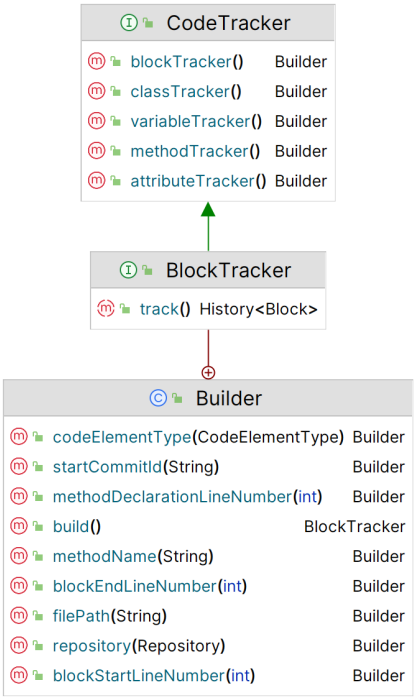


Figure 5.2: `BlockTracker` API UML diagram.

Table 5.1: `BlockTracker` API using the fluent builder pattern (`BlockTracker.Builder` class)

Return Type	Method & Description
`BlockTracker.Builder`	`repository(Repository repository)` Set repository to perform tracking in. The variable passed as the parameter should be of type Repository, created from RefactoringMiner's API GitService.
`BlockTracker.Builder`	`startCommitId(String id)` Set the commit from which to start the tracking process. The commit ID is passed as a string.
`BlockTracker.Builder`	`filePath(String filePath)` Set the file path of the file containing the block that is to be tracked. The file path is passed in as a string.

`BlockTracker.Builder`	`methodName(String methodName)` Set the method name of the container method of the block to be tracked. The method name is passed in as a string.
`BlockTracker.Builder`	`methodDeclarationLineNumber(int line)` Set the line number of the container method declaration. The line number is passed in as an integer.
`BlockTracker.Builder`	`codeElementType(CodeElementType type)` Set the type of block to be tracked, (e.g. `FOR_STATEMENT`, `IF_STATEMENT`, `TRY_STATEMENT`, etc.). It requires the type to be passed as a `CodeElementType`, which is an enum in RefactoringMiner's `LocationInfo` class.All supported block types are covered in Table 5.2.
`BlockTracker.Builder`	`blockStartLineNumber(int line)` Set the line number of the block declaration. The line number is passed in as an integer.
`BlockTracker.Builder`	`blockEndLineNumber(int line)` Set the line number of the block end line. The line number is passed in as an integer.
`BlockTracker`	`build()` Constructs a BlockTracker Class Instance with all the required arguments.

Code Snippet 3 shows how the BlockTracker API can be instantiated and used to track the history of the `TRY_STATEMENT` block starting at line 453 and ending at line 465, in method name `setupChild` declared in line 448 of a file with name `Checker.java`, and start commit `119fd4fb`

Table 5.2: Block types available in `CodeElementType` enum.

CodeElementType	**Description**
FOR_STATEMENT	Simple for loop
ENHANCED_FOR_STATEMENT	Enhanced for loop
WHILE_STATEMENT	While loop
IF_STATEMENT	If statement with or without an else block
DO_STATEMENT	Do while loop
SWITCH_STATEMENT	Switch statement
SYNCHRONIZED_STATEMENT	Synchronized statement
TRY_STATEMENT	Try block with or without resources
CATCH_CLAUSE	Catch block that follows a try block
FINALLY_BLOCK	Finally block that follows a try block

Listing 3 Instantiation of the Block Tracker

```
1  GitService gitService = new GitServiceImpl();
2  try (Repository repository =
3      gitService.cloneIfNotExists("checkstyle\\checkstyle",
4      "https://github.com/checkstyle/checkstyle.git")) {
5
6      BlockTracker blockTracker = CodeTracker.blockTracker()
7          .repository(repository)
8          .filePath("src/main/java/com/puppycrawl/tools/checkstyle/Checker.java")
9          .startCommitId("119fd4fb33bef9f5c66fc950396669af842c21a3")
10         .methodName("setupChild")
11         .methodDeclarationLineNumber(448)
12         .codeElementType(LocationInfo.CodeElementType.TRY_STATEMENT)
13         .blockStartLineNumber(453)
14         .blockEndLineNumber(465)
15         .build();
16
17     History<Block> blockHistory = blockTracker.track();
18 }
```

Calling the track operation on the `BlockTracker` class returns a generic version of the `History` interface that is parameterized over code element types, and in this case, over `Block`. This interface provides a graph of change history as described in Section 4.2.

5.4 CodeTracker REST API

CodeTracker exposes its functionality as a Java API. While this is sufficient for integrating CodeTracker into one's own Java software systems, it lacks the capabilities required for building a web application, like CodeTracker Visualizer (covered in Sections 4.4 and 4.5), which requires a web API, like a REST API in this case. CodeTracker REST API [47] enables client-side applications to communicate and integrate with CodeTracker. It is built on Undertow [48], which is a lightweight Java web server. It also utilizes FasterXML-Jackson [49] for `JSON` parsing, JGit [50] to read locally cloned repositories, and GitHub API for Java by K. Kohsuke [51] to access GitHub via its APIs. Of course, it also uses CodeTracker [1] to generate code change history.

It provides the following endpoints:

(1) **`GET /api/codeElementType`**

This endpoint is used to obtain the type of code element selected by the user given the following input: repository name, commit ID, file path, selection text, line number, GitHub username, and API token (in case of private repositories)

Output: type of code element selected (method, attribute, variable, block, invalid)

(2) **`GET /api/track`**

This endpoint is used to obtain the code change history of a code element selected by the user given the following input: repository name, commit ID, file path, selection text, line number, GitHub username, and API token (in case of private repositories)

Output: code change history of code element selected returned as a JSON.

(3) **`GET /api/getOracleData`**

This endpoint is used by the Oracle Validator version of CodeTracker Visualizer. It is used to get the next file in line for validation from the oracle. It takes no input and returns the file contents as output.

(4) **`POST /api/addToOracle`**

This endpoint is also used by the Oracle Validator version of CodeTracker Visualizer. It is used to move a file from one directory to another, depending on the manual evaluation performed by the user. For example, moving an oracle file from a directory containing *invalid* oracle files to a directory containing *valid* files when the file contains accurate change history. By flipping a parameter, we can also trigger the quick report feature discussed in section 4.5.

Input: commit ID, commit URL, Highlight ID, report (boolean), valid (boolean)

Output: Moves the file to the specified directory and/or creates an issue on GitHub.

5.5 CodeTracker Visualizer

To prevent ourselves from reinventing the wheel, we looked to utilize open-source browser extension solutions that could help cut down on development time. We decided to pick Octotree [52] for this purpose - it is an open-source browser extension that provides a pinnable tab on the left side of the screen, which was ideal according to our design for the application. Upon this, we built the user interface of CodeTracker Visualizer. Octotree internally uses *Node.js* as the framework of choice. It uses a build system powered by *Gulp.js* [53] that is capable of building the source code into a browser extension that can be deployed with minimal effort. We utilize the term *browser extension* since we provide multi-browser support and can generate an extension for Google Chrome (Chromium), Firefox, and Opera, all using the same source code. For the generation of graphs on the side panel, we use D3.js [54], which is a JavaScript library that provides the visualization of graphs and charts.

Below, we shall discuss the navigational mechanisms implemented to enable an automated code change history browsing experience. As discussed in section 4.4, we gather all the inputs required by CodeTracker as soon as a user makes a click. This information, after being passed to the API, is stored in the `TreeView` class of CodeTracker Visualizer. When the user navigates to another page, we serialize this data in the form of a `JSON` string and store it in the browser's local storage with a *UUID* attached to it. We pass this *UUID* around redirects as a query parameter and then repopulate the graph and construct the graphical representation of the code change history. We also provide a *reset* button that a user may utilize to clear all the cache stored by the application in the browser's

local storage.

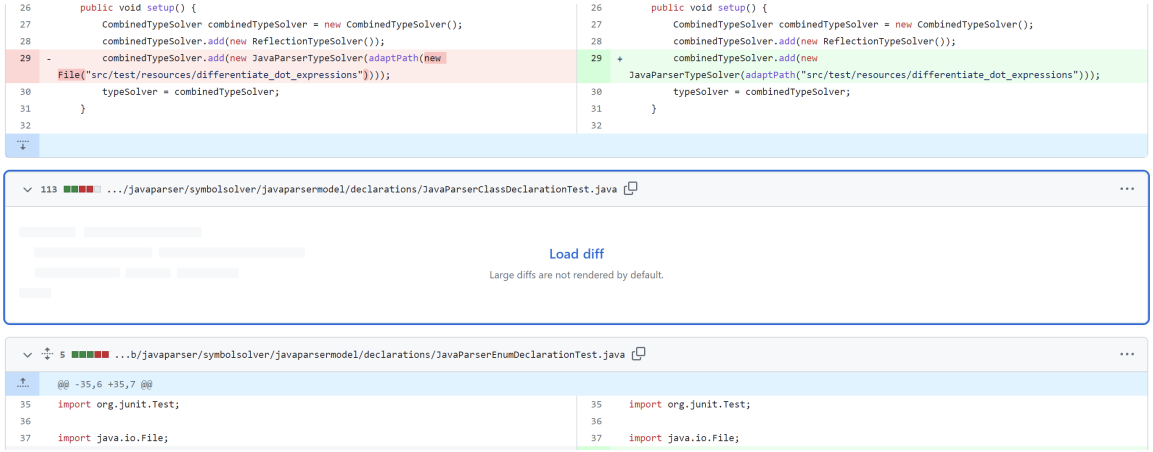


Figure 5.3: Large changes in the code difference view are not loaded by default on GitHub (project JavaParser [8]).

When we navigate to a new page, the code element of interest is at an unknown location on the page. Each file in a GitHub commit view is represented as a file container. The first thing we look for upon navigation is the file container that houses the code element. Two scenarios may occur here, the file container may either have been rendered and is present on the DOM, or it's not. If the file container is not present on the DOM, we emulate a scroll to the bottom of the page, as GitHub lazy loads files that are not in the viewport of the user. We then let the file load onto the DOM and once ready, we start looking for the line at which the code element is present.

When a file container loads, it may or may not have rendered the line of code on which the code element is declared. When a large change occurs within a file, GitHub doesn't load the code difference for the entire file and requires the user to click a load button that then lazy loads the code difference contents. Figure 5.3 shows this scenario. In such cases, we automate this click and obtain the code difference contents. In other scenarios with a regular amount of changes, the line containing the code element will be present in the code difference view if a change was made on or around that line. In cases where a change was not made around the required line, the code difference view hides these lines behind an *expand-view* button, as shown in Figure 5.4. This button expands a few lines in the direction specified by the button with an arrow icon. An *expand-all* button at the top of the file also exists, which makes the complete code difference view load. We automate a click on this button when our desired line hasn't been rendered. We chose to automate a click instead of

making the network call and appending the data onto the DOM ourselves to reduce the complexity of operations performed by our application, as well as make the tool future-proof since any changes in the implementation of the network call will automatically be handled by GitHub.

