Accurate Program Element Tracking in Commit History

Mehran Jodavi

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 (Software Engineering) at
Concordia University
Montréal, Québec, Canada

November 2021

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Mehran Jodavi**
Entitled: **Accurate Program Element Tracking in Commit History**

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

Master of Computer Science (Software Engineering)

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

Signed by the final examining committee:

_____ Chair
Dr. Joey Paquet

_____ External

_____ Examiner
Dr. Tse-Hsun (Peter) Chen

_____ Examiner
Dr. Joey Paquet

_____ Thesis Supervisor
Dr. Nikolaos Tsantalis

Approved by _____
Dr. Leila Kosseim, Graduate Program Director

November 26, 2021 _____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Accurate Program Element Tracking in Commit History

Mehran Jodavi

Tracking program elements in the commit history of a project is essential for supporting various software maintenance, comprehension and evolution tasks. Accuracy is of paramount importance for the adoption of program element tracking tools by developers and researchers. To this end, we propose CodeTracker, a refactoring-aware tool that can generate the commit change history for method and variable declarations with a very high accuracy. More specifically, CodeTracker has 99.9% precision and recall in method tracking, surpassing the previous state-of-the-art tool, CodeShovel, with a comparable execution time. CodeTracker is the first tool of its kind that can track the change history of variables with 96.7% precision and 95.5% recall. To evaluate its accuracy in variable tracking, we extended the oracle created by Grund et al. for the evaluation of CodeShovel, with the complete change history of all 1345 variables and parameters declared in the 200 methods comprising the Grund et al. oracle. We make our tool and extended oracle publicly available to enable the replication of our experiments and facilitate future research on program element tracking techniques.

Acknowledgments

I want to express my special appreciation and thanks to my fantastic supervisor Dr. Nikolaos Tsantalis for taking me on as a master's student, allowing me to pursue my profession and dreams, and believing in me. This work could not have been possible without his involvement, support, dedication and enthusiasm. His consistent reminders that only impossible is impossible have always helped me and have positively changed my attitude toward life.

I would like to acknowledge to my committee members – Dr. Tse-Hsun (Peter) Chen and Dr. Joey Paquet – thank you for helpful critiques, words of encouragement and your precious time for reviewing this thesis.

I would also like to thank my wife, Zohreh, for always being by my side and my best friend and for all the support, compassion, understanding, patience, and love.

Finally, I would like to thank my Mom and Dad for their ongoing support from far and wide, my sister, Marjan, for being the best sister, my lovely nephew, Nikan, for remembering me although he was only three years old when he last saw me, and my brother-in-law, Mohsen, for being there for my family instead of me while I was miles away. I miss them all a lot and hope we can get together soon after a while.

Contents

List of Figures **vii**

List of Tables **viii**

1 Introduction **1**

2 Motivation and Background **4**

2.1 Motivation 4

2.2 Limitations of Current Tracking Tools 6

3 Related Work **9**

4 Research Approach **13**

4.1 Program Element Identifier 13

4.2 Tracking Process 16

4.3 Change Graph Evolution Hooks 27

5 Implementation **30**

5.1 Languages and Platforms 30

5.2 CodeTracker API 31

5.3 Code Element Types 33

6 Evaluation **35**

6.1 Oracle Update and Extension 35

6.2 Method Tracking Accuracy 40

6.3 Variable Tracking Accuracy 42

6.4 Class Tracking Accuracy 43

6.5 Field Tracking Accuracy 44

6.6 Execution Time 45

7 Limitations and Threats to Validity **48**

7.1 Language Specificity 48

7.2 Internal Validity 48

7.3 External Validity 49

7.4 Verifiability 50

8 Conclusions and Future Work **51**

Bibliography **53**

List of Figures

1 Complex code evolution scenario in project Netflix Zuul. 4

2 Typical container structure in Java programs 14

3 Hierarchy of supported change kinds for methods adopted by CodeShovel [13]. Newly supported change kinds are highlighted in darker background colour and blue text color. 17

4 Hierarchy of supported change kinds for variables. 19

5 Hierarchy of supported change kinds for fields. 20

6 Hierarchy of supported change kinds for fields. 20

7 Tracking of a program element extracted from the tracked element. 28

8 Tracking of a program element inlined to the tracked element. 28

9 Tracking of a program element from which the tracked element is extracted. 29

10 CodeTracker API 32

11 History Interface and its dependencies 34

12 Code element Types 34

13 Execution time in milliseconds of CodeTracker and CodeShovel on training and testing sets. 45

List of Tables

1 Updates in the oracle created by Grund et al. [13] 37

2 Number of instances per change kind for variables. 38

3 Number of instances per change kind for classes. 39

4 Number of instances per change kind for fields. 40

5 Method tracking precision/recall at commit level 41

6 Method tracking precision/recall at change level 42

7 Variable tracking precision/recall for CodeTracker 43

8 Class tracking precision/recall for CodeTracker 44

9 Field tracking precision/recall for CodeTracker 44

10 Average percentage of commits processed in each step of the tracking process. 46

Chapter 1

Introduction

Developers routinely track code snippets in the commit history to facilitate various software engineering tasks. Codoban et al. [7] surveyed 217 developers to find the motivations behind examining software history. The most common reasons are to a) recover the rationale behind a snippet of code, b) find the commits that introduced a bug, c) find who are the knowledgeable peers on certain modules and patterns, d) reverse engineer requirements from code, e) keep up with how the code state evolves, f) apply changes from other branches into the main branch. The surveyed developers also expressed some challenges with the usability of existing tools, such as their inability to detect file moves and renames, and their configuration (e.g., setting up `git bisect` to find the commit that introduced a bug). Grund et al. [13] conducted a survey with 42 professional software developers and found that they prefer source code history information at the method/function and class level rather than the file and block level. Moreover, the tools used by the developers to inspect code history, such as `git log` and IntelliJ's history feature, are unable to find the commit that actually introduced a method and deal with complex structural changes (e.g., method moves).

Accurate code snippet tracking is also essential in many areas of software engineering research. Alencar da Costa et al. [10] pointed out that bug-inducing analysis algorithms (e.g.,

SZZ [37, 20, 43]) suffer from broken historical links due to file moves and renames. This further affects the results of defect prediction techniques and empirical studies investigating the characteristics of bug-introducing changes, which rely on the original SZZ algorithm or its variants [33]. Shen et al. [34] showed that automatic source code merging tools often fail to track the changed program elements correctly due to overlapping refactoring operations, and thus are unable to perform the auto-merging. The automatic migration of client software to newer library and framework versions, requires to track the updated API program elements (i.e., methods and fields) from the source to the target version, extract changes in the API signatures, and adapt accordingly the API references in client's code [11, 18, 9]. API program element tracking has been performed both at commit level [6, 5] and release level [24, 23]. However, fine-grained program element tracking at commit level may be more accurate than release level, as comparing directly two releases involves significantly more noise from overlapping changes performed in all commits between the two releases.

The inherent limitations of the line-based text diff and blame tools used in the aforementioned software engineering tasks, motivated researchers to develop techniques for tracking more accurately program elements, such as methods/functions and classes, in the commit history of software projects [13, 14, 15, 39, 12, 38, 16]. These techniques deal with changes that modify the name/signature or location of a program element and can cause a split in its history. Hora et al. [16] found that 25% of classes and methods have at least one *untracked* change (i.e., move, rename, extract, inline refactoring) in their histories. Despite the significant accuracy improvements brought by program element tracking tools, they still have some limitations, which we discuss in the next section.

Our solution offers some significant improvements over the previous state-of-the-art and novel contributions:

1. We fix all inaccuracies that we found in the oracle provided by Grund et al. [13] including the evolution history of 200 methods. Moreover, we extend this oracle by adding the

evolution history of 1345 variables declared in these methods, 165 classes containing these methods, and 806 fields declared in the classes containing these methods.

2. We support new kinds of program element changes, such as documentation and annotation changes, which are not supported by CodeShovel [13] and other tools.

3. We support a more fine-grained change reporting relative to CodeShovel [13] and other tools.

4. We improve both precision and recall in method evolution tracking over the previous state-of-the-art, CodeShovel [13].

5. We extend RefactoringMiner [42, 41] with heuristics for performing partial commit analysis in order to reduce the execution time. We show that the applied heuristics achieve an execution time comparable to that of CodeShovel [13] without jeopardizing precision or recall.

6. We are the first to support the evolution tracking of all code elements like variable declarations with over 97% precision and recall, class declaration with 100% precision and recall, and field declaration with over 98% precision and recall.

The remainder of this document is structured as follows: Illustration of a complex code evolution scenario and providing background information about existing tooling and their limitation are provided in Chapter 2. Chapter 3 is dedicated to Related Works in the context of tools tried to detect source code history. Chapter 4 presents our approach for modelling and reconstructing the changes applied on program elements, such as method and variable declarations, in the commit history of a project. Chapter 5 presents details about the actual implementation of our approach as a tool. Chapter 6 shows the results of our experimental evaluation, and information on the oracle construction. Discussion about limitations and threats to validity follows in Chapter 7 before we conclude in Chapter 8.

Chapter 2

Motivation and Background

This chapter motivates the importance of accurate tracking of program elements in the commit history of software projects. We first illustrate a complex code evolution scenario in Section 2.1. We then provide background information about existing tooling and discuss how they are insufficient to track the evolution of code elements in Section 2.2.

2.1 Motivation

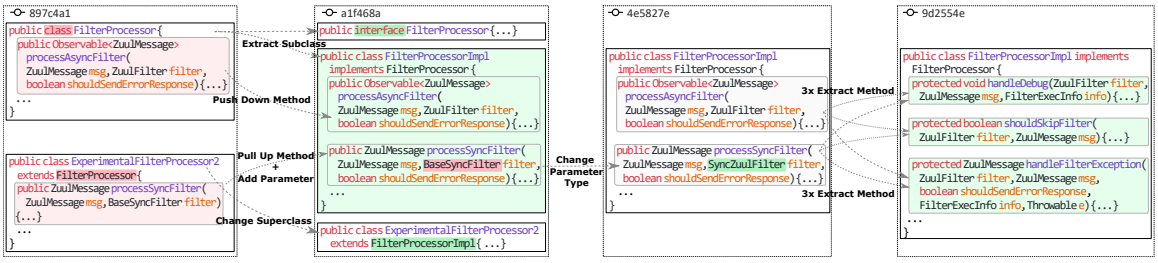

Figure 1: Complex code evolution scenario in project Netflix Zuul.

Code evolution can be very complex as it may involve refactoring operations changing the structure of the project, which overlap with changes from other software maintenance activities, such as feature additions and bug fixes. Negara et al. [31] found that 46% of refactored program entities are also edited or further refactored in the same commit, a

practice commonly referred to as *floss refactoring* [25]. This makes the accurate tracking of program elements in the commit history a rather challenging task. To motivate our work, we present in Figure 1, a complex code evolution scenario detected in project Netflix Zuul by our tool.

In commit `a1f468a` a new subclass named `FilterProcessorImpl` was extracted from class `FilterProcessor`, which was then declared as an interface. Method `processAsyncFilter` was among the methods pushed down from `FilterProcessor`, while method `processSyncFilter` was moved from class `ExperimentalFilterProcessor2`, which eventually extended the newly extracted class `FilterProcessorImpl`. The moved method `processSyncFilter` was also slightly modified by adding a new `boolean` parameter `shouldSendErrorResponse` along with some functionality related to the newly added parameter. In commit `4e5827e`, the type of parameter `filter` changed from `BaseSyncFilter` to `SyncZuulFilter` in method `processSyncFilter`. Finally, in commit `9d2554e` three duplicated code fragments were extracted from `processSyncFilter` and `processAsyncFilter` in methods `handleFilterException`, `handleDebug`, and `shouldSkipFilter`, respectively.

Our tool can properly track these extracted code fragments back to their origin methods and classes by creating a branch for each extracted method on the evolution graphs of the origin methods. However, none of the competitive method tracking tools we experimented with was able to track these extracted methods. CodeShovel [13] and FinerGit [15] report `handleFilterException`, `handleDebug`, and `shouldSkipFilter` as newly added methods and miss their links with the origin methods from which they were extracted. Assuming there exists a bug in one of the extracted methods, which was introduced prior to commit `897c4a1`, CodeShovel [13] and FinerGit [15] would not be able to find the commit(s) and author(s) that introduced this bug, and more importantly would not be able to find that this bug was duplicated in two different origin files and methods.

