Detection of Rename Local Variable Refactoring Instances in Commit History

Mohammad Matin Mansouri

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Science (Software Engineering) at
Concordia University
Montréal, Québec, Canada

January 2018

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Mohammad Matin Mansouri**

Entitled: **Detection of Rename Local Variable Refactoring Instances in Commit History**

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

Master of 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 commitee:

_____ Chair

-

_____ Examiner

Dr Joey Paquet

_____ Examiner

Dr Weiyi Shang

_____ Supervisor

Dr Nikolaos Tsantalis

Approved by _____

Dr Volker Haarslev, Graduate Program Director

31 January 2018 _____

Dr Amir Asif, Dean

Faculty of Engineering and Computer Science

Abstract

Detection of Rename Local Variable Refactoring Instances in Commit History

Mohammad Matin Mansouri

Detecting refactoring instances occurred in successive revisions of software systems can provide wealthy information for several purposes, e.g., to facilitate the code review process, to devise more accurate code merging techniques, to help the developers of API clients to ease their adaptation to API changes, and to enable more accurate empirical studies on the refactoring practice. In the literature there are several techniques proposed for refactoring detection, supporting a wide variety of refactoring types. Yet, almost all of them have missed an extensively-applied refactoring type, i.e., Rename Local Variable refactoring. In addition, all these techniques rely on similarity thresholds (which are difficult to tune), or need the systems under analysis to be fully built (which is usually a daunting task), or depend on specific IDEs (which drastically limits their effectiveness and usability).

In this thesis, we extend the state-of-the-art refactoring detection tool, RefactoringMiner, by defining necessary rules and extending its core algorithms to tailor it for accurately detecting Rename Local Variable refactoring instances. We have evaluated the proposed technique on two large-scale open-source systems, namely dnsjava and Tomcat. Our comparison with REPENT, the state-of-the-art tool in detecting Rename Local Variable refactoring instances, shows that our approach is superior in terms of precision and recall. Moreover, to automatically create a *reference corpus* of refactoring instances which is required for the evaluation, we have built a fully-automated infrastructure (called RefBenchmark) that is able to invoke several refactoring detection tools and find the agreements/disagreements between their results.

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The term **refactoring** was first coined by Opdyke in his PhD thesis [Opd92], and is defined as "the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure [Fow99]". Refactoring transformations involve renaming, moving, splitting, and joining program entities (e.g., local variables, fields, methods, classes, and packages) towards a higher-quality code. It is generally believed that avoiding refactoring incurs technical debt, which in turn increases the maintenance cost in the long run [BL76]. Several empirical studies have shown that refactoring indeed contributes positively to the improvement of design quality of software systems [KMPY06, MSAS06, RSG08, CKO10, BCG$^+$10]. A study at Microsoft revealed that 22% of developers initiate refactorings because of poor readability, and 11% because of poor maintainability [KZN14]. Moreover, 43% of developers mentioned that they actually perceived better code readability, and around 30% perceived improved code maintainability after refactoring. According to this study, developers spend about 10% of their time in each month working on refactoring (roughly 13 hours per month).

There is a long list of different refactoring types. A well-known catalogue of refactorings is provided by Fowler [Fow99], which includes 72 structural changes for improving code quality. Developers can apply these refactorings on the source code in different situations with different motivations [STV16]. Several studies have tried to come up with techniques for *automatically identifying refactoring opportunities* in code [Dal15, BDLMO14]. For example, some approaches find *code smells* as the opportunities for applying refactorings. For instance, a well-known code smell is a *Long Method*, i.e., a method that fulfills more than one functionality and therefore is overly complicated to understand and maintain. A Long Method code smell can be eliminated by applying *Extract Method* refactoring. A recent study, however, showed that code smells are not the main motivations for applying refactorings [STV16].

Applying refactoring transformations can be a challenging, error-prone, and time-consuming task. While developers usually tend to perform refactorings *manually* [VCN+12, MHPB12, STV16]), most modern IDEs include *automated refactorings* as a standard feature to apply these transformations automatically, and check some *preconditions* to make sure that the changes are indeed safe to apply.

Murphy-Hill and Black [MHB08] discovered two tactics of applying refactorings. Developers might apply *floss refactoring*, where the developer uses refactoring as a means to reach a specific goal, such as adding a feature or fixing a bug. In this case, refactorings are interleaved with other changes. On the other hand, in *root canal refactoring*, the main goal of the developer is to apply refactorings to improve the quality of the code. Murphy Hill et al. [MHPB12] showed that floss refactoring is a more common practice.

No matter how or why refactorings are done in the code, it has been shown that refactorings are indeed a very frequent development activity [XS06a, MHPB12].

The rest of this chapter will explain the motivation, problem statement, and the contribution of our thesis in details.