Figure 5.4: The expand-arrow buttons (marked in blue) and the expand-all button (marked in red) are used to lazy load more changes in a GitHub code difference container (project JavaParser [8]).

In some cases where the file is not too large, the *expand-all* button is missing on the GitHub user interface (Figure 5.6) and would require the user to click the *expand-view* arrow button in the desired direction multiple times to find the required line. In this case, we construct an *expand-all* button ourselves, attach the required network call onto it, and emulate a click as done in the above case. Doing this eliminates the need for making repeated network calls to render the contents of the entire file.

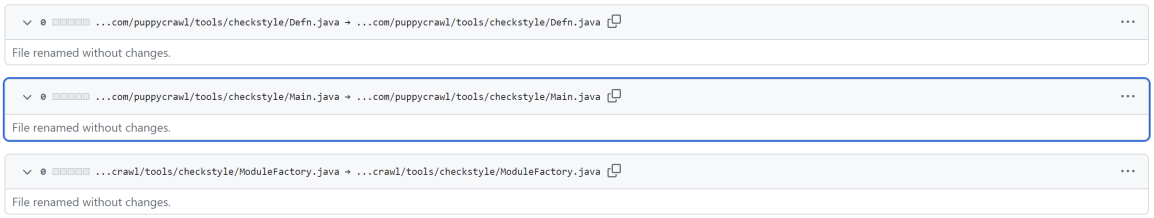


Figure 5.5: Renamed files have a minimized container on GitHub (project Checkstyle [9]).

As shown in Figure 5.5, in special cases where a file path has changed but no changes were made within the file, GitHub shows a minimized container with the change in file path displayed on it. In this case, we highlight the file container as there are no lines present here. In all other cases, we then generate a *Highlight ID*, which is an internal ID GitHub uses within its URLs to perform a highlight onto a line within a specific commit. Attaching this ID to the URL makes the GitHub web application scroll down to the specific line and highlight the code element to the

user. Figure 5.7 shows the highlight marker used to mark the desired code element. We use this approach over performing the scroll via our application since the URL generated this way can be bookmarked or shared and will be replicated on other machines, even on those that do not have CodeTracker Visualizer installed. By interacting directly with the DOM, we perform these actions at remarkable speeds, which in turn makes the user experience seamless even though a multitude of complex actions are taking place under the hood.

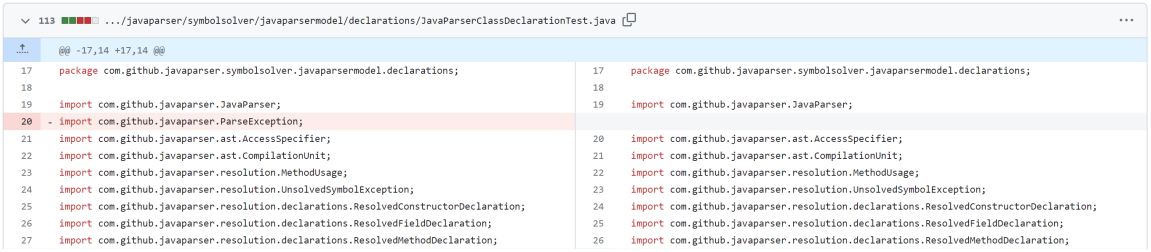


Figure 5.6: The expand-all button missing from the file container (project JavaParser [8]).

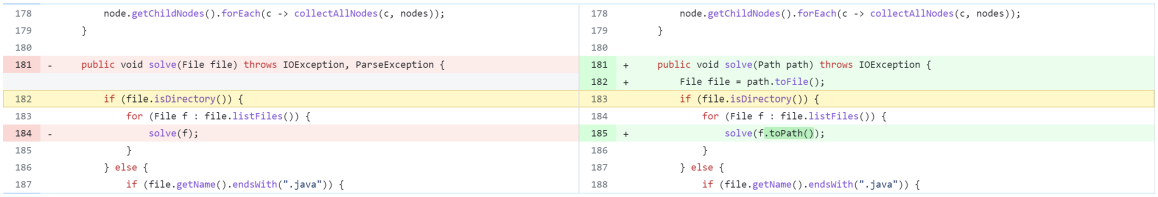


Figure 5.7: Code element highlighting on the GitHub GUI (project JavaParser [8]).

To ensure reliability across the development timeline, we implemented browser automation tests, that would automatically load up the latest version of CodeTracker Visualizer and run it through the tests that we designed compiling all the common use cases and navigational scenarios. For example, one of the tests loads the extension, opens a predetermined commit page, selects a code element, and initiates the tracking process for the code element. It then checks if the history was properly generated and rendered onto the screen, after which, it runs through the commits and verifies if the correct line is being highlighted for these commits. It obtains all the accurate change history information from the oracle on the file system, which it uses to cross-check the activities of the application. For the implementation of these tests, we made use of Jest [55] and Puppeteer [56], two popular JavaScript testing libraries. Code Snippet 4 shows a basic test implementation that checks if the extension loads and is visible to the user. The rest of the tests can be found in the CodeTracker Visualizer GitHub repository [57].

5.6 BlockTracker using GumTree

When CodeTracker was built, its accuracy and precision were compared against CodeShovel [2], the current state of the art at the time. Like CodeTracker, CodeShovel also supported the tracking of methods, which facilitated a fair and accurate comparison between the two tools. However, it does not support the tracking of code blocks. In fact, as seen in Chapter 1 and 2, there are not many, if any, tools that support the tracking of code blocks for modern Java. Since there are no direct competitors, we compare our accuracy against the current state-of-the-art Abstract Syntax Tree (AST) differencing tool, GumTree, which internally produces code statement mappings between two files. Performing an evaluation at this level would emphasize the accuracy of the underlying statement mapping generator we used, RefactoringMiner, against the current state-of-the-art GumTree [5], overall strengthening the accuracy of BlockTracker as a whole.

To evaluate our approach for tracking blocks, and for verifying the mappings generated by RefactoringMiner to be true, we built a version of BlockTracker that integrates with GumTree [5] for obtaining statement mappings rather than RefactoringMiner. This way, we would have a working version of CodeTracker's BlockTracker, that can utilize GumTree's statement mappings to track a program element through its commit history and follow its evolution chain [58].

In this version of BlockTracker, referred to from here on out as the GumTree version, we apply similar heuristics as we did with the RefactoringMiner version of BlockTracker, where we try to avoid making calls to the statement-mapping tool when possible. The GumTree version supports all the block change types supported by the RefactoringMiner version, as long as GumTree provides the required accurate statement mappings. To detect changes on a block, we initially utilized GumTree Actions, which indicate changes made on nodes within the file. However, we found that it produced false positives for our use case, hence we utilized the string representation approach as discussed in Section 4.2.

Since GumTree only accepts an input of two files, we iterate through the commits and pass the file from the current commit r and the parent commit p onto GumTree to obtain the statement mappings. To generate the statement mappings we utilize the *ClassicGumTree* matcher, as it is the

most complete implementation of the available matchers. GumTree defines a `Tree` class to represent all code elements in the source file, from a single block to the entire file itself. All GumTree matchers take in two `Tree` objects as parameters and generate statement mappings between these two trees. However, as discussed in Section 5.1, CodeTracker has individual classes for each code element, and it uses objects of these classes to keep track of the evolution chain, perform heuristics, and generate the change history graphs. These classes are integral to the design of CodeTracker, as are Trees to GumTree. Hence, we wrote adapter methods that would convert a CodeTracker code element object (i.e. `Method` and `Block`) to a GumTree code element object, i.e. `Tree`.

Once we generate the statement mappings, we find the appropriate mapping to the current block. When a match is made between blocks of two different versions, we update the block signature and continue the tracking process. Instances, where a file may have been renamed or moved to a different package, are handled using JGit, which provides the new file path given an old one. We also handle cases where the container method may have moved from one file (f_1) to another (f_2), but the file still exists in both commits r and p. Our initial approach was to check for instances where no matches were made for the container method to start looking for a method move. We would obtain, from JGit, the list of files that were modified in the parent commit p, and try to provide files one by one as input to GumTree until a match was made. However, we quickly realized that this was not a fair comparison as GumTree always expects to have the right file provided by the user. For example, looking at project Commons-Lang [59], in class `DateUtils.java`, tracking the block `if(R233)` back to its introduction, GumTree fails to find a mapping in the parent commit of the file due to the method `modify(R227)` having moved from another class, `CalendarUtils.java`.

Fujimoto et al. [60] presented the Staged Tree Matching technique to detect code elements being moved across files. According to this technique, by comparing the abstract syntax trees (ASTs) of different files in a project, we can detect structural similarities and infer code movements. We shall discuss this in more detail further below. Along with the study, they developed an extension tool for GumTree called Graftast [60], which aimed to allow statement matching between multiple files. As it is designed for use on a single version at a time, the Graftast API requires two directories of files amongst which it can generate statement mappings. Using the API as is to generate statement mappings for a single code element against an entire repository for multiple versions caused a significant

increase in the observed runtime. As discussed in Section 2.2, CodeTracker directly reads the Git repository to obtain the source code and cut down on processing time. To perform a fair comparison of execution time between the RefactoringMiner and GumTree versions of BlockTracker, similar to RefactoringMiner [3, 4] and CodeTracker, we would have to extend the Graftast APIs to read files across commits from the Git repository. We wanted to utilize an approach that could leverage RefactoringMiner's low execution time and strong accuracy on MOVE METHOD refactorings to find the file where the method moved, combined with the staged tree matching technique that can help produce the most accurate statement mappings from GumTree across more than one file [60].

Going ahead with this combination, we utilized the MOVE METHOD and CLASS SPLIT refactoring detection provided by RefactoringMiner to obtain the right file to compare with, and immediately noticed an improvement in accuracy. RefactoringMiner is a state-of-the-art refactoring detection tool with a precision and recall of 99.6% and 94% respectively [4]. Hence, the refactoring information it provides can be considered to be accurate.

Apart from providing the right file, Similar to Fujimoto et al. [60], we adopt a two-round staged tree matching methodology to help assist GumTree. In their study, Fujimoto et al. showed that utilizing this technique produced much more accurate results when generating statement mappings for a code element in a set of multiple files with GumTree. When a method moves from file f_{1_r} in commit r to f_{2_p} in commit p, we apply the first stage of matching. Here, we make GumTree match f_{1_r} to f_{1_p} and pick out the unmapped nodes in f_{1_r}. For the second stage of matching, we make a mapping between the unmapped nodes of f_{1_r} to f_{2_p}, to ensure that only the unmapped nodes have the potential to make a match. This methodology is only applicable in cases where f_{1_p} exists, i.e., the method was moved from one file (f_{1_r}) to another (f_{2_p}) and the former file (f_{1_p}) remains in the parent commit p.