2.2 Limitations of Current Tracking Tools

CodeShovel [13], is the most accurate tool for uncovering Java method histories to-date, as it produces complete and accurate commit change histories for 90% of methods, including 97% of all method changes. However, our experiments have shown that it fails to track properly methods from which a significant part of their body has been extracted to new methods, as it uses a 75% body similarity threshold to match modified methods, and thus erroneously matches the original method with the extracted one. The same limitation holds when methods with a relatively large body are inlined to the tracked methods. Our approach overcomes this limitation by being fully refactoring-aware and detecting method extractions/inlines from/to the tracked methods.

FinerGit [15] and Historage [14] create a finer-grained Git repository. FinerGit improves on the limited capability of Historage to track renamed or moved methods. Pre-processing an entire repository to place each method in its own file, is computationally expensive and requires additional hard disk space, which can be prohibitive, especially for large repositories with many files and a long commit history. As a matter of fact, Grund et al. [13] found that FinerGit ran out of memory or did not finish pre-processing within 15 minutes for the four largest repositories in their validation data set. Moreover, this pre-processing cost did not contribute an accuracy improvement, as the recall of FinerGit was 65% compared to 90% of CodeShovel [13].

Kim et al. [39] proposed an approach to identify function mappings across revisions even when a function's name changes. The computation of text diff and the execution of multiple clone detection tools may have a considerable cost, especially when there are many combinations of deleted and newly added functions to be compared.

A common limitation of all aforementioned tools is that they are designed to support only the tracking of methods, and cannot be extended to support the tracking of other program elements, such as variables and attributes, whose evolution is also interesting for

the developers. Several studies have shown that developers frequently refactor variables and attributes, which makes their tracking in the commit history challenging. Negara et al. [29] found that RENAME LOCAL VARIABLE and RENAME FIELD are among the most popular refactorings applied by developers. Negara et al. [30] surveyed 420 developers, who ranked CHANGE FIELD TYPE as the most relevant and applicable transformation that they perform. Ketkar et al. [19] found that developers who changed the type of a variable or attribute, they also renamed it in 55% of the examined instances.

Godfrey and Zou [12] implemented a tool, named Beagle, that can detect structural changes like rename, move, split, and merge at function, file, and subsystem level. They rely on *origin analysis* to decide if a program entity is renamed or moved and a function call analysis to discover merges and splits of program entities. Although Beagle supports the tracking of program elements at different levels of granularity (i.e., function, file, subsystem), it requires as input two complete versions of a software system in order to extract static relations between program entities (e.g., function calls), and calculate various metrics. This makes Beagle impractical for program element tracking at commit level.

Steidl et al. [38] proposed an incremental origin analysis that applies some heuristics to find moved, renamed, split, and merged source code files. In contrast to Beagle, their approach is commit-based and incrementally reconstructs the history based on clone information and file name similarity. However, the proposed origin analysis is limited to files and thus does not support the tracking of program elements, such as methods, variables and attributes.

Hora et al. [16] introduced the concept of *change graph* to model the evolution of classes, methods, and their related changes in the commit history of a project, and study the phenomenon of *untracked* changes. In this graph, each class or method is represented as a node, while each tracked or untracked change is represented as an edge between two nodes. However, Hora et al.'s change graph is limited in modelling only the evolution of classes and

methods, supports a limited number of refactoring types (5 class-level and 6 method-level refactorings), and uses RefDiff [36] for the detection of refactoring operations, which has inferior precision, recall and performance than RefactoringMiner [42, 41]. Finally, the graph edges model only a small subset of refactoring operations, while other kinds of changes, such as method body and signature changes are omitted. Thus, Hora et al.'s change graph cannot be used to find all commits where a program element changed, i.e., the graph can provide only the commits in which a program element is involved in refactorings.

Summary: None of the currently available tools can track the evolution of fields and local variables. Moreover, the evolution tracking of methods is limited, as commonly applied refactoring types, such as EXTRACT METHOD, are not supported by the currently available tools. According to Negara et al. [29] and Tsantalis et al. [41], EXTRACT METHOD is the most commonly applied refactoring on methods.

Chapter 3

Related Work

In this chapter, we are presenting tools and techniques that tried to retrive and track the history of code elements.

Historical analysis of software repositories has been considered a critical way to collect code evolution and program understanding. As a result, in addition to the features for tracking the histories of lines, line ranges, and files provided by version control systems like Git, some tools have been built to help practitioners and researchers understand better the histories and track the evolution of software systems.

Grund et al introduced CodeShovel [13], the most accurate tool for uncovering Java method histories to-date, as it produces complete and accurate commit change histories for 90% of methods, including 97% of all method changes. CodeShovel is partially *refactoring-aware*. It supports tracking methods with changes in their signature (e.g., method rename, parameter addition/deletion), methods whose parent file has been moved/renamed, and methods moved to another file. CodeShovel takes a path to a Git repository on the local file system, the SHA of the commit to start from, a path of the file containing the method, name and line number of the method for which the history should be produced as input and produces a JSON file with a list of commits that changed the selected method as output. Each commit in the output includes a change type and has basic information like the author,

commit message, date, and more complex information like the time and number of commits between the current method changes and the previous one. It also holds type-specific information for more complex change types (e.g. the old file path and the new file path for a method move). CodeShovel's method tracking uses a similarity algorithm for matching methods across file versions. This method matching procedure computes the similarity of some metrics (e.g., string similarity of the method body, string similarity of the method name, string similarity of parameter names)

FinerGit [15] and Historage [14] create a finer-grained Git repository, in which each Java method exists in its own file, and take advantage of Git mechanisms to track changes on each individual method's corresponding file.

Historage keeps a fine-grained history of methods and constructors in Java, even when they have been renamed. It uses the rename/move detection mechanism of Git that identifies matches based on the similarities of file contents in case of rename and move. Historage stores all Java methods individually and control their histories. Since Historage is created on top of a Git version control system, every Git command can be used. They found that Historage can identify matches practically when renaming and moving exist by conducting empirical evaluation with some open source projects.

FinerGit improves on the limited capability of Historage to track renamed or moved methods, especially for small methods, by formatting each file to include a single token from the corresponding method in each line. This formatting makes Git's line-based similarity computation mechanism more robust in matching small methods, which have been renamed or moved. FineGit takes a Git repository of a Java project and makes another Git repository. In the new repository, every file with extension .mjava represents a Java method that gets extracted from the original files. The git-log command with option –follow for a .mjava file will return the history of the method that corresponds to that file. In order to measure the method tracking performance of FinerGit and compare it with Historage, an oracle

of 182 methods is manually constructed by the authors. Higo et al. applied FinerGit and Historage to the manually created oracle and found that FinerGit with scored 84.52% as maximum F-measure outperforms Historage with a score of 70.23%.They also confirmed that their technique worked well with methods of any size and showed that preprocessing of repositories, even for large repositories, took only a short time to construct finer-grained repositories.

Kim et al. [39] proposed an approach to identify function mappings across revisions even when a function's name changes. Their approach computes the similarity between functions based on the following weighted similarity factors: function name, incoming and outgoing calls, signature, function body text diff, complexity metrics and the results of two clone detection tools (CCFinder and MOSS). If the similarity of two functions is more significant than a predefined threshold, the approach identifies renamed functions. The authors sample 20% revision history from two open-source C projects and ask ten human judges to identify renamed entities to evaluate the approach manually. Evaluation result suggests that the accuracy of the approach in detecting renamings is 91

Godfrey and Zou [12] implemented a tool, named Beagle, that can detect structural changes like rename, move, split, and merge at function, file, and subsystem level. They rely on *origin analysis* that uses syntactic and semantic analyses to decide whether a program entity is newly introduced, renamed, moved, or a changed version of an original entity. They also used a function call analysis to discover merges and splits of program entities. They provide a list of potential origins of a target entity based on different matching techniques like name matching, declaration matching, metrics matching, and call relation matching.

Hora et al. [16] introduced the concept of *change graph* to model the evolution of classes, methods, and their related changes in the commit history of a project, and study the phenomenon of *untracked* changes during software development. The authors show that refactorings disable several tracking strategies to evaluate system evolution. In the *change*

graph that represents traceable changes or changes that split the element's history, each class or method is represented as a node, while each tracked change (i.e., code element that keep its names after a modification) or untracked change (i.e., code element that is renamed after a refactoring) is represented as edge between two nodes. They showed that 21% of the changes at the method level and up to 15% at the class level are untraceable.

Brito et al. [4] introduced a refactoring graph concept to assess refactoring operations over time. The authors analyzed 20 Java and JavaScript projects, extracted 1,525 refactoring subgraphs, and evaluated their properties: operations over time, size, commits, age, homogeneity, ownership, and patterns. They found out that nearly 30% of refactoring activities are part of a refactring subgraph over time, the majority of the refactoring subgraphs are small and mostly created by a single developer.

Chapter 4

Research Approach

In this chapter, we present our approach to creating a program element Identifier for code elements in section 4.1, then we describe our code element tracking process in section 4.2, and finally, we discuss change graph evolutions hooks in section 4.3.

4.1 Program Element Identifier

Each program element e is 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 e, CON_e is the signature of the container in which e belongs to, and SIG_e is the signature of e.

The typical container structure in Java programs is shown in the example of Figure 2. The container of a type declaration c is the tuple $CON_c = (SRC_c, PKG_c)$, where SRC_c is the source folder path and PKG_c is the package name in which c belong to. It is very important to include the source folder path in the container tuple, as it is possible to

```
Zull-core/src/main/java
    package com.netflix.zull;
        public class FilterProcessorImpl {
            protected final FilterLoader filterLoader;
            protected ZuulFilter getErrorEndpoint(ZuulMessage msg) {
                SessionContext context = msg.getContext();
                String endpointName = context.getErrorEndpoint();
                ...
                ZuulFilter errorEndpoint = getFilterByNameAndTyper(
                                    endpointName, FilterType.ENDPOINT);
                if(errorEndpoint == null) {
                    String errorStr = "..." + endpointName;
                    LOG.error("..."  + errorStr, context.getError());
                }
                return errorEndpoint;
            }
        }
```

Figure 2: Typical container structure in Java programs

have a type declaration with the same name and package in two different source folders. The container of a field declaration f is the tuple $CON_f = (CON_{C_f}, SIG_{C_f})$, where C_f is the type declaration in which f belong to, CON_{C_f} and SIG_{C_f} are the container and signature of C_f, respectively. The container of a method declaration m is the tuple $CON_m = (CON_{C_m}, SIG_{C_m})$, where C_m is the type declaration in which m belong to, CON_{C_m} and SIG_{C_m} are the container and signature of C_m, respectively. Finally, the container of a variable/parameter declaration v is the tuple $CON_v = (CON_{M_v}, SIG_{M_v})$, where M_v is the method declaration in which v is declared, CON_{M_v} and SIG_{M_v} are the container and signature of M_v, respectively.

The signature of a **type/enum/annotation** declaration c is the tuple

$$SIG_c = (N_c, K_c, S_c, I_c, T_c, A_c, V_c, M_c) \tag{2}$$

where N_c is the name of c, K_c is the kind of c, which is a categorical variable taking four possible values, namely *class*, *interface*, *enum* and *@interface* (for annotation type declarations), S_c is the super-class type of c, I_c is the list of super-interfaces of c, T_c is

the list of type parameters of c in the case that c is a parameterized type, A_c is the list of annotations of c, V_c is the visibility of c, which is a categorical variable taking four possible values (*public*, *protected*, *private*, or *package-private*), and finally M_c is the list of modifiers of c (*final*, *static*, *abstract*). All elements in the signature of c can change during its evolution, including its kind K_c, as happened with class `FilterProcessor` in the motivating example shown in Figure 1.