1.1 Motivation

Refactoring detection is the process of computing a (likely) set of refactorings that developers applied on the source code. Typically, refactoring detection tools take as input to successive versions (i.e., releases) or revisions (i.e., commits) of the source code, and infer the refactoring operations that took place in between, by analyzing the changes in the source code. There are several use cases for refactoring detection:

Change comprehension and code review It is often required that the developers of a software system understand what refactorings other developers have done to the code, especially during the code review process. For example, when looking at a textual Diff provided by version control systems, it is very difficult to distinguish the refactorings from other types of changes (e.g., bug fixes or feature additions) [KR11, DBG+15] as refactorings are usually interleaved with other changes [MHPB12, NVC+12, STV16]. A technique for extracting meaningful refactoring information from the changes, like ReviewFactor [GSMH14, GSWMH17] or ChangeDistiller [ASK14], can improve code review experience and accuracy. In addition, such a technique can also help in automatically documenting the applied refactorings, e.g., in the commit messages, since it has been shown that developers do not tend to list the refactoring operations they perform on the code in their commit messages [MHPB12].

Code merging Global refactorings (i.e., refactorings that span across the boundaries of a file or a module) can result in changes in several parts of the code. When using version control

systems, such global changes can easily lead to merge conflicts when, for example, an entity is renamed in several places of the code that another developer is also currently working on. A refactoring-aware version control system like MolhadoRef [DMJN08] on the other hand, can avoid such merge conflicts, by automatically detecting the refactorings and resolving the conflicts occurring because of them.

Client adaptation A library can undergo several refactorings, some of which might break its public API, currently used by several clients. Indeed, a previous study showed that more than 80% of the breaking changes to the APIs are because of refactorings [DJ06]. As a result, a technique for detecting these changes and reporting it to the clients would be needed. For example, RefactoringCrawler [DCMJ06] was introduced with this goal in mind. As a step further, refactoring detection tools have been used in automated techniques for client library adaptation [HD05, BTF05, DCMJ06, XS07].

Code completion Since the automated refactoring tools are underused [VCN+12, MHPB12, STV16], an approach that can detect an incomplete refactoring in the IDE can suggest to the developer to automatically complete a refactoring operation in progress [GDMH12, FGL12].

Improving the identification of bug-introducing changes Recent studies have shown that refactorings, such as file/directory renaming, parameter reordering, and variable/parameter renaming, affect significantly the accuracy of algorithms used for identifying bug-introducing changes [DRW14, dCMS+16], since this kind of *semantically equivalent changes* can be misinterpreted as changes introducing a bug. The SZZ algorithm designed by Śliwerski et al. [SZZ05] and subsequent improved versions [KZPW06, WS08], which have been used in a large number of empirical studies, can use refactoring detection tools to avoid flagging changes that do not change system behavior (i.e., refactorings) as bug-introducing.

Improving software traceability Recent studies have shown that refactorings removing redundant information, such as the Extract Method refactoring used for eliminating duplicated code (also known as software clones), affect negatively the performance of automated tracing tools based on information retrieval [MN13, MN14], because code clones actually serve a positive purpose for traceability link recovery. As a result, tracing tools can use refactoring detection tools to recover broken traceability links due to applied Extract or Pull Up Method refactorings in the history of a project.

Empirical stduies Refactoring detection tools have been used in several empirical studies. For instance, Weißgerber and Diehl [WD06a] used RefVis [GW05] to investigate the error-proneness of refactorings. Kim et al. [KCK11] used the MK refactoring reconstruction technique [KNG07]

to investigate the effect of refactoring on bug fixing. Rachatasumrit and Kim [RK12] used Ref-Finder [PRSK10, KGLR10] to investigate the impact of refactoring on regression testing. Bavota et al. [BCL+12] used Ref-Finder to investigate to what extent refactoring activities induce faults. Bavota et al. [BLP+15] used Ref-Finder to investigate whether refactoring activities occur on code components having poor quality metrics or being affected by code smells. More recently, Palomba et al. [PZODL17] used Ref-Finder to investigate the relationship between different types of code changes (i.e., bug fix, feature addition, and general maintenance) and refactorings. The accuracy of the employed refactoring detection tools is a critical factor affecting the conclusions of these studies. Missing refactorings (false negatives) is a serious threat to the generalizability of empirical studies. Detecting incorrect refactorings (false positives) is even more severe, as it makes the conclusions of the empirical study wrong.

For all the aforementioned use cases, it becomes clear that there is a great need for refactoring detection tools having high accuracy, being able to operate at different levels of change granularity (i.e., commit and release level), and being able to scale enough to analyze a large number of commits over a short period of time (i.e., for the use case of extracting refactoring operations from the entire commit history of a project).

1.2 Problem Statement

While refactoring detection has been a very active line of research [DDN00, APM04, GW05, GZ05, WD06b, DCMJ06, XS06b, PRSK10, KGLR10, FGL12, GDMH12, NCV+13, SV17, GSWMH17], as we will see in Section 2, almost all of the proposed techniques in the literature are not able to detect instances of Rename Local Variable refactoring. Moreover, the few techniques that are able to identify the instances of Rename Local Variable refactoring have several limitations that negatively affect their precision (i.e., what portion of the identified instances are correct) and recall (i.e., what portion of all real Rename Local Variable refactoring instances occurring in the code are correctly identified). For example, REPENT [AEP+14], which is the state-of-the-art tool in detecting Rename Local Variable instances, relies on textual Diff to map lines across the two revisions of the code. The mapped lines are used as input to identify Rename Local Variable refactoring instances. This can cause several problems, particularly when the changes are more radical. The textual Diff in these cases can easily map lines that do not contain renamed variables. Figure 1 demonstrates such a scenario where drastic changes in the location of the source code leads to report an incorrect instance of Rename Local Variable refactoring by REPENT.