In instances where a match for the block was not found in the parent version after generating the statement mappings, we try to match the container method of the block. If a match for this method is found in the parent version, we can safely declare that the block was *Introduced* in the current commit, since the block does not exist in the container method in the parent version i.e. M_p, as described in **Step 5** of Section 4.2. If the method does not have a match either, then we know that both the method and the block were *Introduced* in the current commit, and we append the change

history to indicate this. After appending the change history with the respective change made, we convert the GumTree TREE object back to a CodeTracker BLOCK object and continue the tracking process.

Apart from providing it with the right file path, to further assist GumTree, we attempted to match the container method in commit r (M_r) with the file from its parent commit p (f_{1_p}), instead of matching the entire files from both commits (f_{1_r}, f_{1_p}), which would potentially speed up the processing time needed to make a match for M_r to M_p. However, with a small subset of our oracle containing ten blocks from the repository Checkstyle, we found that this led to a decrease in the accuracy of the mappings returned by GumTree when compared to using the entire file to generate mappings on the same data sample. We believe this is because GumTree uses information from M_r's sibling nodes to make more informed mappings. Specifically, in cases where statements are duplicated, GumTree loses precision in computing tiebreakers for multiple candidates, as it lacks the additional information to properly compute tiebreakers [5]. We observed a drop of -2.23% and -1.49% in precision and recall on the change level, respectively. On the commit level, we recorded a drop of -1.12% in precision, however, the recall remained the same. We did not test it with the entire oracle since the drop in accuracy and recall will only compound to become larger over more instances.

We also experimented with alternate matchers, namely the *GumTreeSimple* matcher, and computed its accuracy on the data sample mentioned above. We found that the accuracy again dropped when compared to using the *ClassicGumTree* matcher. On the change level, we observed a drop of -5.83% and -2.98% in precision and recall respectively. On the commit level, we saw a -3.17% drop in precision, while the recall remained the same.

Overall, we found that the accuracy of GumTree's statement mappings was significantly lesser compared to RefactoringMiner, and naturally, the block history generated by this version of BlockTracker was highly inaccurate. We shall take an in-depth look at the difference in performance in Chapter 6.

Listing 4 GUI Functionality test implementation using Jest and Puppeteer

```
const puppeteer = require('puppeteer');

let URL = 'https://github.com/checkstyle/checkstyle/commit/746a9d691';

describe('Basic functionality', () => {

    beforeAll(async () => {
        await page.goto(`${URL}`);

        let loadExtension = async () => {
            const pathToExtension = require('path')
                .join(__dirname, '../tmp/chrome');
            const browser = await puppeteer.launch({
                headless: 'chrome',
                args: [
                    `--disable-extensions-except=${pathToExtension}`,
                    `--load-extension=${pathToExtension}`,
                ],
            });
            const backgroundPageTarget = await browser.waitForTarget(
                (target) => target.type() === 'background_page'
            );
            const backgroundPage = await backgroundPageTarget.page();
            await browser.close();
        }
        await loadExtension();
    });

    it('should load the sidebar', async () => {
        await expect(page.content()).resolves.toContain('octotree');
    });

    it('should pin the sidebar', async () => {
        let pinButton = await page.$('body > nav >
        div.octotree-main-icons > a.octotree-pin')
        await pinButton.evaluate((b) => b.click());
        await page.waitForTimeout(500);
        await expect(page.content()).resolves.toContain('octotree-pinned');
    });
});
```

Chapter 6

Evaluation

This chapter goes over our contributions made to the oracle containing code change history for various code element types in Section 6.1. In Section 6.2, we evaluate the performance and accuracy of BlockTracker, while in Section 6.3, we do the same for the GumTree version of BlockTracker, and perform a direct comparison of it against the RefactoringMiner version in terms of precision and recall. Finally, in Section 6.4, we outline the execution time for both versions of BlockTracker and also dive into the caching mechanism we implemented for the GumTree version of BlockTracker to have a fair comparison of execution time.

6.1 Oracle Contributions

The oracle made available with CodeTracker by Jodavi and Tsantalis [2] contains 2,657 different files containing accurate code change history. This oracle has been constructed as an extension to the oracle provided by Grund et al. containing the change history of 200 methods over 20 open-source project repositories. Of these 200 methods, Jodavi and Tsantalis extended it to include 1,112 attributes and 1,345 variables. The approach followed to extend this oracle was semi-automated. Since the method change history was verified as being accurate by two different research groups, we can consider it to be a reliable source of truth. Using this as a baseline, Jodavi and Tsantalis [1] collected all the variables declared in these methods and performed a backward tracking process to generate code change histories for variables using CodeTracker. Once

these changes were generated, they checked to see if the changes were a subset of the changes made on the body of the respective container method, and since the variable was contained inside the method, it was logically deemed to be accurate change history [1]. A similar approach was also implemented for all attributes declared in the type declaration of a particular method, however, the same conditions of having the change history be a subset do not hold in this case, as an attribute can be used independently of any method declared in its container type declaration. We fix this by only including attributes that were referenced within the respective method and removing the change histories for attributes that were not. Apart from this, we also discovered an error with the Guava graph implementation utilized within CodeTracker. The APIs being used, `ValueGraph.predecessors(N node)` and `ValueGraph.successors(N node)`, are marked by the developers as being unstable and caused valid changes that were being discovered by the tool to be omitted in the change history. For example, in project *JavaParser*, in commit `37f93be`, and file `SourceFileInfoExtractor.java`, attribute `ko` on line 37 was moved over here from file `ProjectResolver.java`. This change is tracked in the oracle presented by Jodavi and Tsantalis [1], however, another change was also made here, which was missing in the oracle. The modifier for the attribute changed, and more specifically, the static keyword was removed. As shown in Figure 6.1, this change, although it was tracked by CodeTracker, was not included in the change history graph being returned due to the unstable nature of the API being used.

Figure 6.1: Unrecorded changes are now added to the oracle.

We fixed this issue in many such cases and provided an updated version of the entire oracle for methods, variables, and attributes. We also perform enrichment of the oracle with additional information on a change that occurred in a commit, where applicable (Figure 6.2). For example, for the method oracle, we added two new types of possible changes, METHOD SPLIT and METHOD MERGE. This is possible since RefactoringMiner now supports multi-mappings, and can detect splits and merges of code elements. We also manually verified the accuracy of each oracle file using the oracle validation tool mentioned in Section 4.5 and made corrections to a few discrepancies we found. A complete breakdown of the changes is shown in Tables 6.1 (Method oracle), 6.2 (Variable oracle), 6.3 (Attribute oracle).

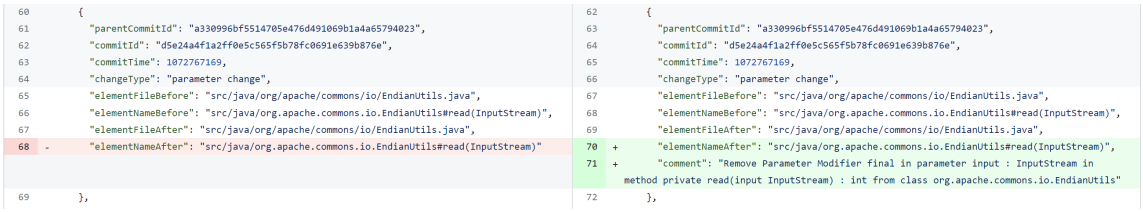


Figure 6.2: Enrichment of the oracle by adding extra information about the change made.

Apart from this, we also extend the oracle to include the code change history of control-flow code blocks. We present a code block change history oracle including 1280 different code blocks that were referenced within the 200 methods picked by Grund et al. [2]. We adopt the semi-automated oracle generation approach detailed above and match the change history as a subset of its container method. The idea employed here is that when a block contained within a method matches its accurate change history exactly to its introduction, we can deduce that the block was introduced alongside the method and underwent changes that have already been verified. We can hence safely declare the change history for this case as being accurate. In cases where the change history is not an exact subset, we manually verified the oracle using the oracle validation tool mentioned in Section 4.5 and made corrections to the block oracle where deemed necessary.

After all of our validations, we generated the change history of all the blocks with GumTree BlockTracker (discussed in Section 5.6) and performed a one-to-one comparison with our oracle. Upon finding a mismatch, we utilized Oracle Validator to manually verify both versions of the code change history and corrected our oracle in cases where required. This helped us correct instances where the code change history may seem to be accurate but has a rather different change being

performed on it upon closer inspection. For example, a block that may seem to have been introduced in a commit may have undergone a drastic change which RefactoringMiner may not match as equal. More specifically, if a block with a minimal body and expression undergoes a major change in both its expression and body, RefactoringMiner may fail to map these blocks, since it performs bottom-up matching [3, 4] as proposed by Fluri et al. [61]. However, since GumTree uses a different method of statement mapping generation, more specifically a tree matching algorithm [5], it may find success in cases where RefactoringMiner could not. There were a small number of cases where such changes as described above took place and we have recorded and corrected all instances.

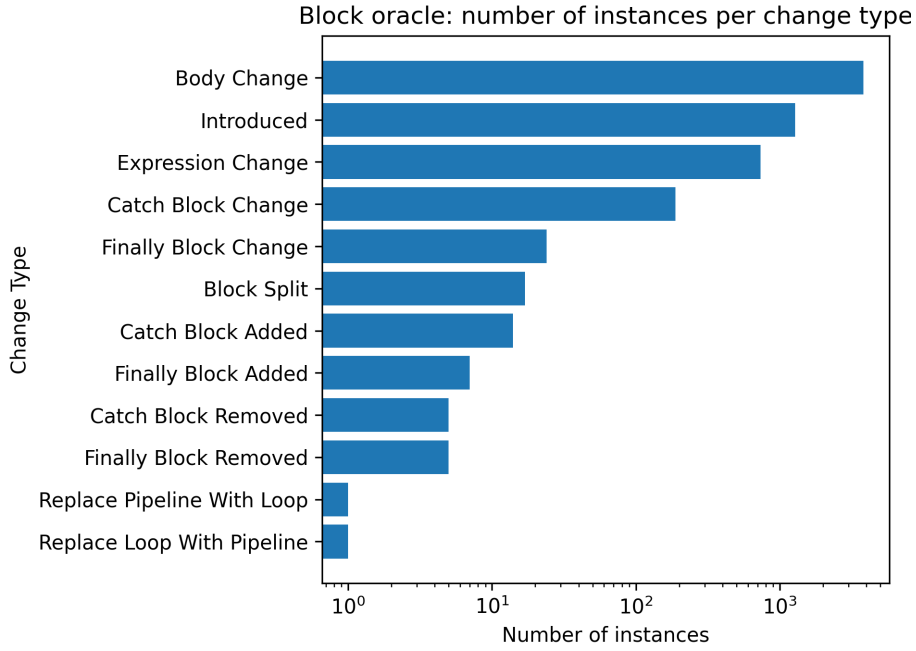


Figure 6.3: Block oracle: number of instances per change type

Tables 6.1 (Method oracle), 6.2 (Variable oracle), 6.3 (Attribute oracle) provide a complete breakdown of the change type instances in the oracle before and after our changes. The changes are grouped by being common with the original oracle (**C** columns), removed from the original oracle (**R** columns), and new changes added to the original oracle (**N** columns) for both training and testing sets.