The signature of a **field** declaration f is the tuple

$$SIG_f = (N_f, T_f, A_f, V_f, M_f) \tag{3}$$

where N_f is the name of f, T_f is the type of f, A_f is the list of annotations of f, V_f is the visibility of f, and finally M_f is the list of modifier of f (*final*, *static*, *transient*, *volatile*).

The signature of a **method** declaration m is the tuple

$$SIG_m = (N_m, R_m, P_m, E_m, T_m, A_m, V_m, M_m, B_m, D_m) \tag{4}$$

where N_m is the name of m, R_m is the return type of m, P_m is the ordered parameter list of m, E_m is the list of thrown exception types of m, T_m is the list of type parameters of m in the case that m is a parameterized method, A_m is the list of annotations of m, V_m is the visibility of m, M_m is the list of modifiers of m (*final*, *static*, *abstract*, *synchronized*), B_m is the hashed value of m's body string representation, and finally D_m is the hashed value of m's Javadoc and inline comments.

The signature of a **variable/parameter** declaration v is the tuple

$$SIG_v = (N_v, T_v, A_v, M_v, S_v) \tag{5}$$

where N_v, T_v, A_v, M_v, S_v are the name, type, annotation list, modifier (i.e., *final*) and scope

of v, respectively. S_v includes all statements v is visible to, and thus v can be referenced from. The scope of a variable starts from the first statement following the declaration of the variable and ends to the last statement within the block in which the variable is declared. Figure 2 depicts the statements within the scopes of variables `endpointName` and `errorStr` in the respective rectangular boxes. The scope is essential for distinguishing variables with the same name and type declared in different blocks of a method.

4.2 Tracking Process

Our solution relies on RefactoringMiner [41] to track a program element in the commit history of a project, and report all changes and refactoring operations performed on it. Despite the fast execution time of RefactoringMiner (44 ms on median and 253 ms on average per commit), running it on the entire commit history of the project is computationally inefficient, as the tracked program element is changing in a relatively small subset of commits, and furthermore it is not always necessary to analyze all modified files in a commit to track a single program element, especially in large commits involving thousands of modified files. Therefore, we developed some heuristics and extended RefactoringMiner to perform partial analysis whenever possible.

Input: Similarly to CodeShovel, our tool is using as input a Git repository URL, a starting commit SHA-1 ID (or HEAD by default), the file path containing the program element of interest, the name of the program element, and the start line number of the program element's declaration. Both name and start line are needed to disambiguate the program element of interest, as it is possible to have multiple program elements with the same name (i.e., overloaded methods, identically named variables declared in different blocks of the same method), and it is possible to declare two or more variables/parameters on the same line.

Output: A graph in which the nodes represent program elements with their unique identifiers (i.e., program elements in different commits), and each edge connecting two nodes includes the list of changes between the corresponding program elements. The edges are directed from *child* commit program elements to the matching *parent* commit program elements. In other words, by traversing the output graph, we visit previous versions of a program element (i.e., *backward* tracking). We decided to model the output as a graph, because it is possible to have forks. For example, when multiple methods from different subclasses are pulled up in a single superclass method, the same program element (i.e., superclass method) is connected with multiple program elements (i.e., subclass methods). In addition, we model extracted and inlined methods as branches in the evolution graph of the main tracked program element, as we will explain in Section 4.3. To facilitate the comparison with CodeShovel [13], we can transform the graph output to a list of commits in which the input program element changed along with the corresponding kinds of changes in each commit.

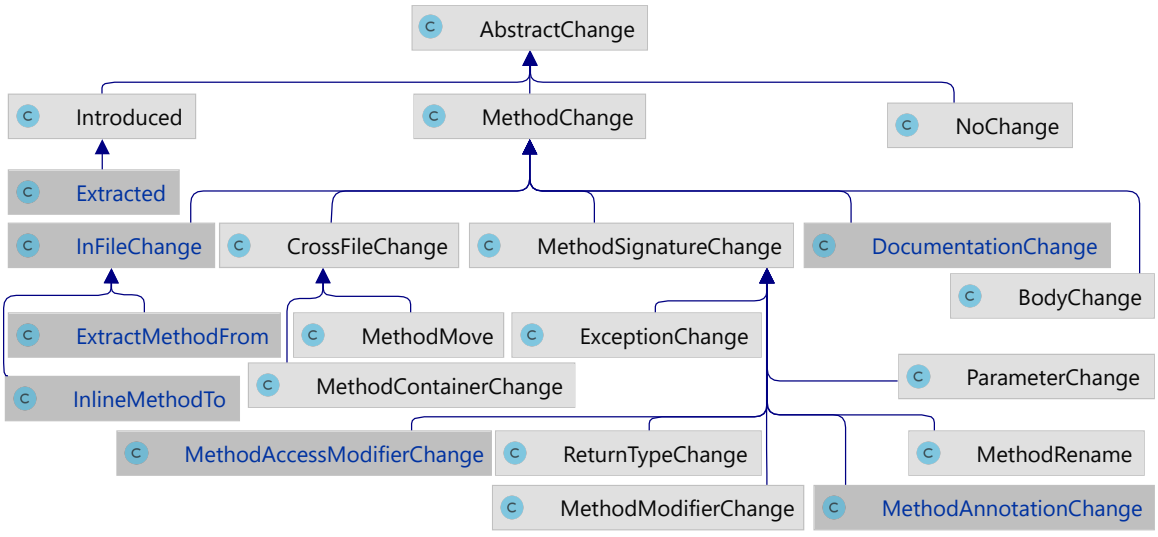

Figure 3: Hierarchy of supported change kinds for methods adopted by CodeShovel [13]. Newly supported change kinds are highlighted in darker background colour and blue text color.

We adopted and extended the change hierarchy supported by CodeShovel [13] for

method tracking, as shown in Figure 3. The newly supported change kinds are highlighted in darker background colour and blue text color, and deal with *InFileChange*, i.e., the extraction of a new method from the body of the tracked method, and the inline of a method within the body of the tracked method. *InFileChange* is essential for supporting the evolution scenario of our motivating example, shown in Figure 1.

In addition, we support three more kinds of changes, namely *AnnotationChange*, *AccessModifierChange*, and *DocumentationChange*. The latter involves changes in the *Javadoc* or *inline* comments within the body of the tracked method. We also separated access modifier change of code elements from the rest of their modifiers because access modifier plays essential roles relative to other modifiers. Developers can hide a code element from an API and broke the API by changing their access modifier or exposing them to be accessible by the public.

Finally, for some change kinds we have a more fine-grained reporting. For example, CodeShovel reports any change(s) in the parameter list of a method as a single *ParameterChange*, while we report individually for each parameter the following fine-grained changes, *Add*, *Remove*, *Rename*, *ChangeType*, *Merge*, *Split*, and *Reorder*.

Figure 4 shows the change hierarchy supported by our tool for variable tracking. *InFileChange* supports the scenario of a variable declaration being moved to another method in the same container, as part of a code fragment extracted to a new method or inlined from a previously existing method, while *CrossFileChange* supports the scenario of a variable declaration belonging to a method moved to another container.

Figure 5 shows the change hierarchy supported by our tool for class tracking. *ClassModifierChange* supports addition and removal of modifiers like *final*, *static*, *abstract*. *ClassDeclarationKindChange* is a change that can perform on classes to change a concrete type to an interface or vice versa. This kind of change is common, especially when a developer wants to extract an interface or remove an existing interface from the codebase

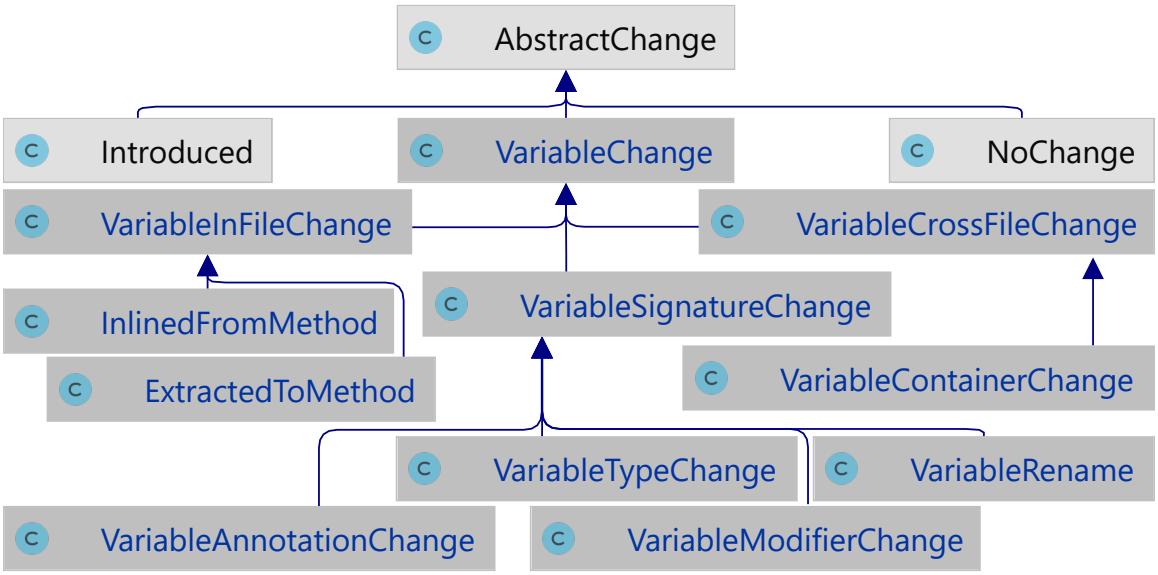

Figure 4: Hierarchy of supported change kinds for variables.

without forcing changes to other components or users of that class. In such cases, they keep the type and change the declaration kind of that type. In the case of extraction, they first extract a subclass from the original class and pull up all fields and methods to the newly extracted subclass. Then they change the declaration kind of the existing type from concrete class to interface. To make the change transparent from the users of that class, they declare the signature of public methods in the interface. In the case of removal, developers do not actually remove the interface. In fact, they change the declaration kind of the interface to a concrete class or an abstract class, then pull up the methods and fields of the class that implemented the interface to this class. In this stage, the subclass and the superclass become practically the same. So they remove the subclass. Since the users of the interface depend on the interface, removing the subclass is safe and does not boke the API.

Figure 6 shows the change hierarchy supported by our tool for field tracking. *AttributeModifierChange* supports addition and removal of applicable field's modifiers like *final*, *static*, *transient*, *volatile*. In addition, *AttributeAnnotationChange* supports addition, removal, and modification all aspects of annotation declared on fields.

The tracking process consists of the following steps:

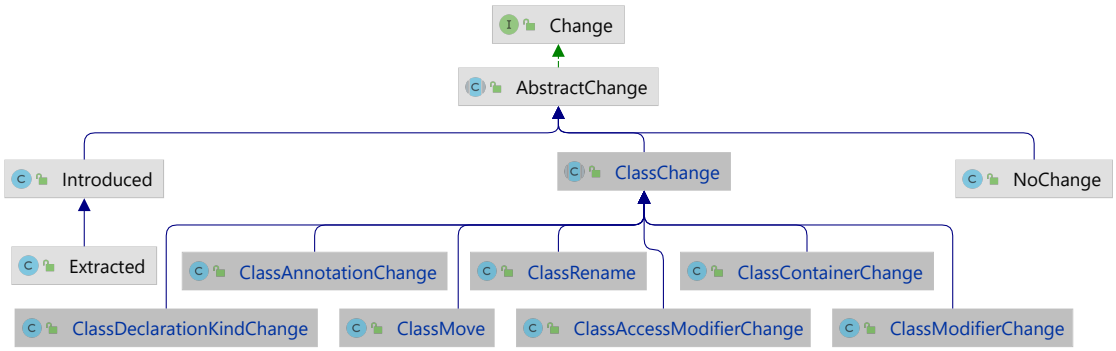

Figure 5: Hierarchy of supported change kinds for fields.