In this example, REPENT reports that the local variable `lastModifiedValue` in method `getLastModified()` has been renamed to `creationDateValue` in method `getCreationDate()`.

```
public Date getCreationDate() {
   ...
   if (creationDate.get() instanceof Date) {
      ...
   } else {
      String creationDateValue = creationDate.get().toString();
      ...
      result = formats[i].parse(creationDateValue);
      ...
   }
   ...
}

...

public Date getLastModified() {
   ...
   if (lastModified.get() instanceof Date) {
      ...
   } else {
      String lastModifiedValue = lastModified.get().toString();
      ...
      result = formats[i].parse(lastModifiedValue);
      ...
   }
   ...
}

...

public long getContentLength() {
   ...

   ...
}
```

```
public long getCreation() {
   ...

   ...
}

public void setCreation(long creation) {
   ...

   ...
}

...

public Date getCreationDate() {
   ...
   if (creationDate.get() instanceof Date) {
      ...
   } else {
      String creationDateValue = creationDate.get().toString();
      ...
      result = formats[i].parse(creationDateValue);
      ...
   }
   ...
}

...

public Date getLastModified() {
   ...
   if (lastModified.get() instanceof Date) {
      ...
   } else {
      String lastModifiedDateValue = value.toString();
      ...
      result = formats[i].parse(lastModifiedDateValue);
      ...
   }
   ...
}
```

Figure 1: Drastic relocation of statements leads to a False Positive in REPENT

As it is observed, the method `getLastModified()` still exists in the target revision, and the local variable `lastModifiedValue` defined in the old revision of this method has been renamed to `lastModifiedDateValue` in the new revision. Similarly, the method `creationDateValue()` still exists in the new revision of the code, but notice that the local variable `creationDateValue` has not been renamed. It is clear that REPENT incorrectly detects this Rename Local Variable refactoring instance because it uses textual Diff for finding the renaming candidates, since the textual Diff does not know about the structure of the code and incorrectly maps the lines containing the `lastModifiedValue` and `creationDateValue` local variables solely based on textual similarity.

In addition to such limitations, to the best of our knowledge, none of the proposed techniques is accompanied by an implemented tool. In Section 2, we will discuss the disadvantages of each of the proposed techniques for detecting Rename Local Variable refactoring instances in more depth.

It has been shown that rename refactorings are the most applied refactorings by software developers [MHPB12], and particularly, Rename Local Variable is the second most applied rename refactoring, after Rename Method refactoring [AEP+14]. We argue that detecting Rename Local

Variable refactoring instances is as important as identifying other types of refactorings in the history of software systems, and that's why we are particularly interested in devising a technique for it. Many of the aforementioned reasons for detecting other types of refactorings hold for Rename Local Variable refactoring, e.g., improving *change comprehension*, improving *merging techniques*, improving *bug-inducing commits analysis*, and improving the validity of *empirical studies*. In general, research has paid special attention to rename refactorings to improve the accuracy of Diff and merge algorithms [LAK+17]. Also, as we will see in Chapter 2, there are several works in the literature that aimed at studying rename refactorings [KR11, AEP+14], yet since *no tool* was available to detect the instances of Rename Local Variable, the authors had to build their own rename detection approaches. We will compare our technique with one of these tools in Chapter 4.

1.3 Contributions

Particularly, this thesis makes the following contributions:

- We aim at complementing the current research in refactoring construction by improving the state-of-the-art refactoring detection technique, namely RefactoringMiner [STV16, TME+18] to support the detection of Rename Local Variable refactorings. RefactoringMiner has been successfully used in previous research [STV16], and due to its superior approach in refactoring construction (compared to the existing techniques) it will be certainly used in more future studies. As a result, improving it by supporting more refactoring types will have a great impact. We discuss in depth the reasons we decided to build our algorithm upon RefactoringMiner in Chapter 3. We have improved the core statement matching algorithm of RefactoringMiner, and introduced the necessary rules applied by a post-processing algorithm into RefactoringMiner for detecting Rename Local Variable refactoring instances.

- In order to automate the comparison of the results of one refactoring detection tool to other existing tools, we have designed a comprehensive *accuracy computation tool* based on refactoring detection tool agreement. This tool is able to read the results of multiple refactoring detection tools, build a unified model of the detected refactorings, compare the results of those tools using the populated models, and construct a *reference corpus* of refactorings (i.e., a dataset of refactoring instances that we are certain they have occurred in the commit history of one or more repositories), and report the accuracy of the tools against the constructed reference corpus. The resulting reference corpus can be easily used by other researchers. In addition to Rename Local Variable refactoring, this tool models a wide variety of other refactoring types. It is possible to extend this tool to support more detection tools and more refactoring types. This tool has been successfully used in other research works [TME+18] for comparing

the precision and recall of competitive refactoring detectors. More details about this tool are given in Chapter 5.

- We make available the most complete reference corpus for the Rename Local Variable refactorings to date. All these refactorings have been manually investigated by the author of this thesis and an independent researcher.