Figure 6.3 and Table 6.4 detail the number of instances per change kind for control flow code blocks. Table 6.5 details the number of instances per block type in the block oracle.

Table 6.6 and Figure 6.4 describe the number of blocks per method processed.

Table 6.1: Updates in the method oracle created by Jodavi et al. [1]

	Training set			Testing set		
Change Type	**C**	**R**	**N**	**C**	**R**	**N**
Body Change	2297	8	10	463	22	23
Documentation Change	437	2	4	92	1	2
Container Change (File Move)	264	2	3	183	4	7
Parameter Change	221	5	8	72	4	5
Introduced	100	0	2	92	8	8
Return Type Change	48	3	0	16	0	0
Annotation Change	42	0	0	23	1	1
Modifier Change	46	0	1	17	0	0
Exception Change	40	0	0	7	0	0
Method Move	21	0	1	12	5	2
Rename	18	3	1	18	0	0
Method Merge	-	-	6	-	-	0
Method Split	-	-	2	-	-	0
Total	3534	23	38	995	45	48

C: common **R**: removed **N**: new

Overall, the block oracle has undergone multiple rounds of validations by multiple researchers from our lab and compared against another state-of-the-art statement mapping generation tool, and can hence be considered reliable. All change history oracles (method, variable, field, and block) are made available on our GitHub repository [6] to enable the replication of our experiments and enable the possibility of future research on code element tracking techniques.

Table 6.6: Number of blocks per method included in the oracle.

Method Name	**Number of Blocks**
checkstyle-Checker-fireErrors	4
checkstyle-Checker-process	2
checkstyle-CommonUtils-createPattern	2
checkstyle-FinalLocalVariableCheck-visitToken	6
checkstyle-JavadocMethodCheck-checkThrowsTags	6
checkstyle-Main-main	9
checkstyle-TreeWalker-processFiltered	4
commons-io-DemuxOutputStream-flush	1
commons-io-EndianUtils-read	1
commons-io-FilenameUtils-wildcardMatch	15
commons-io-IOUtils-contentEqualsIgnoreEOL	2
commons-io-ProxyWriter-write	3
commons-io-Tailer-run	21

commons-io-XmlStreamReader-doHttpStream	3
commons-lang-DateUtils-modify	23
commons-lang-DurationFormatUtils-formatPeriod	19
commons-lang-EqualsBuilder-reflectionAppend	12
commons-lang-FastDatePrinter-parsePattern	12
commons-lang-LocaleUtils-toLocale	10
commons-lang-NumberUtils-createNumber	39
commons-lang-NumberUtils-isCreatable	25
commons-lang-RandomStringUtils-random	14
commons-lang-StrSubstitutor-substitute	17
elasticsearch-BulkRequest-add	1
elasticsearch-ESFileStore-getUnallocatedSpace	1
elasticsearch-IndicesSegmentResponse-addCustomXContentFields	12
elasticsearch-IndicesSegmentResponse-toXContent	2
elasticsearch-IndicesService-verifyIndexMetadata	3
elasticsearch-NodesFaultDetection-handleTransportDisconnect	4
elasticsearch-RestTable-expandHeadersFromRequest	7
flink-CheckpointCoordinator-receiveAcknowledgeMessage	9
flink-ContinuousFileMonitoringFunction-close	3
flink-DispatcherRestEndpoint-initializeHandlers	4
flink-FileSystem-getUnguardedFileSystem	15
flink-KryoSerializer-checkKryoInitialized	3
flink-LocatableInputSplitAssigner-getNextInputSplit	24
flink-PojoSerializer-deserialize	13
flink-RemoteStreamEnvironment-executeRemotely	10
hadoop-ClientRMService-renewDelegationToken	3
hadoop-ConverterUtils-convertFromYarn	1
hadoop-FiCaSchedulerNode-unreserveResource	2
hadoop-FifoScheduler-allocate	5
hadoop-FifoScheduler-getAppsInQueue	2
hadoop-LeveldbConfigurationStore-retrieve	2
hadoop-NodeReportPBImpl-setCapability	1
hadoop-RMServerUtils-normalizeAndValidateRequests	3
hadoop-Sch(...)mpt-getRunningAggregateAppResourceUsage	3
hibernate-orm-AnnotationBinder-bindClass	28
hibernate-orm-CollectionBinder-bind	18
hibernate-orm-DefaultRefreshEventListener-onRefresh	19
hibernate-orm-QueryBinder-bindNativeQuery	11
hibernate-orm-QueryBinder-bindQuery	7
hibernate-orm-RevisionInfoConfiguration-configure	12
hibernate-orm-SimpleValue-buildAttributeConverterTypeAdapter	4
hibernate-search-ArrayBridge-indexNotNullArray	1
hibernate-search-ClassLoaderHelper-getNoArgConstructor	3
hibernate-search-new TwoPhaseIterator-matches	2
hibernate-search-NumericFieldUtils-createNumericRangeQuery	13

hibernate-search-TokenizerChain-createComponents	1
intellij-community-Art(...)mpl-getFileToArtifactsMap	1
intellij-community-CompilerManagerImpl-removeCompiler	1
intellij-community-ExceptionBreakpoint-getThisObject	1
intellij-community-Tra(...)tor-isInContentOfOpenedProject	3
javaparser-Difference-apply	5
javaparser-Difference-applyRemovedDiffElement	13
javaparser-JavaParserFacade-convertToUsage	16
javaparser-JavaParserFacade-getTypeConcrete	1
javaparser-LambdaExprContext-solveSymbolAsValue	15
javaparser-MethodCallExprContext-solveMethod	2
javaparser-MethodCallExprContext-solveMethodAsUsage	7
javaparser-MethodResolutionLogic-isApplicable	18
javaparser-SourceFileInfoExtractor-solve	7
jetty.project-AnnotationIntrospector-introspect	6
jetty.project-FragmentDescriptor-processAfters	6
jetty.project-HttpTransportOverHTTP2-push	3
jetty.project-HttpTransportOverHTTP2-send	15
jetty.project-JsrSession-addMessageHandler	4
jetty.project-Module-equals	4
jetty.project-ServletHolder-doStart	19
jetty.project-StdErrLog-escape	5
jgit-CommitCommand-call	27
jgit-IndexDiff-diff	40
jgit-MergeCommand-call	23
jgit-PackWriter-findObjectsToPack	49
jgit-PackWriter-writePack	15
jgit-PullCommand-call	22
jgit-RebaseCommand-call	24
jgit-RepoCommand-call	34
jgit-UploadPack-sendPack	30
junit4-BlockJUnit4ClassRunner-describeChild	1
junit4-BlockJUnit4ClassRunner-methodBlock	2
junit4-BlockJUnit4ClassRunner-runChild	1
junit4-BlockJUnit4ClassRunner-withPotentialTimeout	1
junit4-ParentRunner-applyValidators	2
junit4-RunnersFactory-createRunners	1
junit4-RunnersFactory-createRunnersForParameters	3
junit5-ClassTestDescriptor-invokeBeforeAllMethods	2
junit5-DefaultLauncher-discoverRoot	2
junit5-NodeTestTask-execute	3
junit5-TestMethodTestDescriptor-execute	3
lucene-solr-ConcurrentMergeScheduler-sync	10
lucene-solr-ConcurrentMergeScheduler-updateMergeThreads	18
lucene-solr-ConstantScoreQuery-rewrite	3

lucene-solr-Field-tokenStream	9
lucene-solr-IndexWriter-shutdown	7
lucene-solr-IndexWriter-writeSomeDocValuesUpdates	13
lucene-solr-MemoryIndex-keywordTokenStream	1
lucene-solr-MemoryIndex-storeDocValues	8
lucene-solr-QueryParserBase-addClause	10
lucene-solr-QueryParserBase-newRangeQuery	2
mockito-InvocationsFinder-findPreviousVerifiedInOrder	1
mockito-MatchersBinder-bindMatchers	1
mockito-PluginLoader-loadPlugin	5
mockito-ReturnsArgumentAt-answer	1
mockito-StringUtil-join	2
mockito-VerificationStartedNotifier-notifyVerificationStarted	2
okhttp-AsyncCall-execute	5
okhttp-ConnectionPool-get	2
okhttp-Http2Codec-writeRequestHeaders	1
okhttp-Http2Connection-newStream	8
okhttp-Http2Connection-pushStream	1
okhttp-JavaNetAuthenticator-authenticate	4
okhttp-OkHttpURLConnection-getResponse	15
okhttp-ReaderRunnable-headers	7
okhttp-RealConnection-connect	15
pmd-ClassTypeResolver-visit	7
pmd-CommentUtil-javadocContentAfter	5
pmd-FormalComment-findJavadocs	2
pmd-JUnitTestsShouldIncludeAssertRule-visit	1
pmd-SourceFileScope-getSubTypes	3
spring-boot-ConfigurationPropertiesBinder-getBindHandler	3
spring-boot-DefaultErrorAttributes-addErrorMessage	2
spring-boot-ErrorPageRegistrarBeanPostProcessor-getRegistrars	1
spring-boot-JsonParserFactory-getJsonParser	3
spring-boot-Ser(...)ext-prepareWebApplicationContext	6
spring-boot-SpringApplication-getBeanDefinitionRegistry	2
spring-boot-StaticResourceJars-addUrlConnection	1
spring-boot-UndertowWebServer-getPortFromChannel	1
spring-framework-Abs(...)sor-writeWithMessageConverters	26
spring-framework-AbstractNestablePropertyAccessor-setPropertyValue	1
spring-framework-Conf(...)der-loadBeanDefinitionsForBeanMethod	13
spring-framework-Conf(...)ser-doProcessConfigurationClass	12
spring-framework-Conf(...)sor-processConfigBeanDefinitions	18
spring-framework-ConstructorResolver-instantiateUsingFactoryMethod	39
spring-framework-GenericConversionService-getConverter	3
spring-framework-ServletHttpHandlerAdapter-service	5
spring-framework-TypeConverterDelegate-convertIfNecessary	36
Total	1280

Table 6.2: Updates in the variable oracle created by Jodavi et al. [1]

	Training set			Testing set		
Change Type	**C**	**R**	**N**	**C**	**R**	**N**
Introduced	835	69	189	294	30	91
Type Change	219	19	52	62	0	12
Modifier Change	101	14	32	50	0	4
Rename	133	7	28	20	1	9
Annotation Change	13	0	0	4	0	0
Total	1301	109	301	430	31	116

C: common **R**: removed **N**: new

Table 6.3: Updates in the attribute oracle created by Jodavi et al. [1]

	Training set			Testing set		
Change Type	**C**	**R**	**N**	**C**	**R**	**N**
Container Change	434	8	23	52	5	89
Introduced	340	14	18	42	9	91
Type Change	96	1	4	13	2	22
Rename	64	4	7	4	1	10
Modifier Change	35	0	9	6	3	34
Access Modifier Change	36	1	2	0	0	33
Field Move	17	1	9	5	0	21
Annotation Change	2	0	0	0	0	0
Total	1024	29	72	122	20	300

C: common **R**: removed **N**: new

6.2 BlockTracker Accuracy

To obtain block-related refactorings, We first instantiate the methods containing the desired block from both the child and parent commit. We then retrieve all the refactorings on these two methods from RefactoringMiner and then infer these refactorings onto the blocks in their bodies (e.g., a MOVE METHOD refactoring also moves the block it contains). Likewise, if the methods have no changes, then the blocks inside them also have no changes.