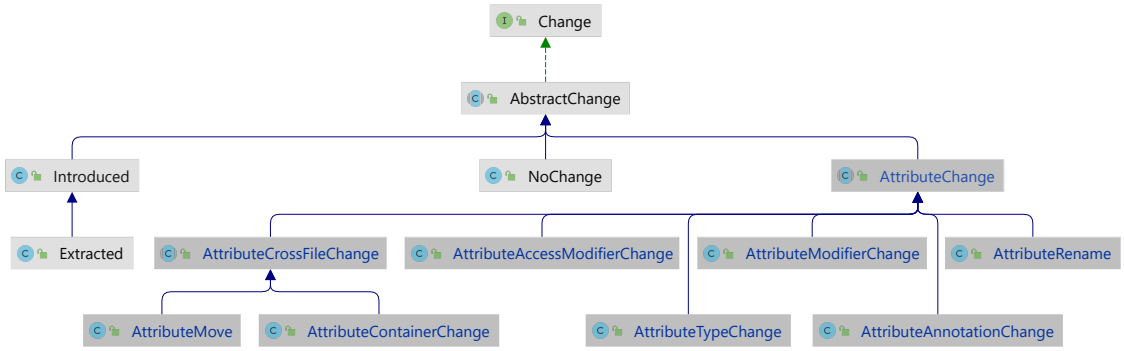

Figure 6: Hierarchy of supported change kinds for fields.

Step #1: Given the input file path in which program element e is located, we first find all commits in the project's history in which file path is modified using the command `git log --follow filePath`. The `--follow` option is particularly important, as it continues listing the history of a file even when it gets renamed or moved. The first step is based on the assumption that if there are no changes in the file containing e in a given commit, then we can skip the analysis of this commit as e has no changes too.

Step #2: For each commit r in the subset of commits obtained from Step #1, we create a partial source code model for r and p (i.e., r's parent commit) by parsing only the source file corresponding to the input file path. If r is the starting commit, then we locate program element e in r's model using its name and start line, and construct its signature (sig_{e_r}) and container (con_{e_r}) as explained in Section 4.1. If r is a subsequent commit, then we have

e_r's signature and container from the previous iteration of the tracking process (i.e., the matched program element from the previously processed commit). Then, we attempt to locate a program element with the same signature and container in p's model. If a match is found, then we link the two program elements (e_r, e_p) with their unique identifiers and report *NoChange*. Such a match is possible when all containers of the tracked program element up to the root have identical signatures, but the file has changes in other irrelevant parts. In such case, there is no need to execute RefactoringMiner. If no match is found, we relax the comparison of signatures, as explained in the next step.

Step #3: If program element e itself or its container is a method (i.e., e is a variable), then we omit B_m (i.e., the hashed value of the method's body string representation) from the method's signature tuple, and we attempt to locate a program element with the same relaxed signature and container in p's model.

If a match is found and e is a method, we link the two methods (e_r, e_p) with their unique identifiers and report *BodyChange*. Such a match is possible when there are changes in the body of the tracked method, but its signature remains unchanged. However, we still need to check if the tracked method is involved in an EXTRACT METHOD or INLINE METHOD refactoring. To avoid an unnecessary execution of RefactoringMiner, we extract all method calls from e_r and e_p, respectively, and filter the calls that do not have a caller expression or have `this` as a caller expression (i.e., the calls to local methods). If there are additional calls in e_r's call list, then we need to check if a local method was extracted from e_p. If there are additional calls in e_p's call list, then we need to check if a local method was inlined to e_r. In both cases, we execute RefactoringMiner on the partial source code models for commits r and p including only T_r (i.e., the type declaration containing e_r) and T_p (i.e., the type declaration containing e_p). For each EXTRACT METHOD or INLINE METHOD refactoring returned by RefactoringMiner, we introduce the extracted/inlined program elements as *evolution hooks* in the output graph (Section 4.3 includes more details). However, if the two

method call lists are identical, then there is no need to execute RefactoringMiner.

If a match is found and e is a variable, we link the two variables (e_r, e_p) with their unique identifiers and report *NoChange*. Such a match is possible when e_r and e_p still have the same name, type, modifier, annotations, and statements in their scopes, despite the changes in the body of their container method. However, if no matching variable is found, but the container methods con_{e_r} and con_{e_p} are matched, we extract all variables declared within the body of con_{e_r} and con_{e_p} and omit S_v (i.e., the list of statements within the variable's scope) from all variable signature tuples. If the two lists of relaxed variable signatures are identical, then we link e_r with the corresponding variable e_p (i.e., the variable having the same position in the list of variables declared within con_{e_p}). Such a match is possible when e_r and e_p still have the same name, type, modifier, annotations, but there are some syntactic changes in the statements within the scope of the variables. However, if the two lists of relaxed variable signatures are not identical, then we execute RefactoringMiner on the partial source code models for commits r and p including only T_r (i.e., the type declaration containing con_{e_r}) and T_p (i.e., the type declaration containing con_{e_p}). RefactoringMiner will initially match the container methods con_{e_r} and con_{e_p} and perform a thorough analysis after matching the statements within their bodies, in order to find *renamed*, *inlined*, *extracted*, *split*, *merged*, *moved* (due to method extraction or inline), *added*, *deleted*, and *matched* variables. We locate e_r in the reported refactorings, link e_r with the corresponding variable e_p, and report all changes found between them.

If by the end of Step #3 no match is found for e, this is an indication that there are major changes in the signature of e itself or its container. This scenario is addressed in the next step.

Step #4: Assuming that e_r is contained within type declaration T_r in commit r, and exists a type declaration T_p in commit p, where both T_r and T_p have an identical name and container signature, then we attempt to locate e_p within T_p, by executing RefactoringMiner on the

partial source code models including only T_r and T_p. RefactoringMiner initially matches the method pairs with identical signatures (i.e., method name and parameter types), and then compares all combinations of the remaining unmatched methods from T_r with the remaining unmatched methods from T_p to find the best matching method pairs with changes in their signatures [41].

If e_r is a method, we check if there exists a pair (e_r, e_p) in the best matching method pairs. If so, we link the two methods (e_r, e_p) with their unique identifiers and report all changes in their signatures and bodies. In addition, we add any local methods extracted from e_p or inlined to e_r as *evolution hooks* in the output graph (Section 4.3), as this information is provided by RefactoringMiner when comparing two type declarations.

If e_r is a variable, we check if there exists a pair (con_{e_r}, con_{e_p}) including the container of e_r in the best matching method pairs. If so, we retrieve all variable-related refactorings (i.e., *rename*, *inline*, *extract*, *split*, *merge*, *move*, *add*, *delete*, and *match*) extracted by RefactoringMiner for this pair of methods. We locate e_r in the reported refactorings, link e_r with the corresponding variable e_p, and report all changes found between them.

If by the end of Step #4 there is still no match found for program element e, this is an indication that either e itself or its container has been moved to another file, or the type declaration T_r containing e_r has been renamed or moved to another package. This scenario is addressed in the next and final step.

Step #5: At this stage, we keep the partial source code model including only T_r for commit r, 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). Then, we execute RefactoringMiner on these two source code models, and collect all reported refactorings, including RENAME CLASS, MOVE CLASS and MOVE METHOD. Step #5 is the most time-consuming step in the tracking process, as we include all modified files in commit p. During our experiments, we discovered some commits (e.g., in project hadoop [26]) in which the developers moved

thousands of source code files from one source folder to another. In this scenario, the moved files have a change in their file path, but their contents remain identical. 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 infer the corresponding MOVE SOURCE FOLDER refactoring operations simply by analyzing the changes in their file paths. Finally, we support three scenarios in which additional files need to be included in r's source code model to correctly track program element e.

(1) Changes on e can only be inferred from changes in other program elements: RefactoringMiner infers signature-level refactorings for method pairs not having a body (i.e., *interface* or *abstract* methods), or method pairs that could not be matched based on statement mapping information (i.e., methods with large differences in their bodies due to functionality changes) from the refactorings/changes detected on method pairs having identical signatures with the unmatched method pairs [41]. The intuition is that a change in the signature of an *abstract* or *interface* method should propagate to all concrete implementations of that method (i.e., overriding methods). Assuming that e_r is contained within type declaration T_r in commit r, we get the extended superclass and implemented interface types of T_r and check for each one of these types if it corresponds to a modified file in commit r to ensure the super type is a local type declaration of the analyzed system. If a super type S_r indeed corresponds to a modified file in commit r, then we use regular expressions to check if other modified files in commit r extend or implement S_r and add them to r's source code model. This approach enables the inference mechanism of RefactoringMiner with the least possible computation cost. For example, in project OkHttp [44], the method pair `synStream—headers` in inner class `SpdyConnection.Reader` is matched by additionally including class `MockSpdyPeer.InFrame` to r's source code model, as both classes implement the `FrameReader.Handler` interface.

(2) e is copied into a new file: In some projects, which are libraries with public APIs,

we found that developers tend to copy the methods they want to deprecate into a new file, and then declare the original methods or their container class as `@deprecated`. Assuming that e_r is copied in type declaration T_r in commit r from type declaration T'_p in commit p. Without additionally including the original type declaration containing the copied method T'_r to r's source code model, then e_r would be detected as *moved* from T'_p to T_r, instead of *introduced* in T_r as a new method. To address this issue we use a regular expression to check if other modified files in commit r include a `@deprecated` annotation with a `@link` to e_r's signature (e.g., `copy` methods copied from `IOUtils` to `CopyUtils` in project commons-io [27]), or a `@deprecated` annotation with a reference to T_r name (e.g., deprecated classes `IOUtil` and `EndianUtil` referring to newly added classes `IOUtils` and `EndianUtils`, respectively, in project commons-io [28]) and add them to r's source code model. Moreover, we check if other modified files in commit r have the same name as T_r, but different package (e.g., methods copied from deprecated class `org.apache.commons.lang.NumberUtils` to new class `org.apache.commons.lang.math.NumberUtils` in project commons-lang [8]) and add them to r's source code model.

(3) e **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). Assuming that e_r is moved in type declaration T_r in commit r from type declaration T'_p in commit p. Without additionally including the original type declaration containing the moved method T'_r to r's source code model, then T'_p would be detected as *renamed* to T_r (if multiple members from T'_p have been moved to T_r), instead of T_r being extracted from T'_p, and T'_r being matched with T'_p. To address this issue we use a regular expression to check if other modified files in commit r create an instance of

T_r (e.g., methods moved to extracted class `SourceFileInfoExtractor` from class `ProjectResolver` in project javaparser [40]), or are extended by T_r (e.g., methods pushed down to extracted subclass `AbstractNestablePropertyAccessor` from origin class `AbstractPropertyAccessor` in project spring-framework [32]) and add them to r's source code model.

Step #5a: Assuming that e_r is contained within type declaration T_r in commit r, we check all class-related refactorings (i.e., *rename*, *move* class) to find a pair of type declarations (T_r, T_p) involving T_r. If such a pair is found, we obtain the corresponding class-level diff object from RefactoringMiner, which includes all pairs of matched methods.

If e_r is a method, then we check if there exists a pair (e_r, e_p) in the matching method pairs. If so, we link the two methods (e_r, e_p) with their unique identifiers, and report a *FileMove* change (i.e., T_r is renamed/moved to T_p) in addition to any changes in their signatures and bodies. Moreover, we add any local methods extracted from e_p or inlined to e_r as *evolution hooks* in the output graph (Section 4.3), as this information is provided by RefactoringMiner when comparing two type declarations.

If e_r is a variable, we check if there exists a pair (con_{e_r}, con_{e_p}) including the container of e_r in the matching method pairs. If so, we retrieve all variable-related refactorings (i.e., *rename*, *inline*, *extract*, *split*, *merge*, *move*, *add*, *delete*, and *match*) extracted by RefactoringMiner for this pair of methods. We locate e_r in the reported refactorings, link e_r with the corresponding variable e_p, and report a *MovedWithMethod* change (i.e., con_{e_r} has been relocated to con_{e_p}) in addition to any changes found between the variables.

If by the end of Step #5a there is still no match found for program element e, this is an indication that either e itself or its container has been moved to another file.

Step #5b: Assuming that e_r is contained within type declaration T_r in commit r, we check all method-related refactorings involving moves (i.e., *move*, *pull up*, *push down* method) to find a pair of method declarations (e_r, e_p) if e_r is a method, or (con_{e_r}, con_{e_p}) if e_r is a

variable, involving a move from T_r. If such a pair is found, we proceed in the same way as described in Step #5a with the only difference being the report of a *MethodMove* change instead of a *FileMove* if e_r is a method.