1.4 Thesis Organization

There have been many approaches introduced in the literature for refactoring construction, and we discuss the most important contributions that we are aware of in Chapter 2. Chapter 3 is dedicated to explaining how RefactoringMiner works and how the proposed approach fits into it, including our improvements on the RefactoringMiner's statement matching algorithm, the proposed rules, and the introduced post-processing algorithm for detecting instances of the Rename Local Variable refactoring. In Chapter 4, we present the accuracy of RefactoringMiner and compare it with the state-of-the-art tool in detecting rename refactorings, namely REPENT, and discuss about the results of the comparison. In Chapter 5, we provide details about our *accuracy computation tool* that we call RefBenchmark, which we used to compare the results of our approach with REPENT. Chapter 6 concludes the thesis and discusses some possible future works.

Chapter 2

Related Work

There are various approaches for identifying refactoring operations in the source code. Some approaches perform *live* refactoring detection by analyzing the edits a developer is performing on the source code while working in the IDE (Section 2.1). Other approaches perform *post-mortem* refactoring detection by analyzing the changes that end up in the repository after a commit (Section 2.2). Finally, we present approaches specialized in identifying Rename Local Variable refactorings, and compare them with our proposed solution (Section 2.3).

2.1 Refactoring Detection in the IDE

The proposed technique in this thesis focuses on identifying the instances of rename local variable refactorings by analyzing the source code history of software systems. However, any type of refactoring can be also detected by monitoring changes performed on the code within the IDE. This allows *live* detection of the refactorings. It is important to study these approaches to understand the advantages that they can provide and their limitations, and to compare the techniques used in inferring the refactorings with our approach.

In general, there are two categories of tools that identify refactorings in the IDE. The first category includes tools that identify *complete refactorings*, while the second category includes the techniques that identify *incomplete refactorings* (i.e., the refactorings that are just initiated by the developer), and then suggest to the developer to automatically complete the refactoring. In the following subsections, we present these techniques in more detail.

2.1.1 Identifying Complete Refactorings in IDE

Negara et al. extend CodingTracker [NVC+12], a tool that translates fine-grained code edits (e.g., typing characters) to AST node operations, i.e., *add*, *delete*, and *update* AST nodes, to aggregate a sequence of these operations to infer high-level changes, and then refactorings [NCV+13]. The tool supports identification of the following refactorings: *Encapsulate Field*, *Rename Class*, *Rename Field*, *Rename Method*, *Convert Local Variable to Field*, *Extract Constant*, *Extract Method*, *Extract Local Variable*, *Inline Local Variable*, and *Rename Local Variable*. The tool was used to study whether refactorings end up in the version control systems, or whether developers use automated tools for applying refactorings.

Ge and Murphy-Hill proposed GhostFactor [GMH14], which identifies manually-performed refactorings in the IDE and checks them against preconditions to make sure that the behavior of the code remains the same after applying the refactorings. GhostFactor listens to the events in the IDE and saves snapshots of the edited files. The approach compares the latest snapshot with a given number of previous snapshots and parses them to their ASTs. GhostFactor can be fed with different refactoring detection approaches that work with the ASTs. The authors provide their own algorithms for detecting three kinds of manual refactorings, namely *Extract Method*, *Change Method Signature*, and *Inline Method*. The algorithms look merely at the changes in the ASTs, and uses thresholds to detect similar entities involved in the refactorings.

Ge at al. introduced ReviewFactor [GSWMH17], a tool that allows code reviewers to distinguish refactorings and other changes in the code. ReviewFactor first mines IDE logs for detecting automated refactorings that have occurred on older versions of the code. Then, it replays those refactorings on the version of the code before the code review, to create an intermediate version of the code. Then, ReviewFactor compares this intermediate version with the version under code review to find manual refactorings done on the code. ReviewFactor extends GhostFactor by adding *software entity mapping*, a technique for detecting added/removed/changed entities across the two versions of the code. The mapped entities between the two versions are the ones that have the same entity type (for example, they are both class declarations) and their respective parents are a pair of mapped entities.

After mapping the elements across the two versions, ReviewFactor builds a *mapping tree* where pairs of mapped entities are nodes, which are hierarchically organized to reflect the parent-child relationship of the entities in the original source code. These mapping trees are then compared to subtrees that model refactorings to detect refactoring candidates. The approach then computes the ratio of the text distance and the average length of the entities' underlying source code in the refactoring candidates. This indicates that the changed software entities in a subtree that models a refactoring instance are cohesively related. If the computed ratio is lower than a threshold, the

changed entities are reported as a refactoring. ReviewFactor supports five refactoring types: *Rename Type*, *Rename Method*, *Move Method*, *Extract Method* and *Inline Method*.