In cases where a change has occurred in the method body, we obtain the block's string representation and statement mappings from RefactoringMiner and use this information to deduce the various changes that have occurred (e.g., EXPRESSION CHANGE, BODY CHANGE) between both versions. Therefore, the FPs are due to incorrect statement mappings, while the FNs are due to

Table 6.4: Number of instances per change type for blocks.

Change Type	Training set	Testing set
Body Change	3294	526
Introduced	964	316
Expression Change	612	120
Catch Block Change	139	50
Finally Block Change	23	1
Block Split	14	3
Catch Block Added	7	7
Finally Block Added	4	3
Catch Block Removed	2	3
Finally Block Removed	4	1
Replace Pipeline With Loop	1	0
Replace Loop With Pipeline	1	0
Total	5065	1030

RefactoringMiner's inability to match some statement pairs.

For example, in project *Checkstyle* [10] (Figure 6.5), block `if(L219)` is mapped to `if(R213)` by RefactoringMiner, causing an incorrect fork in the change history of the block, making the rest of the history inaccurate.

Looking at project *Apache Flink* [11] (Figure 6.6), block `if(L354)` is mapped to `if(R374)`, which in this case is incorrect. Block `if(R374)` was actually introduced in this commit, but RefactoringMiner is unable to deduce this. This is another case that affects our overall accuracy.

In project *Checkstyle* [12] (Figure 6.7), block `if(R618)` is erroneously declared as INTRODUCED, while it should have been matched to block `if(L610)` in the parent commit. The expression and body of the block both change in this commit. While the expression change is reasonable, the body has changed completely, making RefactoringMiner miss this particular mapping.

Another example in project *Checkstyle* [13] (Figure 6.8), block `if(L165)` moves from method `processCommandLine` to method `main`, now being block `if(R113)`. A mapping between these two blocks is not made by RefactoringMiner, causing the history to abruptly end in this commit.

In project *Jetty.Project* [14] (Figure 6.9), block `if(L309)` in the parent commit is split into `if(R301)` and `if(R319)`, the condition in the expression is inverted, and the body of the if

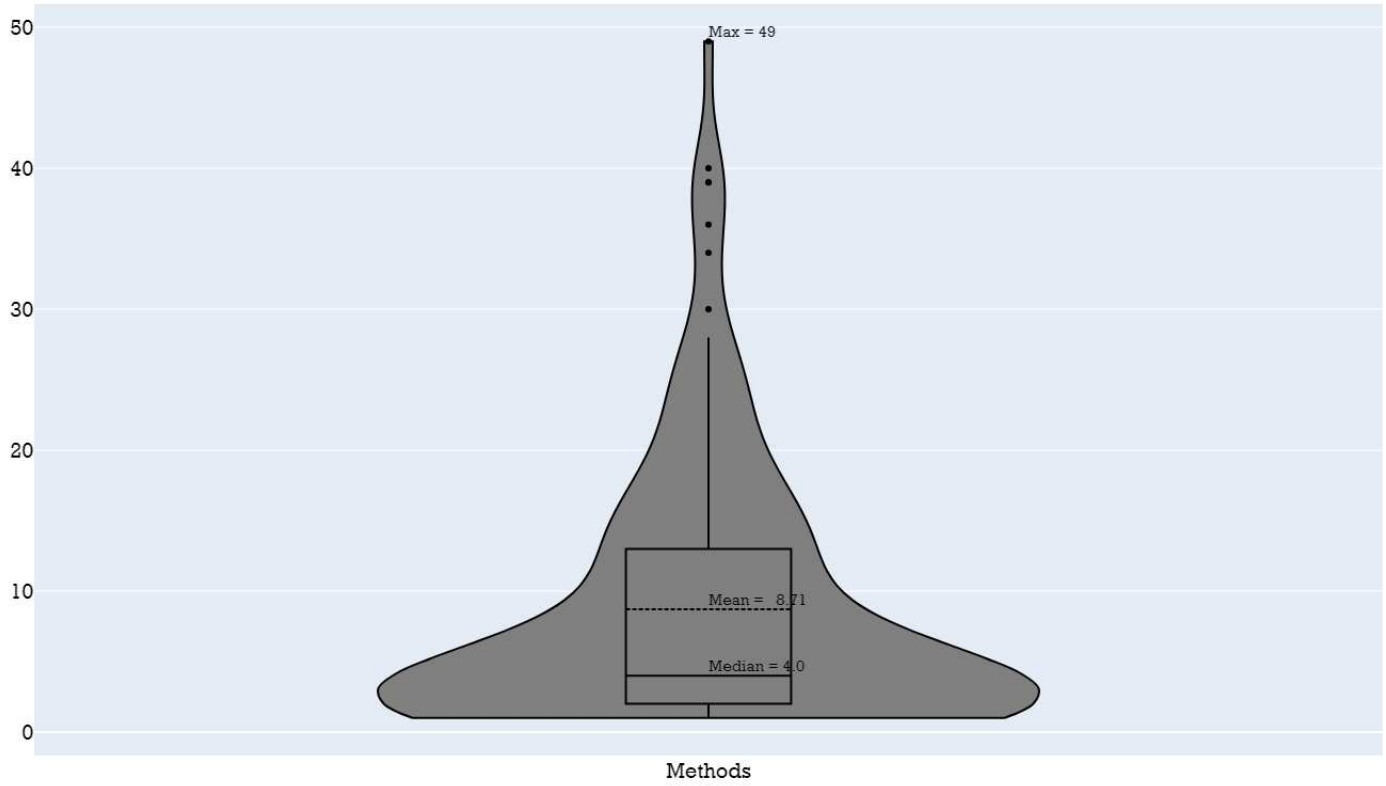


Figure 6.4: Violin plot showing the number of blocks per method.

block is moved to the `else` block. In this case, block `if(R319)` is incorrectly declared as INTRODUCED, when it should instead be matched with block `if(L309)` in the parent commit. This is due to RefactoringMiner not generating a mapping between the two code blocks.

Despite the large number of changes, RefactoringMiner is theoretically able to correctly identify the scenario and provide accurate mappings, however, in this case since the block is split into two identical clones with a plethora of modifications from the original, it fails to recognize the changes. Although this case was inaccurate, the silver lining here is that the other block from this split, `if(R301)`, is correctly matched to `if(L309)` and can be traced back to its introduction.

The problems we discussed here have been created into issues on RefactoringMiner's GitHub repository, and as the fixes for these issues come in, the accuracy of our tool shall also improve.

Table 6.7 contains the computed precision and recall of CodeTracker while tracking code blocks. We report these values at two levels, the first being the *commit level* (i.e., finding the commits in

Figure 6.5: Incorrect mapping (False Positive), project *Checkstyle*. [10]

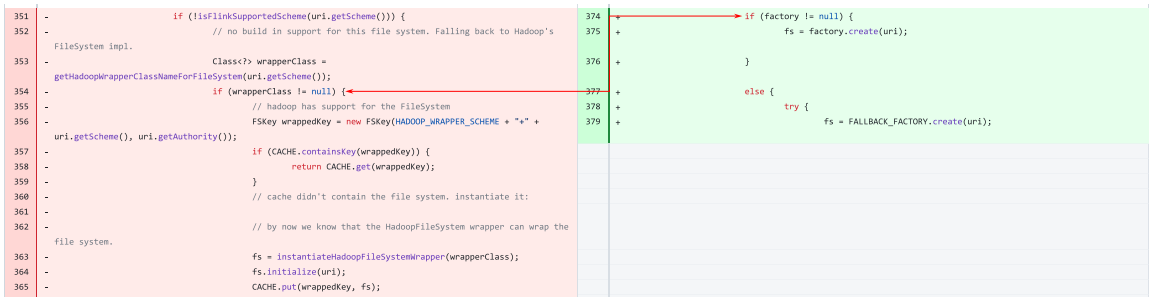
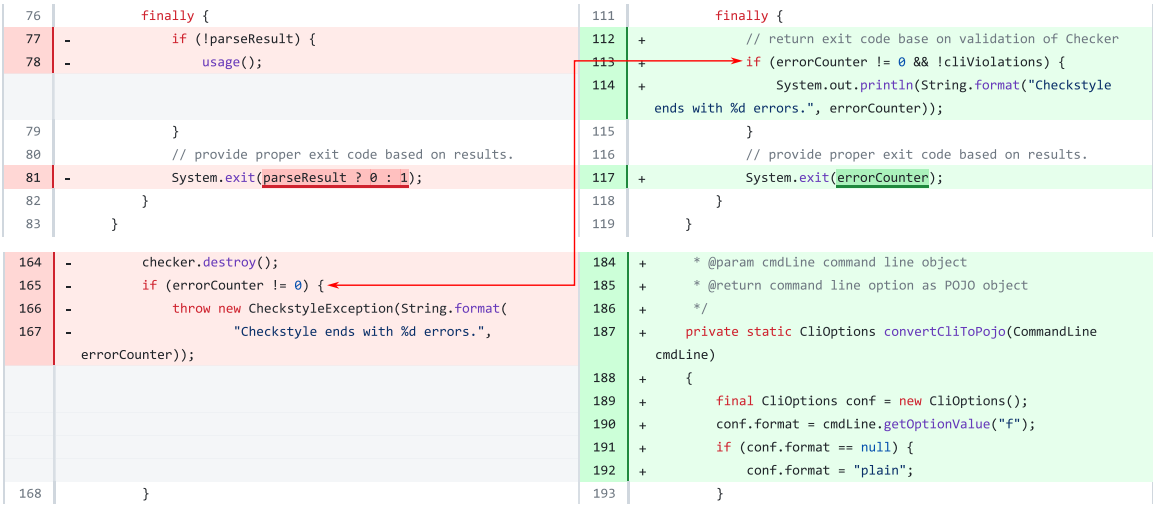