If by the end of Step #5b there is still no match found for program element e, we report that e_r has been *Introduced* in commit r. We further examine the refactorings reported by RefactoringMiner to find if e_r is introduced by an EXTRACT METHOD refactoring, and add the method(s) from which e_r is extracted as *evolution hooks* in the output graph (Section 4.3). If a matching program element e_p is found throughout Steps #2 to #5, we use its signature (sig_{e_p}) and container (con_{e_p}) to continue tracking program element e in the remaining commits obtained from Step #1.

4.3 Change Graph Evolution Hooks

It is very likely that a developer would like to inspect the evolution of program elements which are extracted from or inlined to the main tracked program element. The intuition behind this feature is that the extracted/inlined program elements were originally part or became part of the tracked program element at some point in its evolution. To support this feature our program element tracking process (Section 4.2) introduces the extracted/inlined program elements as *evolution hooks* in the change graph of the main tracked program element. The developer can attach on demand the evolution sub-graph of an extracted/inlined program element by expanding the corresponding *evolution hook*.

Figure 7 shows how we model the evolution of an extracted program element on the change graph of a tracked program element. Assuming an EXTRACT METHOD refactoring takes place in commit r, then we create a node for the extracted element e'_r with a unique identifier $I_{e'_r}$ including the commit ID of r, since e'_r starts existing in commit r, its signature ($sig_{e'_r}$) and container ($con_{e'_r}$) constructed as explained in Section 4.1. When the developer decides to expand the e'_r node, we use $I_{e'_r}$ and start commit r as input for our tracking

process, which is executed *forwards* in this case (i.e., from parent commit to child commit), and attach the resulting graph on the e'_r node, as shown in Figure 7.

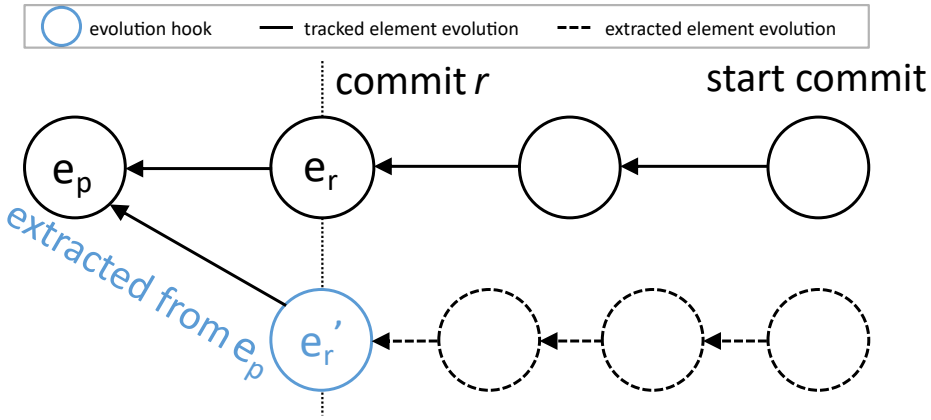

Figure 7: Tracking of a program element extracted from the tracked element.

Figure 8 shows how we model the evolution of an inlined program element on the change graph of a tracked program element. Assuming an INLINE METHOD refactoring takes place in commit r, then we create a node for the inlined element e'_p with a unique identifier $I_{e'_p}$ including the commit ID of p (i.e., the parent commit of r), since e'_p last exists in commit p, its signature ($sig_{e'_p}$) and container ($con_{e'_p}$) constructed as explained in Section 4.1. When the developer decides to expand the e'_p node, we use $I_{e'_p}$ and start commit p as input for our tracking process, and attach the resulting graph on the e'_p node, as shown in Figure 8.

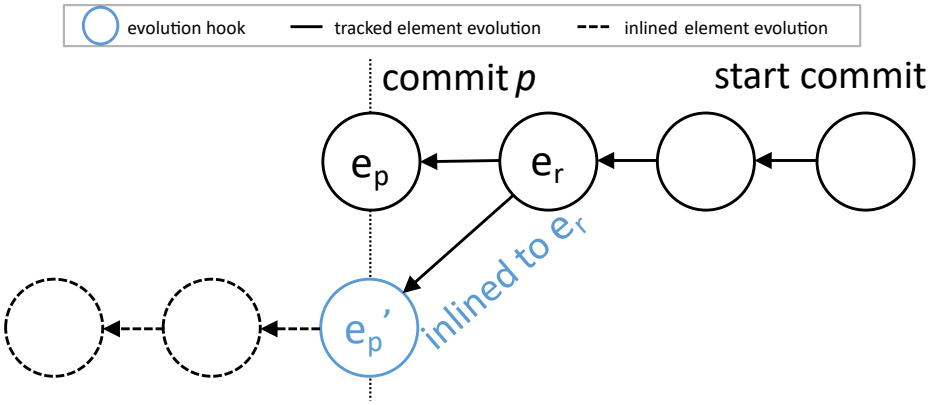

Figure 8: Tracking of a program element inlined to the tracked element.

It is also very likely that a developer would like to inspect the evolution of the method(s)

from which the tracked program element is extracted, similar to the evolution scenario shown in our motivating example (Section 2.1). Figure 9 shows how we model the evolution of a program element, which is the origin of extraction for a tracked program element. Assuming an EXTRACT METHOD refactoring takes place in commit r, then we create a node for the origin element e'_p with a unique identifier $I_{e'_p}$ including the commit ID of p (i.e., the parent commit of r), since e_r was originally contained in e'_p in commit p, its signature ($sig_{e'_p}$) and container ($con_{e'_p}$) constructed as explained in Section 4.1. When the developer decides to expand the e'_p node, we use $I_{e'_p}$ and start commit p as input for our tracking process, and attach the resulting graph on the e'_p node, as shown in Figure 9.

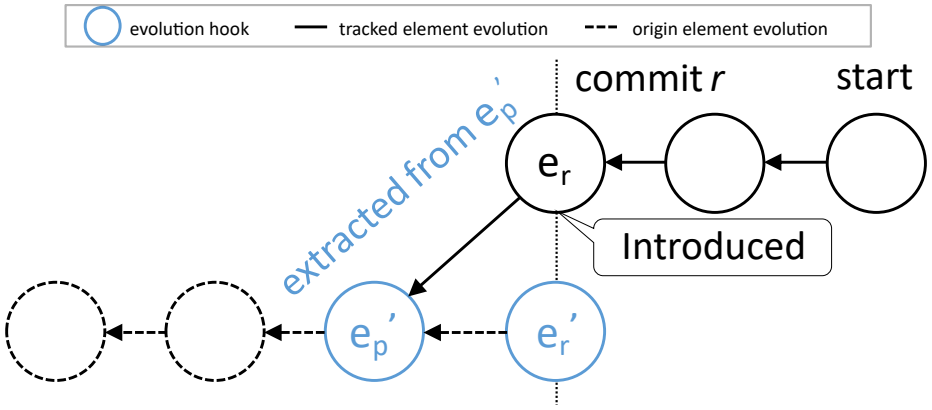

Figure 9: Tracking of a program element from which the tracked element is extracted.

Chapter 5

Implementation

In this chapter, we describe some details about the implementation of CodeTracker. In Section 5.1, we discuss the languages and platforms that our tool is depending. Then Section 5.2 demonstrates the CodeTracker API. Finally, Section 5.3 reveals Code Element Types in more detail.

5.1 Languages and Platforms

CodeTracker is implemented in Java and uses Maven as its build tool. The most important dependencies of CodeTracker are RefactoringMiner, JGit, and Guava.

To track a program element in the commit history of a software project and report all changes and refactoring operations performed on it, CodeTracker uses RefactoringMiner [41]. RefactoringMiner uses the JGit API to obtain the contents of required files from the parent and child commits directly from the .git folder of the repository. Then it parses the files extracted from each revision using the Eclipse JDT Abstract Syntax Tree (AST) Parser and converts them to a source code model. We reused some part of this functionality to create a partial source code model by parsing specific Java files and converting them to our code element types that we describe later in Section 5.3, whenever needed. So our tool

indirectly depends on Eclipse JDT Abstract Syntax Tree (AST) Parser.

To model the output as graph-structured data, CodeTracker leverages Guava's graph library, which contains three top-level graph interfaces: Graph, ValueGraph, and Network. Graph is the simplest and most fundamental graph type. It defines the low-level operators for dealing with node-to-node relationships. ValueGraph has all the node-related methods that Graph does, but adds a couple of methods that retrieve a value for a specified edge. Network has all the node-related methods that Graph does, but adds methods that work with edges and node-to-edge relationships. We opted to use directed ValueGraph because we need to associate a value to the edges representing the changes between two versions of a code element.

As we describe in Section 4.2, at some point in our tracking algorithm, we need to find all commits in the project's history in which a file path is modified. To do so, we need to use the command `git log --follow filePath`. To perform this Git operation, we use the JGit library. So our tool depends on this library directly.

5.2 CodeTracker API

Since source code history is a crucial contributor to program evolution comprehension for practitioners and plays an essential role in many areas of software engineering research, we decided to expose the functionality of CodeTracker as an API. The simplicity of the API was a major and important goal, so we decided to design a single entry point for CodeTracker as a core API. The core API of CodeTracker, as shown in figure 10, contains four different tracker classes for each code element: `ClassTracker`, `MethodTracker`, `VariableTracker`, and `AttributeTracker`. All of them are instantiable via their builder. We use builders to improve the extensibility of the API in the future, so we will not be forced to preserve the signatures of our API methods to maintain backward compatibility.

The CodeTracker interface can instantiate an appropriate builder for a code element.

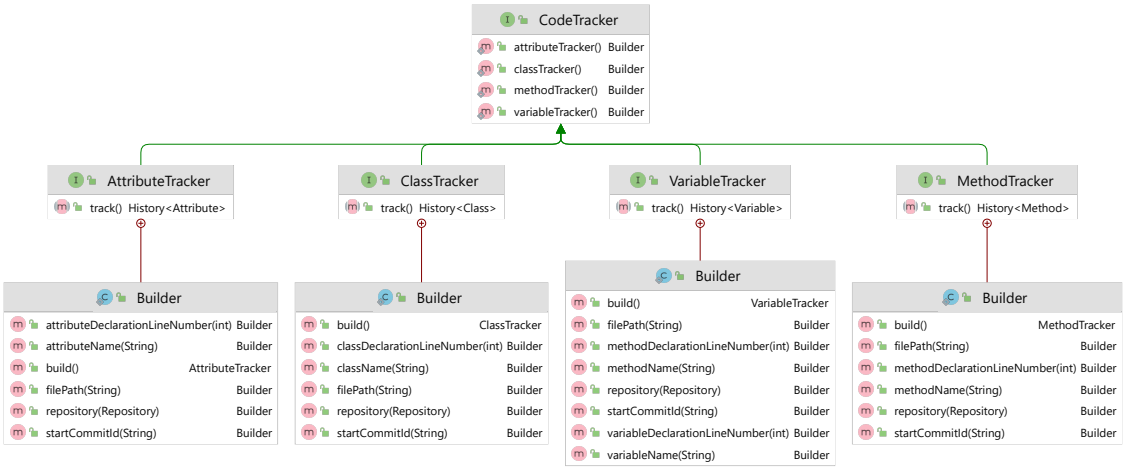

Figure 10: CodeTracker API

It provides a method that returns a builder for each of the four code elements it supports. Builders mainly collect the repository information and the inputs described in Section 4.2 to identify the element of interest. Code Snippet 5.1 shows how `MethodTracker` API can be instantiated and used to track the history of a method with name `fireErrors` declared in line 384 of a file with name `Checker.java`, and start commit `119fd4fb`.