2.1.2 Automated Refactoring Completion in the IDE

BeneFactor [GDMH12] "automatically detects an ongoing manual refactoring [within the IDE], reminds the developers that automatic refactoring is available, and can finish the manual refactoring after the developer's explicit invocation, without requiring her to undo any code changes". The authors conducted a formative study and observed how developers refactor code in a lab setting. They discovered different *patterns* or *workflows* for some refactoring types. BeneFactor detects refactorings by monitoring changes in the IDE, after certain events (e.g., save), and compares the sequence of events with the discovered workflows. The more the sequence of the changes is similar to a refactoring workflow, the higher the confidence of the sequence of changes to be actually a refactoring. When this confidence is higher than a pre-selected threshold, BeneFactor tells the developer that the change might be a refactoring and suggests to complete it. BeneFactor supports the following types of refactoring: *Rename Field*, *Extract Method*, *Extract Constant*, *Extract Local Variable*,*Inline Local Variable*, *Introduce Parameter*, *Change Method Signature*, and *Pull Up Field*.

Similarly, WitchDoctor [FGL12] tries to identify the sequence of changes that can lead to a refactoring and suggest auto-completion for them. The approach first monitors the events in the IDE, including low level keystrokes, and matches a sequence of these events to an AST node operation (for example, a sequence of code changes is translated to insertion of a method invocation in the AST). WitchDoctor avoids using an AST diff approach since, during code development, in many situations the code is not parsable until the developer finishes the changes. The approach then uses a pattern matching algorithm to find patterns in the AST node operations, to detect possible refactorings. The refactorings are defined using a declarative specification, similar to Ref-Finder [PRSK10, KGLR10] (discussed in Section 2.2). When a matching pattern is found, and the pattern contains enough information for a refactoring to be done, WitchDoctor suggests to the developer to complete the refactoring. If the developer chooses to complete the refactoring, WitchDoctor rolls back the changes manually done by the developer leading to a refactoring, and invokes the IDEs refactoring engine to complete the refactoring. The supported refactorings are *Rename Local Variable*, *Rename Field*, *Rename Class*, *Rename Method*, *Extract Local Variable*, and *Extract Method*.

2.2 Refactoring Construction From the Change History

2.2.1 Demeyer et al. [DDN00]

Demeyer et al. [DDN00] intention behind refactoring detection was to help reverse engineers to understand how and why software has evolved. Their approach uses four heuristics based on changes in metrics (e.g., method size, class size, and inheritance) for finding refactorings across different versions of the software. The approach calculates the values for these metrics for the modified parts of the source and target revisions and assesses the difference between the computed values. The approach is able to find four classes of refactorings, namely *Split into Superclass/Merge with Superclass*, *Split into Subclass/Merge with Subclass*, *Move [functionality] to Other Class (Superclass, Subclass or Sibling Class)*, and *Split Method/Factor Out Common Functionality*. The authors evaluate their approach against three software systems (namely, *Visualworks*, *HotDraw*, and *Refactoring Browser* which are developed in *Smalltalk*), and show it is capable of detecting refactoring instances, and discuss the false positives that are found.

In contrast to our approach, Demeyer et al. [DDN00] only use metrics to find refactoring instances, therefore, this approach needs to specify thresholds. Metrics are not always good representatives for complex changes in the source code. For example, if the same number of classes are added and removed across two revisions, the metric-based approach can fail in correctly determining the change, and thus, it can miss some of the refactorings, while our AST-based approach can detect them. In addition, the authors mention that their approach can fail in the presence of *rename refactorings* [DDN00], while our technique is immune to this problem.

2.2.2 Van Rysselberghe and Demeyer [RD03]

Van Rysselberghe and Demeyer [RD03] argue that we can improve the approaches of building software by studying the evolution of successful software systems. Having this in mind, the authors study the *move method refactoring* on the history of a large system, namely *Tomcat*. The authors use clone detection to identify the methods which are moved across different revisions of the system.

The proposed approach can only find one type of refactoring. Moreover, the accuracy of the proposed approach depends on the accuracy of the employed clone detection technique. All clone detectors have several configuration options, e.g., the percentage of similarity that two clone pairs should have to be reported as clones, which is defined as a threshold. This makes the entire approach threshold-based.

2.2.3 Antoniol et al. [APM04]

Antoniol et al. [APM04] propose an approach, inspired from Information Retrieval techniques, to detect changes in the classes across versions of a system, e.g., when a class is replaced with another one, or when it was split into two classes, or when two classes were merged into one. Inspired from the Vector Space Model in information retrieval, the authors represent classes in two revisions of the code as vectors of the *TF-IDF* weights of the identifiers appearing in the classes. The similarity between the classes is calculated using the *cosine similarity* of the vectors corresponding to the classes, and it is used for identifying class-level changes. For example, if a class `A` is split into two classes `A'` and `B`, the vectors `A` and `A' + B` should be similar, i.e., the cosine of the angle between them should be greater than a threshold. The approach was validated against 40 releases of an open-source project, namely *dnsjava*.

2.2.4 Godfrey and Zou [GZ05]

Godfrey and Zou [GZ05] use *origin analysis* for detecting split and merge transformations between two revisions of source code, at the function and file levels. To detect these changes, the approach first analyzes functions and extracts their attributes (e.g., LOC, the number of variables, etc.) and relationships (callers and callees of each function) in both revisions. The approach then finds similar entities in the two versions to detect split/merge transformations based on the extracted attributes and relationships. The authors use a linear ranking mechanism and the user can examine the best entity matches. Finally, by conducting a study on *PostgreSQL*, the authors show that the presented approach can detect files and functions split/merge activities throughout the history of the software. In this case study, the found *merge* transformations including *Full and Partial Clone Elimination*, *Service Consolidation* (i.e., when two or more functions that perform different services but called at the same time by the same clients are merged into a new, larger function), and *Parameterization* (i.e., when two similar functions are combined into a new function by adding a parameter to distinguish different functionalities). In addition, the *split* transformation which was found in the case study was a *Pipeline Expansion* (i.e., when a function is broken down into two, each of which accepting the same input as the original method and generating part of the output that it used to generate).