Figure 6.6: Incorrect mapping (False Positive), project *Apache Flink*. [11]

Figure 6.7: Missed mapping (False Negative), project *Checkstyle*. [12]



```
76          finally {
77  -           if (!parseResult) {
78  -               usage();
79              }
80              // provide proper exit code based on results.
81  -           System.exit(parseResult ? 0 : 1);
82          }
83      }

164 -       checker.destroy();
165 -       if (errorCounter != 0) {
166 -           throw new CheckstyleException(String.format(
167 -                   "Checkstyle ends with %d errors.",
        errorCounter));
168         }

111         finally {
112 +           // return exit code base on validation of Checker
113 +           if (errorCounter != 0 && !cliViolations) {
114 +               System.out.println(String.format("Checkstyle
        ends with %d errors.", errorCounter));
115             }
116             // provide proper exit code based on results.
117 +           System.exit(errorCounter);
118         }
119     }

184 +     * @param cmdLine command line object
185 +     * @return command line option as POJO object
186 +     */
187 +    private static CliOptions convertCliToPojo(CommandLine
        cmdLine)
188 +    {
189 +        final CliOptions conf = new CliOptions();
190 +        conf.format = cmdLine.getOptionValue("f");
191 +        if (conf.format == null) {
192 +            conf.format = "plain";
193         }
```

Figure 6.8: Missed mapping (False Negative), project *Checkstyle*. [13]

```
298             makeUnavailable(ue);
299 -           throw ue;
300         }
301
302         try
303         {
304             checkServletType();
305         }
306         catch (UnavailableException ue)
307         {
308             makeUnavailable(ue);
309 -           if (!_servletHandler.isStartWithUnavailable())
310 -               throw ue; //servlet is not an instance of
        javax.servlet.Servlet
311         }

300             makeUnavailable(ue);
301 +           if (_servletHandler.isStartWithUnavailable())
302 +           {
303 +               LOG.ignore(ue);
304 +               return;
305 +           }
306 +           else
307 +               throw ue;
308         }
309
310 +
311 +       //servlet is not an instance of javax.servlet.Servlet
312         try
313         {
314             checkServletType();
315         }
316         catch (UnavailableException ue)
317         {
318             makeUnavailable(ue);
319 +           if (_servletHandler.isStartWithUnavailable())
320 +           {
321 +               LOG.ignore(ue);
322 +               return;
323 +           }
324 +           else
325 +               throw ue;
326         }
```

Figure 6.9: Missed mapping (False Negative), project *Jetty.Project*. [14]

Table 6.5: Number of instances per block type included in the oracle.

Block Type	Number of Instances
If Statement	929
Enhanced For Statement	87
Try Statement	81
Catch Clause	80
While Statement	34
Synchronized Statement	23
For Statement	18
Finally Block	15
Switch Statement	10
Do Statement	3
Total	1280

Table 6.7: Block tracking precision/recall for CodeTracker.

Dataset	Level	TP	FP	FN	Precision	Recall
Training	Commit	4590	68	106	98.54	97.74
	Change	4890	115	159	97.70	96.85
Testing	Commit	901	37	28	96.06	96.99
	Change	952	63	49	93.79	95.10
Overall	Commit	5491	105	134	98.12	97.62
	Change	5842	178	208	97.04	96.56

which a code block changed), and the *change level* (i.e., finding the kinds of changes that occurred in the commits in which a code block changed). In section 6.3, we shall look into how our performance compares to GumTree [5] in terms of statement mapping accuracy.

6.3 BlockTracker using GumTree Accuracy

As discussed in section 5.6, we developed a version of BlockTracker that uses GumTree as a statement mapping provider instead of RefactoringMiner. We did this to compare the accuracy of our solution versus the current state-of-the-art in statement mapping and code difference detection. We shall now compare the performance of both tools and then discuss our findings.

Table 6.8: Block tracking precision/recall at change level (GumTree comparison)

Dataset	Tool	TP	FP	FN	Precision	Recall
Training	GumTree	3901	525	1148	88.14	77.26
	RefactoringMiner	4890	115	159	97.70	96.85
Testing	GumTree	841	138	160	85.90	84.02
	RefactoringMiner	952	63	49	93.79	95.10
Overall	GumTree	4742	663	1308	87.73	78.38
	RefactoringMiner	5842	178	208	97.04	96.56

Table 6.8 and 6.9 show the commit and change level accuracy of block tracking with CodeTracker using RefactoringMiner versus GumTree. From our experiments, it is evident that RefactoringMiner provided significantly more accurate statement mappings when compared to GumTree. Looking at the true-positive to false-positive ratio, we can infer that GumTree misses more cases than makes an incorrect match. This is due to the fact that a single incorrect mapping will cause the rest of the evolution chain to be incorrect.

Table 6.9: Block tracking precision/recall at commit level (GumTree comparison)

Dataset	Tool	TP	FP	FN	Precision	Recall
Training	GumTree	3838	255	858	93.77	81.73
	RefactoringMiner	4590	68	106	98.54	97.74
Testing	GumTree	849	64	80	92.99	91.39
	RefactoringMiner	901	37	28	96.06	96.99
Overall	GumTree	4687	319	938	93.63	83.32
	RefactoringMiner	5491	105	134	98.12	97.62

One of the prominent reasons for GumTree's relatively poor performance is due to the occurrence of inaccurate statement mappings. For example, looking at *Commons-Lang* [62], in file `EqualsBuilder.java`, `if(R533)` is introduced in this commit. However, GumTree falsely matches this block to `if(L503)` in its parent commit. Errors of this kind cause the code change history to have irrelevant changes, causing GumTree BlockTracker's overall precision to go down.

Our observations show that GumTree inherently does not map one type of code block to another. This causes it to miss a lot of potential mappings that it could've made in cases where the type of the block was changed and ultimately makes a large dip in the accuracy of its mappings. For example,

in Apache's project, Commons-Lang [63], in file `DurationFormatUtils.java`, `if(L319)` is modified to `while(R319)` to be able to replicate the action in its body more than once. While Refactoring Miner accurately maps these two code elements, GumTree does not. In this particular instance, due to not making the match, GumTree's version of BlockTracker reports the while-block as being INTRODUCED, causing it to miss out on six other changes that occur on the block before it is actually introduced. Instances of such kind cause a major drop in the recall presented in Table 6.8 and 6.9 for GumTree BlockTracker.

Overall, RefactoringMiner BlockTracker achieves higher precision with +9.31% and +18.18% in recall on the change level. For the commit level, we see a similar greater number in performance with a +4.49% increment in precision and +14.30% in recall.

6.4 Execution Time

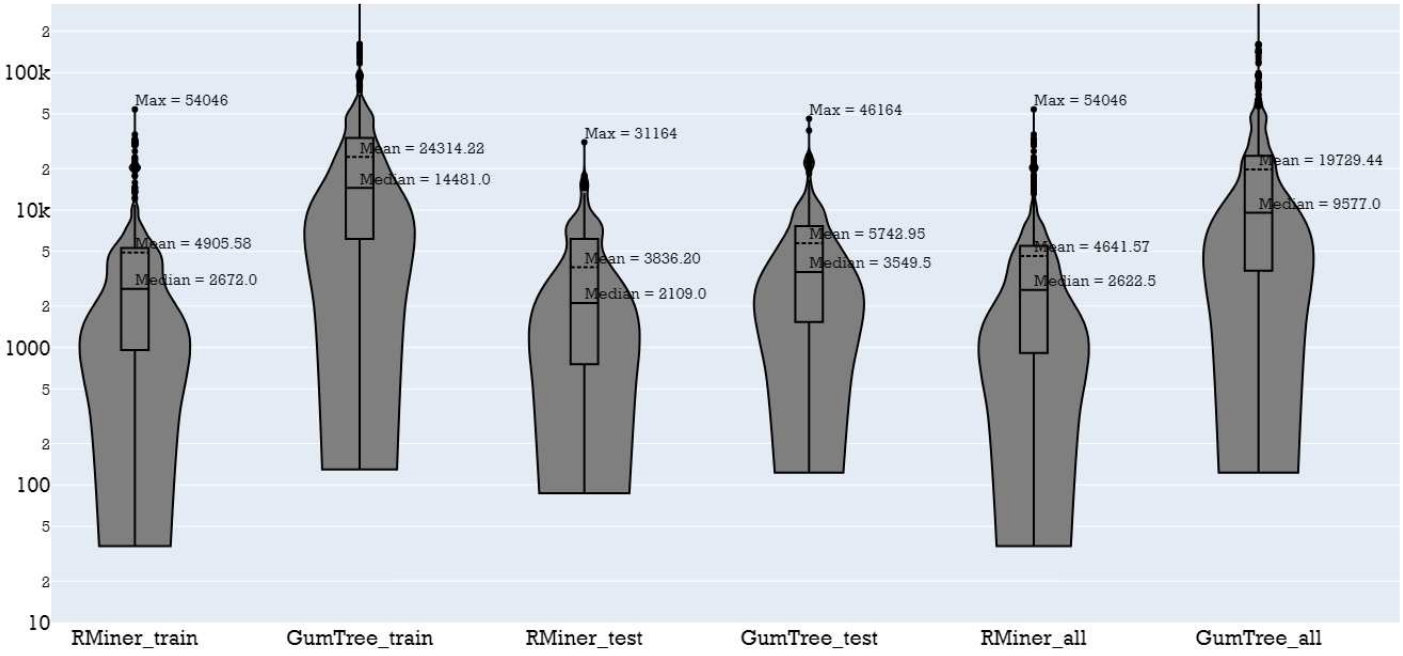


Figure 6.10: Violin plot showing the execution time of RMiner versus GumTree BlockTracker

Figure 6.10 shows violin plots [64] depicting the execution time of CodeTracker for tracking the

entire change history of each block in the oracle. The y-axis is on a logarithmic scale and the units are all in milliseconds. Each tool was executed separately on the same machine with the following specifications: AMD Ryzen 7 5800H CPU @ 3.20GHz × 8, 16 GB 3200 MHz DDR4, 512 GB PCIe SSD, Windows 11 Home operating system, and Java 11.0.15 x64 with a maximum of 8GB Java heap memory (i.e., `-Xmx8g`). All 20 project repositories used in the oracle were locally cloned before running the tool.

We measure execution time as the time taken by the tool for tracking a block to its introduction in its commit history. We also include the time taken by RefactoringMiner to parse the code and generate refactoring information and statement mappings. From our experiments, CodeTracker processes the entire commit change history of a block in an overall median of 2622.50 ms and an overall average of 4641.57 ms. These fast execution times are achieved due to the heuristics implemented by Jodavi and Tsantalis [1] for RefactoringMiner that allow for granular analysis of the source code as and when required, with many instances not requiring the processing of refactoring information at all.