Listing 5.1: Instantiation of a Method Tracker

```
GitService gitService = new GitServiceImpl();
try (Repository repository =
    gitService.cloneIfNotExists("tmp/checkstyle",
    "https://github.com/checkstyle/checkstyle.git")){

   MethodTracker methodTracker = CodeTracker.methodTracker()
    .repository(repository)
    .filePath("src/main/java/com/puppycrawl/tools/checkstyle/Checker.java")
    .startCommitId("119fd4fb33bef9f5c66fc950396669af842c21a3")
    .methodName("fireErrors")
    .methodDeclarationLineNumber(384)
```

```
    .build();

    History<Method> methodHistory = methodTracker.track();
  }
```

As shown in code snipped 5.1 and Figure 10, the result of calling of `track` operation on tracker classes is a generic version of `History` interface that is parameterized over code element types. For example, `AttributeTracker`, `ClassTracker`, `VariableTracker`, and `MethodTracker` will return `History<Attribute>`, `History<Class>`, `History<Variable>`, and `History<Method>` respectively, after calling track method on their instances. As shown in Figure 11, `History` interface provides a history graph in which the nodes represent different version of the element of interest, and each edge connecting two nodes includes the list of changes between the corresponding program elements. To avoid propagating the dependency to Guava's graph library to the users, we provided our graph API that hides the dependency to the Guava library.

5.3 Code Element Types

Figure 12 demonstrates the code element types that are a composition of the UML model instantiated by RefactoringMiner. For each code element, we generate a unique identifier based on the way we discussed in Section 4.1. All required information for generating the program element identifier is available in the models that we generate from parsing source code using RefactoringMiner.

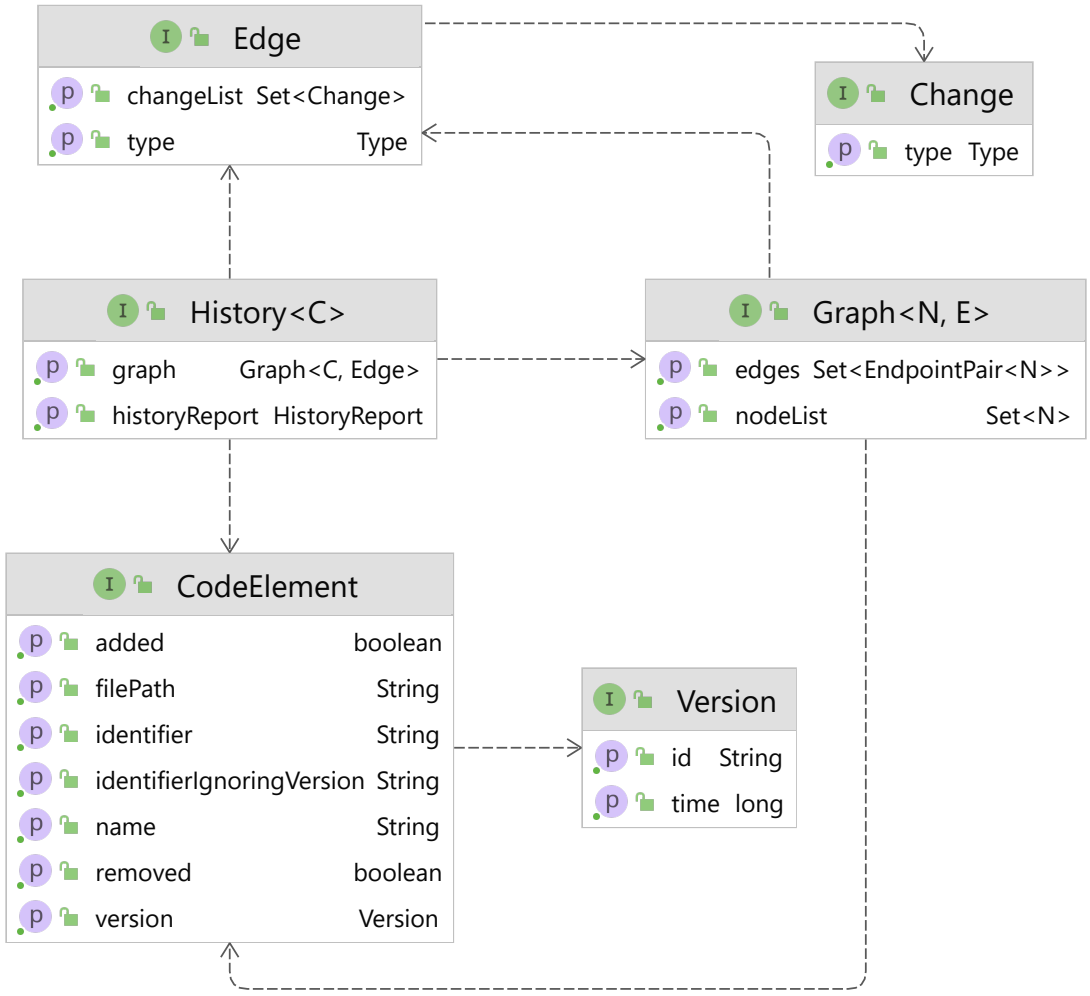

Figure 11: History Interface and its dependencies

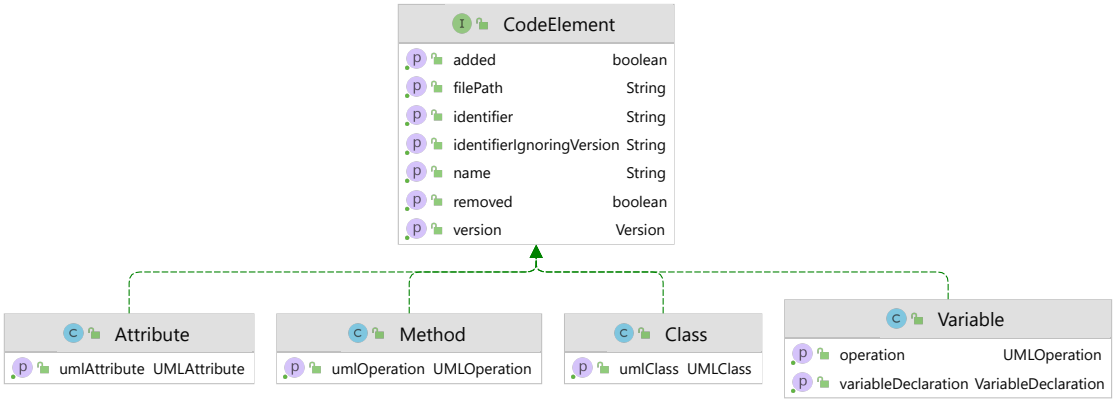

Figure 12: Code element Types

Chapter 6

Evaluation

This chapter provides details about the way we update and extend the oracle we used to evaluate the performance of our tool in section 6.1, demonstrates method, variable, class, and field tracking accuracy in section 6.2, section 6.3, section 6.4, and section 6.5, respectively. Finally, it illustrates the execution time of CodeTracker and CodeShovel in section 6.6.

6.1 Oracle Update and Extension

Grund et al. [13] created an oracle with the change history of 200 methods from 20 popular open-source project repositories. In particular, they used 100 of these methods (*training set*) to optimize the threshold values used in CodeShovel, until they achieved 100% training accuracy, and the remaining 100 methods (*testing set*) to validate the accuracy of CodeShovel. We decided to use both the training and testing sets to evaluate the accuracy of our tool and compare with that of CodeShovel, since Grund et al. spent 100 hours of manual validations to construct their oracle, and thus we can consider it as very reliable. However, after executing our tool we found major differences in the commit history of some methods. After careful inspection, we found out that 18 methods from the *training set* and 9 methods from the *testing set* were matched with a method extracted from their body at some point in their

change history. As a result, after the commit in which the the originally tracked method is erroneously matched with an extracted one, the oracle includes the change history of the extracted method, instead of the original method. In all these cases, a significant portion of the originally tracked method (over 75%, or even the entire method body) is extracted to a new method. Silva et al. [35] found that developers in many cases extract a large portion or even the entire body of a method into a new one, either to introduce an alternative method signature (i.e., different input/output parameter types), or to deprecate a method that is no longer needed. The original method remains in the code base and delegates to the extracted one in order to preserve the public API (i.e., backwards compatibility). Despite the strong similarity of the extracted method with the original method, it is not correct to match them, as the original method still remains in the code base with an identical signature (i.e., method name and parameter types) in most cases, and thus should be further tracked. As we discussed in Section 4.3, we model such cases as branches in the change graph of the tracked method using *evolution hooks*, and continue tracking the changes on the original method. Moreover, we found some cases where the oracle reports a method being moved to another file (i.e., *MethodMove* change), while in reality the type declaration that contains the method has been renamed or moved, and thus a *FileMove* change is the correct one.

We corrected all discrepancies found in the oracle by removing the false change instances due to incorrect method matches and adding the new change instances resulting after the correction of method matches. All removed and new change instances have been manually validated by inspecting the changes on GitHub. Table 1 provides a detailed overview of the changes being *common* with the original oracle (C columns), the changes *removed* from the original oracle (R columns), and the *new* changes added to the original oracle (N columns) for both training and testing sets.

We further extended this oracle with the change history of the local variables and parameters declared in these 200 methods. Since their number is large (967 variables in the

Table 1: Updates in the oracle created by Grund et al. [13]

	Training set			Testing set		
Change Kind	**C**	**R**	**N**	**C**	**R**	**N**
Body Change	2276	234	29	459	68	26
File Move	238	24	27	160	7	27
Parameter Change	220	30	6	68	19	8
Return Type Change	47	10	4	15	2	1
Modifier Change	44	12	2	16	4	1
Exception Change	40	9	0	5	3	2
Rename	14	8	7	16	6	2
Method Move	19	23	3	14	27	3
Introduced	81	18	19	83	17	17
Documentation Change	—	—	439	—	—	96
Annotation Change	—	—	42	—	—	24
Total	2979	368	578	836	153	207

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

training set and 378 variables in the *testing set*), we decided to follow a semi-automated approach to construct the oracle, instead of sampling a small number of variables and manually tracking their changes in the commit history. We leverage information from the method tracking oracle, as we know for sure that the commits in which a variable changed is a subset of the commits in which its container method changed. Next, for each variable, we execute our tool to perform backward tracking on the commits in which the container method of the variable changed. The output is a subset of these commits along with the changes found in each commit for the tracked variable. There are two possible termination conditions:

1. The variable tracking reaches the commit in which the container method was introduced. This means that the variable exists since the introduction of its container method.

2. The variable is introduced in a commit before reaching the container method introduction commit. This means that the variable was added as part of new functionality implemented in the container method, or because some other method was inlined to the container

method.

We validate all reported changes by manually inspecting the corresponding commits. If a reported change is correct, we add it in the oracle. If a reported change is incorrect, this means that there was a wrong variable match performed by our tool. In that case, we manually determine the correct match/change for the tracked variable, add it in the oracle, and continue the tracking process from the parent commit for the correctly matched variable. This process is repeated until we reach one of the two termination conditions for each tracked variable. Reaching a termination condition guarantees that we have no missing variable changes in our oracle. The number of instances per change kind for the variables are shown in Table 2.

Table 2: Number of instances per change kind for variables.

Change Kind	**Training set**	**Testing set**
Introduced	967	378
Rename	142	27
Annotation Change	13	4
Type Change	259	67
Modifier Change	122	52
Total	1504	531

We continued extending the oracle with the change history of the classes that containing 200 methods of the original oracle. There are 165 classes in total (77 classes in the *training set* and 88 classes in the *testing set*) that we construct the changing oracle for them with a similar approach that we perform to create the oracle for variables. We reuse information from the method tracking oracle, as we know that the commit sets in which a class changed contains the commits in which a container change is performed on a method of that class. For each sample, we run our tool to find the change history of that class. We automatically confirmed all the changes we already have in the method's oracle as container change. In addition, we automatically confirmed all the introduced changes reported for the initial

commit of a repository. We manually validated all other changes reported by our tool and have not validated automatically in the previous step. The number of instances per change kind for the classes are shown in Table 3.

Table 3: Number of instances per change kind for classes.