Although our work is similar to this study in the sense that both aim at finding refactorings by analyzing source code repositories, the proposed approach by Godfrey and Zou is completely different from ours. The authors use code attributes and metrics to find refactoring instances while we employ an AST-based approach. Moreover, our approach does not need any human intervention in the detection process.

2.2.5 Weißgerber and Diehl [WD06b]

Weißgerber and Diehl [WD06b] proposed a technique where refactoring activities are found by first identifying candidates whose pairs of code elements, like classes and methods, have similar signatures. Then, for each refactoring candidate, the approach uses a clone detection tool, namely CCFinder [KKI02], to compare the bodies of the code elements.

The authors manually inspected the commit messages of two open-source projects to find documented refactorings to compute the recall, and used random sampling to estimate the precision of their approach. Their technique is able to detect the following refactorings: *Rename Method*, *Hide or Unhide Method* (which happens when the visibility of a method within a class has become more or less restrictive), *Add or Remove Parameter*, *Move Class*, *Move Interface*, *Move Field*, *Rename Class*, and *Move Method*.

As mentioned, using a clone detector in a technique means that the approach is threshold-based, and will be sensitive to the configurations of the used clone detector. Moreover, it has been shown that using commit logs for identifying refactorings is not reliable, since developers usually do not mention refactoring activities in commit log messages [MHPB12]. As a result, there could be many refactorings that are missed due to the use of commit logs for constructing the ground truth (i.e., the actual refactorings that occurred in the source code).

2.2.6 Xing and Stroulia [XS06b]

Xing and Stroulia [XS06b] argue that finding refactorings in the history of the code is the best way to understand the software design rationale. They introduce an approach (developed in a tool called JDEvAn [XS08]) to detect refactorings based on the UMLDiff algorithm [XS05]. The tool first extracts facts from the source code and creates a structural model of the system and stores then in a PostgreSQL database. Then, UMLDiff is used to analyze the extracted facts from two versions of a system to detect the differences. UMLDiff uses similarity thresholds for mapping entities from one revision to another to detect the entities that are moved, renamed, added, removed, or remained unchanged in the system. These differences are also stored in the database, and are queried to detect the refactoring instances.

The authors conducted two case studies on *HTMLUnit* and *JFreeChart*. The results show that JDEvAn finds all the documented refactorings. The identified refactorings are *Convert Top-Level Type to Inner*, *Move Subsystem/Package/Class*, *Pull-Up Method/Field/Behavior/Constructor Body*,

Push-Down Method/Field/Behavior, *Move Method/Field/Behavior*, *Rename Subsystem/Package/-Class/Method/Field Add/Remove Parameter*, *Information Hiding*, *Generalize/Downcast Type*, *Extract/Inline Subsystem/Package/Class/Subclass/Superclass/Method*, *Extract Interface*, *Form Template Method*, *Replace Inheritance with Delegation and the reverse*, *Die-hard and Legacy Classes*, *Convert Anonymous Class to Inner*, *Introduce Factory Method*, *Introduce Parameter Object*, *Encapsulate Field*, and *Preserve Whole Object*.

While we use a similar approach in modeling the revisions of software systems, our detection approach is different from JDEvAn. JDEvAn uses UMLDiff, and UMLDiff is threshold-based, but we avoid any threshold in our technique. Also, UMLDiff is known to have problems in detecting move or rename method changes [XS06b].

2.2.7 RefactoringCrawler (Dig et al. [DCMJ06])

Dig et al. [DCMJ06] proposed RefactoringCrawler for detecting refactoring instances in the code, with the goal of identifying changes in the components' interfaces. The authors argue that by detecting the applied refactorings in the component's source code, we can automatically update the clients which depend on the component to comply with the new interface. The detection process consists of two major phases, namely *syntactic* and *semantic* analysis.

In the syntactic analysis phase, RefactoringCrawler parses the source code into a lightweight AST, where the parsing stops at the declaration of methods and fields in classes. Then the approach builds a tree, in which the nodes represent source-level entities (e.g., packages, classes, methods, and fields). Nodes are arranged hierarchically in the tree based on their fully qualified names (e.g., `package.Class` is a child of the node `package`). The nodes are later connected using the references between the entities, converting the tree to a graph.

Next, inspired from Information Retrieval techniques, *Shingle Encoding* [Bro97] is used to find refactoring candidates based on the entities' similarity in the graph. *Shingles* are "fingerprints" for strings (e.g., method bodies) and enable the detection of similar code fragments much more robustly than the traditional string matching techniques that are not immune to small deviations like renamings or minor edits.

Then, for each candidate pair of the matched entities, the *semantic analysis* detects those pairs that are refactored using seven strategies that are based on the similarity of references in the graph (e.g., method calls, imported packages, and method parameter(s)).