We shall now discuss the execution time of GumTree BlockTracker and compare it to RefactoringMiner BlockTracker. As described in Section 5.6, we provide the MOVE METHOD and SPLIT CLASS refactoring information to GumTree BlockTracker via RefactoringMiner to evaluate GumTree's statement mapping accuracy most fairly. However, computing possible refactorings for each commit alongside generating statement mappings with GumTree becomes computationally expensive, and results in a significant increase in the time required to process a commit. For this experiment, we decided to adopt a memoized approach to obtain refactoring information and point GumTree to the right file in each commit.

The memoization approach is as follows: we integrated RefactoringMiner into GumTree BlockTracker to check for a potential MOVE METHOD or SPLIT CLASS refactoring for each commit using the class `GitHistoryRefactoringMiner`, which provides a method `detectAtCommit`. This method takes in as parameters a repository, commit ID, and an anonymous class extending the `RefactoringHandler` abstract class. Overriding the handle method in this class, we obtain the commit ID and the list of refactorings occurring within the commit. Here, we note all the MOVE

OPERATION refactoring and SPLIT CLASS refactoring occurrences. In commits where such a refactoring occurs, we obtain the new file path of where the method was moved to and store it in a cache in memory, with a unique key for the cache entry based on the method signature and commit ID. After processing all the blocks from our oracle, we then serialize this cache into a JSON file. After the cache file has been created, we no longer interact with RefactoringMiner and instead deserialize the cache before the execution of the oracle and make hits to it to discover method moves and their new file paths in constant time.

It is only after this that we measured the execution time of GumTree BlockTracker and compared it to RefactoringMiner BlockTracker. Hence, we can also inadvertently make a comparison of the execution time of RefactoringMiner versus GumTree with respect to statement mapping generation.

From Figure 6.10, we can make a clear observation that RefactoringMiner-based BlockTracker has an average execution time of 4641.57 ms, making it x4.25 times faster than GumTree-based BlockTracker, which has an average execution time of 19729.44 ms. Moreover, RefactoringMiner-based BlockTracker has a median execution time of 2622.50 ms, making it x3.65 times faster than GumTree-based BlockTracker, which has a median execution time of 9577.00 ms. Both findings confirm that RefactoringMiner-based BlockTracker has a considerably faster execution time (3-4 times faster) over GumTree-based BlockTracker.

Chapter 7

Limitations and Threats to Validity

In this chapter, we discuss the limitations of our approach in section 7.1. We shall then look into the internal validity threats in section 7.2 and devote section 7.3 to external validity threats. Finally, we discuss the verifiability of our tools and experiments in section 7.4.

7.1 Limitations

As highlighted by Jodavi and Tsantalis [1], a major limitation of CodeTracker is the fact that, in its current state, it can only support Java code bases. Since the limitation stems from the dependency of CodeTracker utilizing RefactoringMiner for static code analysis, CodeTracker will not be able to support additional languages until RefactoringMiner does the same.

A limitation of CodeTracker Visualizer is that it is built upon the GitHub user interface, and although easily extendable, it currently does not support code change history tracking for other online version control platforms like GitLab and BitBucket.

7.2 Internal Validity

The utilization of a self-made oracle for our evaluations could be a major threat to our internal validity. To minimize the potential of this threat, we utilized the updated and extended oracle presented by Jodavi and Tsantalis [1], which was verified for two person-months. This oracle was

originally built by Grund et al. [2] and underwent 100 hours of manual validation as well. The validations performed by two independent research groups make this oracle extremely reliable. Building upon this, we extended the oracle to include enhanced change history information for existing oracle instances of methods, attributes, and variables. We also discovered and fixed discrepancies found in the oracle, as discussed in Section 6.1. We provide an extended version of this oracle with the introduction of the change histories of 1280 code blocks, building upon the method history oracle, which can be assumed as being completely accurate.

We have also utilized an oracle validation tool, as discussed in Section 4.5, to manually verify each commit change history indicated in the entire oracle (methods, attributes, variables, blocks). Since no other oracles for block change history exist, we compared the block oracle against statement mappings generated by GumTree, to have a secondary source of truth for our oracle's validity. This process has been discussed in section 6.3. Overall, We dedicated three person-months to perform multiple rounds of these validations.

7.3 External Validity

Jodavi and Tsantalis [1] showed that CodeTracker performs with an overall precision and recall of 99.97% and 99.97% on the oracle presented by them.

To build the block oracle, we utilized this base oracle as it is considered as being highly reliable. To speed up and minimize the overall need for validation and adopt the semi-automated approach discussed in Section 6.1, the use of this oracle was crucial. However, since CodeTracker was built on this oracle, its design is familiar with the patterns that occur in these repositories and most of them have already been accounted for. Utilizing this tool to track a code element in a repository not seen before may produce inaccurate results, affecting our generalizability. However, before exploring new repositories and adopting them to manually generate an oracle, we wanted to utilize a trusted oracle to first focus on building a resilient base infrastructure for BlockTracker and verifying the accuracy of code block statement mappings being returned by RefactoringMiner.

7.4 Verifiability

We make publicly available the source code [6] for the extended version of CodeTracker with block tracking capabilities along with the extended oracle with block change history. We also make available the source code for CodeTracker Visualizer [57] and CodeTracker REST API [47]. Finally, our implementation of GumTree BlockTracker is also present in the CodeTracker repository in its branch [58]. We hope our contributions will improve the efficiency of code reviews and enable seamless access to the powerful APIs provided by CodeTracker and assist future research and development on code element history tracking.

Chapter 8

Conclusion and Future Work

CodeTracker provides a streamlined approach to the tracking of code element change history with reliable accuracy. It can track method, attribute, and variable change histories to their introduction commits. CodeTracker provides superior tracking abilities to line-based history detection tools due to its refactoring-aware nature. We present the addition of control flow code block tracking to CodeTracker, to allow for further fine-grained tracking of code elements. Along with that, we present the addition of a block oracle that details code change histories for all blocks present in the Grund et al. [2] method oracle's start methods. We also developed CodeTracker Visualizer, a code change history visualization and navigation tool that enhances the code review process while allowing for easy access to CodeTracker's APIs. It does so by visually overlaying change history information onto the GitHub user interface. We also built a fork of this tool to aid in the process of oracle validation. Finally, we perform a direct comparison of statement mapping accuracy for RefactoringMiner versus GumTree by building two versions of BlockTracker that obtain statement mappings from each tool respectively.

In summary, the main conclusions and lessons learned are as follows:

(1) CodeTracker has enhanced granularity by being extended to track code blocks and performs with high accuracy with a precision of 98.12% and recall of 97.62%.

(2) CodeTracker Visualizer enables code element tracking from the browser and visually presents the changes that occurred on a code element with direct navigational capabilities to each

specific commit, significantly speeding up the time required to track a code element's change history. It also provides an extended version that can aid in oracle validations.

(3) We extend the oracle presented by Jodavi and Tsantalis [1] to include 1280 new instances of blocks and their change histories.

(4) RefactoringMiner provides statement mappings x4.25 times faster than GumTree. In terms of accuracy, it has an average increment in precision and recall of +6.9% and +16.24%, respectively.

8.1 Potential Applications

Apart from providing developers with fine-grained code change history, CodeTracker's block-tracking capabilities can be utilized in many different research areas.

(1) **Bug inducing commit analysis:** Given a code block containing a bug, identify when and by whom the bug was introduced. CodeTracker can be used to track the change history of the code block and greatly trim down the number of commits to be analyzed. Paired with CodeTracker Visualizer, navigating between the changes occurring on a code block becomes instantaneous.

(2) **Evolution analysis of duplicates:** Given a class with duplicate code blocks, understand and analyze the evolution it has gone through. With the enhancement of Refactoring Miner to now support *multi-mappings*, CodeTracker can track all duplicates and generate an evolution chain for each instance of the code block, helping researchers understand the evolution of software systems.

(3) **Mining Software Repositories (MSR) applications:** Given different control structures, is one type more prone to changes than another type? For example, does a `switch` block change more than an `if-else-if` ladder? What about a traditional loop versus a Java `Stream API`? Another area to explore could be given different languages (e.g. Java and C#), is a control structure in one language more susceptible to changes when compared to the other? These are all questions that CodeTracker can help answer.

8.2 Future work

The primary motive behind this thesis was to enhance the granularity of code element tracking provided by CodeTracker, and we achieved this by extending CodeTracker to be able to track code blocks. A clear next step at this point would be to extend the granularity to track a single statement. This would make CodeTracker completely granular in the way it allows tracking. Support for code block mapping to conditional statements could also be explored.

Another next step would be to explore new repositories that are not a part of the current oracle to discover and account for patterns that CodeTracker may not already consider. An additional benefit of this would be the extension of the oracle to contain more instances. The oracle in its current state provides a solid base to test future code change history detection tools against, however, it can be extended to include more instances of each code element type.

A potential aspect that could be considered a good next step would be the extension of CodeTracker to support more languages. A good starting point could be Kotlin, since it is closely related to Java, and work on extending RefactoringMiner for Kotlin [65] is already ongoing. This can further be extended to languages like Python, which also has community-built versions of RefactoringMiner [66, 67].

Finally, a great next step for CodeTracker Visualizer would be to extend the application to support integration with GitLab and BitBucket, which are two other popular version control platforms in the industry [16]. Integration with more platforms will allow for easy multi-platform usage and provide greater flexibility to its users.

Bibliography

[1] Mehran Jodavi and Nikolaos Tsantalis. Accurate method and variable tracking in commit history. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 183–195, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394130. doi: 10.1145/3540250.3549079. URL https://doi.org/10.1145/3540250.3549079.

[2] Felix Grund, Shaiful Chowdhury, Nick C. Bradley, Braxton Hall, and Reid Holmes. Codeshovel: Constructing method-level source code histories. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 1510–1522. IEEE Press, 2021. ISBN 9781450390859. doi: 10.1109/ICSE43902.2021.00135. URL https://doi.org/10.1109/ICSE43902.2021.00135.

[3] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 483–494, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180206. URL http://doi.acm.org/10.1145/3180155.3180206.

[4] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 48(3):930–950, 2022. doi: 10.1109/TSE.2020.3007722.

[5] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference*

on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014, pages 313–324, 2014. doi: 10.1145/2642937.2642982. URL http://doi.acm.org/10.1145/2642937.2642982.

[6] Jodavi Mehran, Nikolaos Tsantalis, and Mohammed Tayeeb Hasan. Codetracker github repository, 2023. URL https://github.com/flozender/code-tracker.