Change Kind	**Training set**	**Testing set**
Introduced	77	88
Rename	25	20
Class Move	97	52
Annotation Change	33	16
Move Source Folder	110	134
Access Modifier Change	12	4
Modifier Change	14	11
Declaration Kind Change	3	3
Total	371	328

Finally, we built an oracle containing the change history of all fields declared in the classes listed in the class's oracle. We extracted 806 items from the training oracle. We used a similar approach to previous steps to create this oracle. We reuse the class and method oracles information to automatically validate some changes like container change since the container of methods and fields of a class is similar. Also, we used the introduced commits of class oracle to automatically confirm the introduced commits of fields that are added with their class at the same commit. We validate all the other reported changes that could not confirm automatically by manually inspecting the corresponding commits. If a reported change is correct, we add it in the oracle. In case of incorrect detection, we put the correct change in the oracle and rerun the CodeTracker for the class that we believe is a good match in the history of the element of interest. The number of instances per change kind for the fields are shown in Table 4.

All change history oracles (method, variable, class, and field) are publicly available online [17] to enable the replication of our experiments and facilitate future research on program element tracking techniques.

Table 4: Number of instances per change kind for fields.

Change Kind	Training set
Container Change	950
Introduced	806
Type Change	165
Rename	144
Modifier Change	79
Field Move	73
Access Modifier Change	71
Annotation Change	9
Total	2297

6.2 Method Tracking Accuracy

We computed the precision and recall of CodeTracker and CodeShovel at two levels of granularity, namely *commit level* (i.e., finding the commits in which a method changed), and *change level* (i.e., finding the kinds of changes that occurred in the commits in which a method changed). When there are only changes in inline comments within the body of a method, CodeShovel reports a *BodyChange*. To ensure a fair comparison, we considered such cases as true positives, despite being labelled as *DocumentationChange* in the updated oracle. Moreover, for any kind of change in the parameter list of a method, CodeShovel reports a *ParameterChange*, while CodeTracker reports more fine-grained parameter changes, such as *Add*, *Remove*, *Rename*, *ChangeType*, *Merge*, *Split*, *Reorder*, *Add/Remove Modifier*, and *Add/Remove/Modify Annotation*. We also considered the coarse-grained *ParameterChange* reports as true positives.

Based on the results shown in Table 5, our tool, CodeTracker, has a consistent performance in both *training* and *testing* sets at commit level, with an overall precision of 99.97% and recall of 99.97%. On the other hand, CodeShovel has a worse precision and recall on the *testing set* compared to the *training set*, as the similarity thresholds used internally by CodeShovel were optimized on the *training set*. This is an inherent limitation of approaches

relying on code similarity thresholds, as it is very difficult to derive *universal* threshold values that can work well for all projects, regardless of their architectural style, application domain, and development practices [41]. Overall, at commit level, CodeTracker achieves an increase of +7.5% in precision and +3.76% in recall over CodeShovel.

Table 5: Method tracking precision/recall at commit level

Dataset	Tool	TP	FP	FN	Precision	Recall
Training	CodeShovel	2751	219	74	92.63	97.38
	CodeTracker	2824	1	1	99.96	99.96
Testing	CodeShovel	776	68	65	91.94	92.27
	CodeTracker	839	0	0	100	100
Overall	CodeShovel	3527	287	139	92.48	96.21
	CodeTracker	3663	1	1	99.97	99.97

Based on the results shown in Table 6, our tool, CodeTracker, has a similar performance at change level as commit level, with an overall precision of 99.87% and recall of 99.91%. On the other hand, at change level, CodeShovel has a decrease of around 2% in both precision and recall compared to the commit level results. The FPs and FNs of CodeShovel are mainly due to mismatching 18 methods from the *training set* and 9 methods from the *testing set* with a method extracted from their body at some point in their change history. As a result, the majority of the FPs are body and signature changes found in the history of these 27 mismatched extracted methods, while the majority of the FNs are body and signature changes missed from the remaining history of the 27 original methods, in addition to missed changes in method annotations and Javadoc, which are not supported by CodeShovel. The 3 FPs for CodeTracker in the *training set* are due to a merge method scenario in project javaparser [40], for which RefactoringMiner reports methods `solve(Node)` and `solveField(Node)` to be moved from `ProjectResolver` to the same method `solve(Node)` in class `SourceFileInfoExtractor`. Although the reported moves are technically correct, we considered one of them as the correct one, since our method evolution model supports

only *one-to-one* method mappings, similar to CodeShovel [13]. The remaining FPs and FNs for CodeTracker are *MoveMethod* changes misreported as *FileMove* by RefactoringMiner, because the child commit model did not include the origin file of the method (i.e., the three heuristics applied in Step #5 did not match the origin file of the method). Overall, at change level, CodeTracker achieves an increase of +9.1% in precision and +5.84% in recall over CodeShovel.

Table 6: Method tracking precision/recall at change level

Dataset	Tool	TP	FP	FN	Precision	Recall
Training	CodeShovel	3412	304	145	91.82	95.92
	CodeTracker	3556	3	1	99.92	99.97
Testing	CodeShovel	915	136	128	87.06	87.73
	CodeTracker	1037	3	3	99.71	99.71
Overall	CodeShovel	4327	440	273	90.77	94.07
	CodeTracker	4593	6	4	99.87	99.91

Finding: CodeTracker exhibits 99.9% precision and recall at both commit and change levels. Compared to the previous state-of-the-art tool, CodeTracker achieves an increase of +7.5% in precision and +3.76% in recall at commit level, and an increase of +9.1% in precision and +5.84% in recall at change level.

6.3 Variable Tracking Accuracy

Table 7 shows the precision and recall of CodeTracker at *commit* and *change* level. RefactoringMiner detects variable-related refactorings based the AST node replacements found in the pairs of mapped statements between two code fragments (i.e., the body of a method in the child and parent commits). As a result, the FPs are due to incorrect statement mappings, while the FNs are due to its inability to match some statement pairs. For example, in project junit5 [3], variable `method` is erroneously reported as *Introduced* instead of *Matched*, because RefactoringMiner is unable to match a `Collections.forEach()` call with

the equivalent `enhanced-for` statement. Moreover, missed *Matched* variables lead to more FNs, as the remaining changes in the history of the variables are also missed. This is particularly evident in the *training* set, as the methods it includes have longer change histories compared to those in the *testing* set. For all problems we found, we created issues in RefactoringMiner's issue tracker. Fixing these issues will further improve the accuracy of variable tracking.

Table 7: Variable tracking precision/recall for CodeTracker

Dataset	Level	TP	FP	FN	Precision	Recall
Training	Commit	1419	25	34	98.27	97.66
	Change	1452	40	51	97.32	96.61
Testing	Commit	504	3	8	99.41	98.44
	Change	519	5	9	99.05	98.30
Overall	Commit	1923	28	42	98.56	97.86
	Change	1971	45	60	97.77	97.05

Finding: CodeTracker exhibits 98.6% precision and 97.9% recall at commit level, and 97.8% precision and 97.1% recall at change level in tracking the change history of variable declarations.

6.4 Class Tracking Accuracy

Table 8 shows the precision and recall of ClassTracker at *commit* and *change* level. It shows that CodeTracker can detect all changes related to the classes without any miss. The results comply with the results for the method tracking in which CodeTracker can correctly report all container changes.

Finding: CodeTracker shows an outstanding performance in tracking the change history of class declarations and can detect all class-related changes with 100% precision and 100% recall in both commit and change levels.

Table 8: Class tracking precision/recall for CodeTracker

Dataset	Level	TP	FP	FN	Precision	Recall
Training	Commit	350	0	0	100	100
	Change	371	0	0	100	100
Testing	Commit	325	0	0	100	100
	Change	328	0	0	100	100
Overall	Commit	675	0	0	100	100
	Change	699	0	0	100	100

6.5 Field Tracking Accuracy

Table 9 shows the precision and recall of Field Tracking at *commit* and *change* level. RefactoringMiner detects refactorings in two-phase. In the first phase, its algorithm matches code elements top-down, starting from classes and continuing to methods and fields. Two code elements match if they have an identical signature. In the second phase, its algorithm matches the remaining code elements bottom-up, starting from methods/fields and continuing to classes to find the best match. As a result, the FPs are due to incorrect element matchings or false introduce change detection in the event of missing a possible match. At the same time, the FNs are due to RefactoringMiner's inability to map code elements or not continue the history due to an incorrectly introduced change finding by RefactoringMiner.

Table 9: Field tracking precision/recall for CodeTracker

Dataset	Level	TP	FP	FN	Precision	Recall
Training	Commit	2196	27	31	98.79	98.61
	Change	2241	56	56	97.56	97.56

Finding: CodeTracker exhibits 98.79% precision and 98.61% recall at commit level, and 97.56% precision and 97.56% recall at change level in tracking the change history of field declarations.

6.6 Execution Time

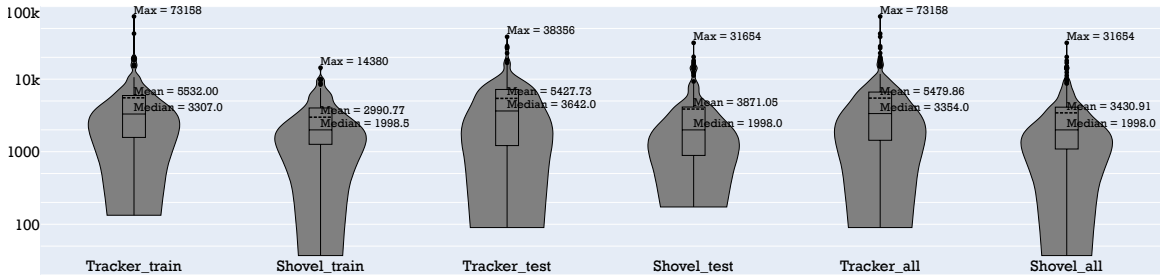

Figure 13: Execution time in milliseconds of CodeTracker and CodeShovel on training and testing sets.

Figure 13 shows the execution time of CodeTracker and CodeShovel for tracking the entire change history of each method in the training and testing sets, respectively (the y-axis is in logarithmic scale and the units are in milliseconds). Each tool was executed separately on the same machine with the following specifications: Intel Core i7-8565U CPU @ 1.80GHz $\times$ 8, 16 GB DDR3 memory, 512 GB SSD, Ubuntu 20.04.2 LTS operating system, and Java 11.0.11 x64 with a maximum of 8GB Java heap memory (i.e., `-Xmx8g`). All 20 project repositories from the training and testing sets were locally cloned before running the tools. For each tool, we recorded the total time taken for tracking a method in its entire commit change history, including the time taken for parsing the source code files and detecting the changes on the tracked method in each commit, using the `System.nanoTime` Java method. On median, CodeShovel processes the commit change history of a method in 2 seconds, while our tool, CodeTracker, takes 3.35 seconds (1.675 times slower). On average, CodeShovel processes the commit change history of a method in 3.4 seconds, while CodeTracker takes 5.5 seconds (1.62 times slower). However, CodeTracker has a more consistent performance between the *training* and *testing* sets with similar median and average execution times. This consistent performance can be explained from the observation that CodeTracker has a slightly higher percentage of commits reaching Step #5 (i.e., the most time consuming step of the tracking process) in the *testing set* compared to the *training set* (6.13% vs. 4.24%

as shown in Table 10), but at the same time has a much higher percentage of commits whose processing completes in Step #1 (i.e., the least time consuming step of the tracking process) in the *testing set* compared to the *training set* (82.33% vs. 67.52% as shown in Table 10). In other words, the *testing set* includes more commits in which the tracked method is moved or its container type declaration is moved/renamed, but at the same time has much more commits in which the tracked method remains unchanged.

Table 10: Average percentage of commits processed in each step of the tracking process.

Dataset	**Step #2**	**Step #3**	**Step #4**	**Step #5**
Training	62.93%	26.78%	5.04%	5.25%
Testing	70.91%	12.11%	3.84%	13.14%

On the other hand, CodeShovel has a longer average execution time on the *testing set* compared to the *training set*, despite the fact that the *testing set* has shorter commit change histories (34.5 and 47.5 commits on median and average, respectively) than the *training set* (73 and 86.3 commits on median and average, respectively). More specifically, CodeShovel has a larger than 10 seconds execution time for 10 methods of the *testing set* (8 of which are from project `intellij-community`), and only one method of the *training set*. Typically, the methods with longer execution times have multiple commits in which their container type declaration is moved/renamed (CodeShovel's phase 3), or the tracked method is moved (CodeShovel's phase 4). Both phases 3 and 4 are the most time consuming for CodeShovel as it widens its search to consider all other files that were modified in a commit.