The evaluation of this approach shows that it can find more than 85% of the applied refactorings on real-world components. RefactoringCrawler is able to detect the instances of seven refactoring types, namely *Rename Package*, *Rename Class*, *Rename Method*, *Pull-Up Method*, *Push-Down Method*, *Move Method*, and *Change Method Signature*.

Similar to our approach, RefactoringCrawler uses an AST-based approach for analyzing the source code; however, it only identifies refactorings which can change the way a component can be accessed (i.e., public API changes). Moreover, using Shingles might lead to miss some refactorings, as the similarity measure needs a threshold. This is also the same for the similarity calculated between the entity references in the semantic analysis. In addition, RefactoringCrawler needs the project revisions to be built for the analysis. In practice, this is a challenging task, given that usually only a small portion of the change history of software systems can be successfully compiled [TPB+17].

The precision of RefactoringCrawler was calculated by manually looking at the reported refactorings. To compute the recall of RefactoringCrawler, the authors manually investigated the release notes of three projects to discover the actual refactorings that occurred in the source code, and see whether their approach is able to find them. However, this might lead to an incomplete reference corpus. It has been shown that only 21% of the release notes contain information about the refactorings, and they are usually general statements specifying the refactored components, and do not mention the types of refactoring performed [MBP+17].

2.2.8 RefacLib (Taneja et al. [TDX07])

One of the issues of RefactoringCrawler is that the approach relies on the references between the entities in the source code, e.g., method calls. In an API, there could be a large number of methods that are not called by other methods within the same system, since they are public interfaces for the API that are supposed to be used by the API clients. As a result, RefactoringCrawler cannot detect refactorings that occurred for these methods.

To solve this problem, Taneja et al. [TDX07] proposed RefacLib, which improves RefactoringCrawler by using a *heuristic-based* analysis, replacing the semantic analysis done by RefactoringCrawler. RefacLib gathers facts from the source code and Javadoc comments, and computes similarity measures to assign an overall score that reflects the likelihood of a candidate to be a refactoring. As an example of a heuristic, RefacLib looks at the names of the two potentially matching entities, by breaking down them to their subparts (e.g., `performUpdates` to `perform` and `Updates`), and assigning a similarity score between the two sets of subparts. The score is computed based on different measures that reflect, for example, whether there is a complete match between the subparts, or between the synonyms of the subparts.

RefacLib supports the following refactorings: *Change Method Signature*, *Rename Class*, *Push Down Method*, *Rename Package*, *Rename Method*, *Pull-Up Method*, and *Move Method*. By running RefacLib on the same subject systems that RefactoringCrawler was evaluated with, it was observed that RefacLib performs generally better than RefactoringCrawler. However, both approaches suffer from the fact that there is a need for defining thresholds.

2.2.9 ChangeDistiller (Fluri et al. [FWPG07])

Fluri et al. propose ChangeDistiller to identify changes across different versions of the source code. ChangeDistiller augments a tree differencing algorithm, proposed by Chawathe et al. [CRGMW96], to adapt it to the changes that can take place in the source code ASTs. To match two leaf nodes of two ASTs, ChangeDistiller uses *bi-gram* similarity between the source code corresponding to the leaf nodes. Two leaf nodes are matched if this similarity is greater than a threshold. For the inner nodes, ChangeDistiller computes the number of matched leaves in the subtrees rooted under each inner node, normalized by the maximum number of leaves in these subtrees. If this ratio is greater than a threshold, the inner nodes are marked as matched.

The output of this step is a set of basic tree edit operations (e.g., added/removed/moved/updated nodes) which are stored in a database. ChangeDistiller has a taxonomy of more abstract source code changes based on these basic edit operations [FG06], which are extracted by querying the database.

The authors used three open-source projects to examine the accuracy of ChangeDistiller in correctly identifying source code changes. They fed ChangeDistiller with the source code of these projects from different revisions, having configured ChangeDistiller with the original configuration. A subset of the results was selected and two users looked at them independently to manually classify the changes in them.

ChangeDistiller is able to detect several types of changes in the code (e.g., when the parent of a statement is changed), but it can also identify some simpler refactorings, e.g., *Rename Method*, *Rename Parameter*, and *Return Type Change*. However, it cannot detect the instances of *Rename Local Variable*, and it still needs to be configured using thresholds.

2.2.10 Ref-Finder (Prete et al. [PRSK10, KGLR10])

Prete et al. [PRSK10, KGLR10] introduced Ref-Finder to overcome the limitations of previous approaches in detecting refactorings in the history of software systems. Ref-Finder is able to detect *atomic* and *complex* refactorings. Refactorings like *move method* or *move attribute* are atomic, while complex refactorings are the ones that consist of atomic refactorings. Ref-Finder supports most of the refactorings (63 of 72) introduced by Fowler [Fow99]. The complete list of refactorings and their corresponding detection rules is given in a technical report [PRK10]. Among these refactorings, *Rename Local Variable* is missing.

Ref-Finder models the versions of the software as logic predicates that represent code entities, their containment relationships, and structural dependencies. Refactorings are encoded as logic rules. For some rules, the similarity between two methods is needed, which is defined using a threshold.