[7] Ruslan Diachenko and Roman Ivanov. Checkstyle, 2015. URL https://github.com/checkstyle/checkstyle/commit/3ef918920.

[8] Danny van Bruggen. Javaparser, 2018. URL https://github.com/javaparser/javaparser/commit/ee729dcafe.

[9] Ivan Sopov. Checkstyle, 2014. URL https://github.com/checkstyle/checkstyle/commit/f1efb2767.

[10] Andrei Selkin. Checkstyle, 2016. URL https://github.com/checkstyle/checkstyle/commit/bf69cf167.

[11] Stephan Ewen. Apache flink, 2017. URL https://github.com/apache/flink/commit/536675b03.

[12] Oleg Sukhodolsky. Checkstyle, 2003. URL https://github.com/checkstyle/checkstyle/commit/cd89321522d9bf7fc10547e743fb8bbb4c993791#diff-fea9f91af0be0914b80d0451274454c2dbf87da35662aa256201ddb897d08c81R618.

[13] Roman Ivanov. Checkstyle, 2015. URL https://github.com/checkstyle/checkstyle/commit/1a2c318e2.

[14] Jan Bartel. Jetty.project, 2013. URL https://github.com/eclipse/jetty.project/commit/9c168866ffbb349d56501d11801f0418bdee3596#diff-cf3ebfaa4e05c6197de38fb2e55da070503cf81220a7eb52e29570421d3f283dR319.

[15] Wikipedia. Block (programming), 2023. URL https://en.wikipedia.org/w/index.php?title=Block_(programming)&oldid=1156930578.

[16] Stack Overflow. Stack overflow developer survey, 2022. URL https://survey.stackoverflow.co/2022.

[17] Katsuhisa Maruyama, Eijiro Kitsu, Takayuki Omori, and Shinpei Hayashi. Slicing and replaying code change history. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE '12, page 246–249, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312042. doi: 10.1145/2351676.2351713. URL https://doi.org/10.1145/2351676.2351713.

[18] Mihai Codoban, Sruti Srinivasa Ragavan, Danny Dig, and Brian Bailey. Software history under the lens: A study on why and how developers examine it. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–10, 2015. doi: 10.1109/ICSM.2015.7332446.

[19] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 492–501, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933751. doi: 10.1145/1134285.1134355. URL https://doi.org/10.1145/1134285.1134355.

[20] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. Code review quality: How developers see it. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1028–1038, 2016. doi: 10.1145/2884781.2884840.

[21] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig. Is it dangerous to use version control histories to study source code evolution? In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pages 79–103, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31057-7.

[22] Sitaram Chamarty, Hiren D. Patel, and Mahesh V. Tripunitara. An authorization scheme for version control systems. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*, SACMAT '11, page 123–132, New York, NY, USA, 2011. Association

for Computing Machinery. ISBN 9781450306881. doi: 10.1145/1998441.1998460. URL https://doi.org/10.1145/1998441.1998460.

[23] Tammo Freese. Refactoring-aware version control. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 953–956, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933751. doi: 10.1145/1134285.1134461. URL https://doi.org/10.1145/1134285.1134461.

[24] Thomas Zimmermann. Fine-grained processing of cvs archives with apfel. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology EXchange*, eclipse '06, page 16–20, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595936211. doi: 10.1145/1188835.1188839. URL https://doi.org/10.1145/1188835.1188839.

[25] YoungSeok Yoon, Brad A. Myers, and Sebon Koo. Visualization of fine-grained code change history. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, pages 119–126, Sep. 2013. doi: 10.1109/VLHCC.2013.6645254.

[26] Francisco Servant and James A. Jones. History slicing: Assisting code-evolution tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316149. doi: 10.1145/2393596.2393646. URL https://doi.org/10.1145/2393596.2393646.

[27] Yoshiki Higo, Shinpei Hayashi, and Shinji Kusumoto. On tracking java methods with git mechanisms. *Journal of Systems and Software*, 165:110571, jul 2020. doi: 10.1016/j.jss.2020.110571. URL https://doi.org/10.1016%2Fj.jss.2020.110571.

[28] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Historage: Fine-grained version control system for java. pages 96–100, 09 2011. doi: 10.1145/2024445.2024463.

[29] Sunghun Kim, Kai Pan, and E.J. Whitehead. When functions change their names: automatic detection of origin relationships. In *12th Working Conference on Reverse Engineering (WCRE'05)*, pages 10 pp.–152, 2005. doi: 10.1109/WCRE.2005.33.

[30] M.W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005. doi: 10.1109/TSE.2005.28.

[31] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Incremental origin analysis of source code files. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 42–51, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328630. doi: 10.1145/2597073.2597111. URL https://doi.org/10.1145/2597073.2597111.

[32] Andre Hora, Danilo Silva, Marco Tulio Valente, and Romain Robbes. Assessing the threat of untracked changes in software evolution. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 1102–1113, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3180212. URL https://doi.org/10.1145/3180155.3180212.

[33] Thomas D. LaToza and Brad A. Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450305471. doi: 10.1145/1937117.1937125. URL https://doi.org/10.1145/1937117.1937125.

[34] Reid Holmes and Andrew Begel. Deep intellisense: A tool for rehydrating evaporated information. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, page 23–26, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580241. doi: 10.1145/1370750.1370755. URL https://doi.org/10.1145/1370750.1370755.

[35] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 175–184, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605587196. doi: 10.1145/1806799.1806828. URL https://doi.org/10.1145/1806799.1806828.

[36] Amy J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, page 344–353, USA, 2007. IEEE Computer Society. ISBN 0769528287. doi: 10.1109/ICSE.2007.45. URL https://doi.org/10.1109/ICSE.2007.45.

[37] Kaifeng Huang, Bihuan Chen, Xin Peng, Daihong Zhou, Ying Wang, Yang Liu, and Wenyun Zhao. Cldiff: Generating concise linked code differences. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 679–690, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359375. doi: 10.1145/3238147.3238219. URL https://doi.org/10.1145/3238147.3238219.

[38] Yun Young Lee, Darko Marinov, and Ralph E. Johnson. Tempura: Temporal dimension for ides. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 212–222, May 2015. doi: 10.1109/ICSE.2015.42.

[39] Xi Ge, Saurabh Sarkar, and Emerson Murphy-Hill. Towards refactoring-aware code review. 06 2014. doi: 10.1145/2593702.2593706.

[40] Everton L. G. Alves, Myoungkyu Song, and Miryung Kim. 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*, FSE 2014, page 751–754, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2661674. URL https://doi.org/10.1145/2635868.2661674.

[41] Rodrigo Brito and Marco Tulio Valente. Raid: Tool support for refactoring-aware code reviews, 2021. URL https://doi.org/10.48550/arXiv.2103.11453.

[42] Gary Gregory. Apache commons-lang, 2018. URL https://github.com/apache/commons-lang/blob/a36c903d4f1065bc59f5e6d2bb0f9d92a5e71d83/src/main/java/org/apache/commons/lang3/LocaleUtils.java#L114.

[43] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 858–870, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2950305. URL https://doi.org/10.1145/2950290.2950305.

[44] Emerson Murphy-Hill, Chris Parnin, and Andrew Black. How we refactor, and how we know it. volume 38, 05 2009. doi: 10.1109/ICSE.2009.5070529.

[45] Till Rohrmann and Stephan Ewen. Apache flink, 2016. URL https://github.com/apache/flink/commit/72b295b3b52dff2d0bc5b78881826e8936c370ff#diff-2c7004a2d412c3566de5ff6fb9e6d027742328c2acad60685e47ce4ba9df0810R641.

[46] Martin Fowler. Fluent interface, 2023. URL https://martinfowler.com/bliki/FluentInterface.html.

[47] Mohammed Tayeeb Hasan and Nikolaos Tsantalis. Codetracker rest api github repository, 2023. URL https://github.com/flozender/codetracker-api.

[48] JBoss and Flavia Rainone. Jboss undertow, 2023. URL https://undertow.io.

[49] FasterXML. Fasterxml jackson, 2023. URL https://github.com/FasterXML/jackson.

[50] Eclipse. Eclipse jgit, 2023. URL https://projects.eclipse.org/projects/technology.jgit.

[51] Kohsuke Kawaguchi. Github api for java, 2023. URL https://github-api.kohsuke.org/.

[52] ovity. Octotree, 2023. URL https://github.com/ovity/octotree.

[53] Gulpjs. Gulp, 2023. URL https://gulpjs.com/.

[54] Observable. D3.js, 2023. URL https://d3js.org.

[55] Jestjs. Jest, 2023. URL https://github.com/jestjs/jest.

[56] Argos CI. Jest-puppeteer, 2023. URL https://github.com/argos-ci/jest-puppeteer.

[57] Mohammed Tayeeb Hasan and Nikolaos Tsantalis. Codetracker visualizer github repository, 2023. URL https://github.com/flozender/codetracker-extension.

[58] Mohammed Tayeeb Hasan and Nikolaos Tsantalis. Codetracker gumtree version, 2023. URL https://github.com/flozender/code-tracker/tree/add-gumtree-mappings.

[59] Stephen Colebourne. Apache commons-lang, 2003. URL https://github.com/apache/commons-lang/commit/73ee6c3d2.

[60] Akira Fujimoto, Yoshiki Higo, Junnosuke Matsumoto, and Shinji Kusumoto. Staged tree matching for detecting code move across files. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, page 396–400, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379588. doi: 10.1145/3387904.3389289. URL https://doi.org/10.1145/3387904.3389289.

[61] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007. doi: 10.1109/TSE.2007.70731.

[62] Pascal Schumacher. Apache commons-lang, 2018. URL https://github.com/apache/commons-lang/commit/2e9f3a801.

[63] Henri Yandell. Apache commons-lang, 2006. URL https://github.com/apache/commons-lang/commit/4f514d5eb.

[64] Jerry L. Hintze and Ray D. Nelson. Violin plots: A box plot-density trace synergism. *The American Statistician*, 52(2):181–184, 1998. ISSN 00031305. URL http://www.jstor.org/stable/2685478.

[65] Zarina Kurbatova and JetBrains Research. Kotlinrminer, 2023. URL https://github.com/JetBrains-Research/kotlinRMiner.

[66] Hassan Atwi, Bin Lin, Nikolaos Tsantalis, Yutaro Kashiwa, Yasutaka Kamei, Naoyasu Ubayashi, Gabriele Bavota, and Michele Lanza. Pyref: Refactoring detection in python projects. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 136–141, 2021. doi: 10.1109/SCAM52516.2021.00025.

[67] Malinda Dilhara, Ameya Ketkar, Nikhith Sannidhi, and Danny Dig. Discovering repetitive code changes in python ml systems. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, New York, NY, USA, 2022. ACM. ISBN 978-1-4503-9221-1/22/05. doi: 10.1145/3510003.3510225. URL http://doi.acm.org/10.1145/3510003.3510225.