To evaluate the speedup achieved through steps 2–5 of our approach, we executed RefactoringMiner with its default operation mode (i.e., with two complete source code models as input, including all modified/added files in commit r and all modified/removed files in commit p) right after step #1. The median execution time was 8.76 and 12.29 seconds, while the average execution time was 32.83 and 42.41 seconds on the *training* and *testing* sets, respectively. Therefore, the speedup of our approach is between 2.6–3.4 times on the

median execution time, and between 5.9–7.8 times on the average execution time, without jeopardizing accuracy, as almost the same precision and recall is achieved in both ways (with the exception of very few false positives/negatives discussed in Section 6.2).

Finding: Despite CodeTracker having a slower execution time than CodeShovel (1.75 times on median and 2 times on average), both tools have execution times within the same order of magnitude. The median and average execution time of CodeTracker is under 4 and 7 seconds, respectively, which is acceptable given the considerably increased precision and recall over CodeShovel, the additional computation of *evolution hooks* (Section 4.3), and the parallel tracking of the local variables and parameters declared in the tracked method (i.e., CodeTracker can perform parallel tracking of a method and its variable declarations without an additional computational cost). The achieved execution time can warrant applications in both research (e.g., large-scale MSR and software evolution studies) and practice (e.g., *blame*-like tracking of method and variable change history within the context of maintenance and program comprehension tasks).

Chapter 7

Limitations and Threats to Validity

In this chapter we discuss our limitation in section 7.1, and our threats to validity in section 7.2 and section 7.3. Finally discuss verifiability of ot work in section 7.4.

7.1 Language Specificity

CodeTracker depends on RefactoringMiner 2.0 [41] for the detection of refactorings and changes on the tracked program element, which limits its applicability to Java programs. However, recently there have been efforts to extend RefactoringMiner for supporting other programming languages, e.g., Python [2, 1] and Kotlin [22, 21]. With respect to CodeTracker, supporting another language would simply require to adjust the program element signature definitions and the regular expressions used in Step #5 to the characteristics and structure of this particular language.

7.2 Internal Validity

The main threat to the internal validity is related to the construction of the oracle used for evaluating the precision and recall. To mitigate this threat we relied on an existing oracle,

which was constructed by Grund et al. [13] after manually validating the change history for 200 methods (100 hours of validation). We further corrected all discrepancies found in the oracle caused by 27 methods (18 from the *training* and 9 from the *testing* set) being mismatched with a method extracted from their body, by manually inspecting and validating all new change instances that resulted after correcting the method matches. Moreover, we validated and added two new kinds of method changes (i.e., annotation and documentation changes) that were not previously supported. Based on the updated (and highly reliable) method history oracle, we constructed the change history of the variables declared in the body of these 200 methods following a semi-automated approach, as explained in Section 6.1, and manually inspecting all change instances. Overall, the manual validation effort for correcting and extending the Grund et al. oracle was approximately two person-months.

7.3 External Validity

Our experiments were conducted on a relatively small dataset including the change history of 200 methods from 20 different open source projects (i.e., 10 methods from each project), which might affected the generalizability of our findings. However, we decided to rely on the same dataset used for the evaluation of CodeShovel [13] to ensure a fair comparison between the two tools. Moreover, constructing an oracle from scratch with methods from other projects would involve a lot of manual effort and probably include more errors from incorrect validations. On the other hand, our extended oracle was essentially validated by two independent research groups, which makes it more reliable.

Regarding change kinds, we considered all possible changes that can be performed on method and variable declarations with the exception of changes in the initializer of variables, as the number of changes for this particular kind was very large, and manually inspecting all of them was prohibitive.

7.4 Verifiability

We make the source code of CodeTracker and our extended oracle publicly available [17] to enable the replication of our experiments and facilitate future research on program element tracking techniques.

Chapter 8

Conclusions and Future Work

Tracking source code history is an important activity that developers frequently perform to learn and make choices while reviewing or writing code. Also, it is essential in many areas of software engineering research. Since code evolution can be very complex as it may involve refactoring operations changing the project's structure, line-based history detection tools often return incomplete results. We developed a history detection tool named CodeTracker, a refactoring-aware tool that works on code elements levels and tracks the history of all code element types. Our evaluation shows that CodeTracker offers significant improvements over the previous state-of-the-art and novel tools.

In summary, the main conclusions and lessons learned are:

1. CodeTracker has high accuracy in tracking methods (99.9% precision/recall), variables (96.7% precision, 95.5% recall), classes (100% precision/recall), and attributes (98.7% precision, 98.6% recall).

2. The proposed heuristics for setting up RefactoringMiner to perform partial commit analysis, resulted in an execution time comparable to that of CodeShovel (i.e., CodeTracker is slower by 1.75 times on median and 2 times on average).

3. The speedup over the default operation mode of RefactoringMiner is 2.6–3.3 times on the

median, and between 5–6 times on the average execution time. Despite this considerable speedup, almost the same precision and recall is achieved in both ways with the exception of very few false positives/negatives.

The main goal of this study was to build an accurate tracking tool working on the code element level. CodeTracker gets the start element and tries to find its history individually. But some changes result from a significant difference in a higher level, like changing the architecture of a software project or removing a technical depth. So the next step of this study could be studying the history of all elements or a group of related code elements to find higher-level changes. One of the contributions of this work was improving the existing oracle of change history of some methods from 20 open-source projects and creating a new oracle for other code elements like class, attribute, and variable. So the next step for this part is refining and extending this oracle using different tools or techniques. We published our tool as an API to be accessible for researchers and practitioners for different uses but creating a user interface that shows the history of code elements could be more helpful and another future work. Finally, creating such a tool for other languages by extending our work is another step for this study.

Bibliography

[1] H. Atwi, B. Lin, N. Tsantalis, Y. Kashiwa, Y. Kamei, N. Ubayashi, G. Bavota, and M. Lanza. Pyref, 2021. `https://github.com/PyRef/PyRef`.

[2] H. Atwi, B. Lin, N. Tsantalis, Y. Kashiwa, Y. Kamei, N. Ubayashi, G. Bavota, and M. Lanza. Pyref: Refactoring detection in python projects. In *Proceedings of the 21st IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM 2021, 2021.

[3] S. Brannen. Junit5, 2021. `https://github.com/junit-team/junit5`.

[4] A. Brito, A. Hora, and M. T. Valente. Characterizing refactoring graphs in java and javascript projects. *Empirical Software Engineering*, 26(6):1–43, 2021.

[5] A. Brito, L. Xavier, A. Hora, and M. T. Valente. Apidiff: Detecting api breaking changes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 507–511, Los Alamitos, CA, USA, mar 2018. IEEE Computer Society.

[6] A. Brito, L. Xavier, A. Hora, and M. T. Valente. Why and how java developers break apis. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 255–265, Los Alamitos, CA, USA, mar 2018. IEEE Computer Society.

[7] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey. Software history under the lens: A study on why and how developers examine it. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ICSME '15, page 1–10, USA, 2015. IEEE Computer Society.

[8] S. Colebourne. Apache commons-lang, 2021. `https://github.com/apache/commons-lang/commit/2d06a7ce8`.

[9] B. E. Cossette and R. J. Walker. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. 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.

[10] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. Hassan. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, July 2017.

[11] D. Dig and R. Johnson. How do apis evolve? a story of refactoring. *J. Softw. Maint. Evol.*, 18(2):83–107, Mar. 2006.

[12] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31(2):166–181, Feb. 2005.

[13] F. Grund, S. A. Chowdhury, N. Bradley, B. Hall, and R. Holmes. CodeShovel: Constructing method-level source code histories. In *Proceedings of the International Conference on Software Engineering*, ICSE 2021, 2021.

[14] H. Hata, O. Mizuno, and T. Kikuno. Historage: Fine-grained version control system for java. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, page 96–100, New York, NY, USA, 2011. Association for Computing Machinery.

[15] Y. Higo, S. Hayashi, and S. Kusumoto. On tracking java methods with git mechanisms. *Journal of Systems and Software*, 165:110571, 2020.

[16] A. Hora, D. Silva, M. T. Valente, and R. 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.

[17] M. Jodavi. Codetracker source code and oracle, 2021. `https://github.com/jodavimehran/code-tracker`.

[18] P. Kapur, B. Cossette, and R. J. Walker. Refactoring references for library migration. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, page 726–738, New York, NY, USA, 2010. Association for Computing Machinery.

[19] A. Ketkar, N. Tsantalis, and D. Dig. Understanding type changes in java. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 629–641, New York, NY, USA, 2020. Association for Computing Machinery.

[20] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead. Automatic identification of bug-introducing changes. In *21st IEEE/ACM International Conference on Automated Software Engineering*, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[21] Z. Kurbatova. Kotlinrminer, 2021. `https://github.com/JetBrains-Research/kotlinRMiner`.

[22] Z. Kurbatova, V. Kovalenko, I. Savu, B. Brockbernd, D. Andreescu, M. Anton, R. Venediktov, E. Tikhomirova, and T. Bryksin. Refactorinsight: Enhancing ide representation of changes in git with refactorings information. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2021, 2021.

[23] M. Mahmoudi and S. Nadi. The android update problem: An empirical study. In *15th International Conference on Mining Software Repositories*, MSR '18, pages 220–230, New York, NY, USA, 2018. ACM.

[24] T. McDonnell, B. Ray, and M. Kim. An empirical study of API stability and adoption in the android ecosystem. In *IEEE International Conference on Software Maintenance*, ICSM '13, pages 70–79, Washington, DC, USA, 2013. IEEE Computer Society.

[25] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, Jan 2012.

[26] A. C. Murthy, C. Douglas, D. Das, G. Roelofs, J. Naisbitt, J. Wills, J. Eagles, K. Ramachandran, L. Lu, M. Konar, R. Evans, S. Agarwal, S. Seth, T. Graves, and V. K. Vavilapalli. Hadoop, 2021. `https://github.com/apache/hadoop/commit/dbecbe5df`.

[27] J. Märki. Apache commons-io, 2021. `https://github.com/apache/commons-io/commit/6a1bb4d53`.

[28] J. Märki. Apache commons-io, 2021. `https://github.com/apache/commons-io/commit/7748ed364`.

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

[30] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 803–813, New York, NY, USA, 2014. Association for Computing Machinery.

[31] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig. Is it dangerous to use version control histories to study source code evolution? In *26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 79–103, Berlin, Heidelberg, 2012. Springer-Verlag.

[32] S. Nicoll. Spring framework, 2021. `https://github.com/spring-projects/spring-framework/commit/2dc674f35`.

[33] G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto. Evaluating szz implementations through a developer-informed oracle. In *Proceedings of the International Conference on Software Engineering*, ICSE 2021, 2021.

[34] B. Shen, W. Zhang, H. Zhao, G. Liang, Z. Jin, and Q. Wang. IntelliMerge: A refactoring-aware software merging technique. *Proc. ACM Program. Lang.*, 3(OOPSLA):170:1–170:28, Oct. 2019.

[35] D. Silva, N. Tsantalis, and M. T. Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 858–870, New York, NY, USA, 2016. Association for Computing Machinery.

[36] D. Silva and M. T. Valente. RefDiff: Detecting refactorings in version histories. In *14th International Conference on Mining Software Repositories*, MSR '17, pages 269–279, Piscataway, NJ, USA, 2017. IEEE Press.

[37] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *International Workshop on Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[38] D. Steidl, B. Hummel, and E. 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.

[39] 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 143–152, 2005.

[40] F. Tomassetti. Javaparser, 2021. `https://github.com/javaparser/javaparser/commit/37f93be64`.

[41] N. Tsantalis, A. Ketkar, and D. Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 2020.

[42] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. 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.

[43] C. Williams and J. Spacco. SZZ revisited: Verifying when changes induce fixes. In *Workshop on Defects in Large Software Systems*, pages 32–36, New York, NY, USA, 2008. ACM.

[44] J. Wilson. Okhttp, 2021. `https://github.com/square/okhttp/commit/a91124b6d`.