Querying this model can reveal refactorings that occurred across the two versions of the system.

To evaluate Ref-Finder, the authors run it on the examples of the Fowler's refactoring book, and two software systems. To find the correct refactorings, the authors run Ref-Finder with a low threshold and manually validated the reported refactorings. The authors then run Ref-Finder with a higher threshold and compare the two sets of refactorings to compute the recall. The precision is calculated by manually validating a random set of reported refactorings.

The authors report that Ref-Finder has a high precision and recall (74% and 96%, respectively). However, later studies have reported a much lower accuracy for Ref-Finder (precision of 27% [KHFG16] and 35% [SGMHJ13], recall of 24% [SGMHJ13]). This could be explained by the fact that the reference corpus used for evaluating the performance of Ref-Finder was created using the tool itself (with a lower threshold). The tool may miss cases if the used algorithm has problems, and as a result the reference corpus will not be complete.

Another major limitation of this approach is due to the definition of the logic rules. Some of the rules are too general and therefore might introduce several false positives. Moreover, Ref-Finder needs the versions of the systems under analysis to be built, and this is not practical due to the small portion of the commits that can be built [TPB+17].

2.2.11 RefDiff (Silva and Valente [SV17])

Silva and Valente [SV17] proposed RefDiff, which uses heuristics based on static analysis and code similarity to detect 13 refactoring types on consecutive revisions of the code. These refactoring types include *Rename Type*, *Move Type*, *Extract Superclass*, *Rename Method*, *Pull-Up Method*, *Push-Down Method*, *Move Method*, *Extract Method*, *Inline Method*, *Pull-Up Field*, *Push-Down Field*, and *Move Field*.

First, RefDiff parses the revisions of the project analysis to generate higher level models from them, which capture the information about the source code entities. The entities of interest are types, methods, and fields. Then, RefDiff tries to find relationships between the entities of the two revisions (e.g., whether two types or methods are the same). Some of the relationships need the involved entities to be similar to a certain percentage (in other words, more than a certain threshold). RefDiff represents a source code fragment as a bag of tokens, and computes the similarity of the code entities using a variation of the TF-IDF weighting scheme (similar to the Antoniol et al.'s approach [APM04], while Antoniol treats *identifiers* in classes as "words", here *tokens* are treated as words).

To determine the similarity threshold values, and to achieve the best compromise between the precision and recall, the authors applied a calibration process on a randomly-selected set of ten commits from ten different open-source projects, for which the applied refactorings are known and

have been confirmed by the project developers themselves [STV16]. The authors evaluated the accuracy of their tool using a reference corpus of seeded refactorings applied by graduate students in 20 open-source projects.

Similar to RefDiff, our approach also relies on detecting differences in the entities before and after refactoring, by creating an abstract model of the system. However, we do not rely on any similarity threshold for detecting the instances of *Rename Local Variable* refactoring (or any other type of refactoring).

2.3 Rename Local Variable Refactoring

As discussed in the introduction of this thesis, detecting the instances of *Rename Local Variable* refactoring has several applications. However, from the mentioned techniques for detecting refactoring operations in the history of software systems, only WitchDoctor [FGL12] and Negara et al.'s approach [NCV+13] (which is based on CodingTracker [NVC+12]) support detecting the instances of *Rename Local Variable* refactoring. Unfortunately, both these techniques detect refactorings within the IDE, and not using the change history of software systems.

However, there are other works in the literature where a technique for identifying the instances of renaming in identifiers in general, and Rename Local Variable refactorings in particular, from the history of systems was required as a means to study different phenomena (e.g., merging in version control systems, or empirical studies on how developers rename code entities). Here, we briefly discuss these works.

2.3.1 RENAMING DETECTOR (Malpohl et al. [MHT00, MHT03])

Malpohl et al. [MHT00]) proposed one of the first approaches for detecting renamed identifiers in the source code to facilitate the merging of files in version control systems. The approach, called RENAMING DETECTOR, first parses the input files to ASTs. The def-uses of the identifiers is then detected by employing Symbol Analysis, similar to what a compiler does when generating symbol tables. The approach then identifies sequences of tokens in the old version of the system that map to a sequence of tokens in the new version. Then, the approach compares the identifiers in the two versions by comparing the token sequences around the identifier definitions. Three kinds of similarities are considered, "declaration similarity" (i.e., whether the tokens surrounding an identifier in a variable declaration or method signature match in the two versions), "implementation similarity" (i.e., how much the sequence of tokens in the bodies of two methods are similar), and "reference similarity" (i..e, whether the token sequences around the *uses* – or references – of an identifier are similar). Finally, the approach uses an expert system to weight the relevance of each similarity for each possible pair

of identifiers to find the pairs that correspond to one another. The authors, however, do not provide information on how the rules of the expert system are defined, but it appears that the expert system uses thresholds on the three mentioned similarity measures to decide about whether a paired match is actually a rename or not.

2.3.2 DiffCat (Kawrykow and Robillard [KR11])

Kawrykow and Robillard [KR11] argue that while change-based differencing approaches can provide much more useful information in comparison to text-based methods, there is still room to improve change-based approaches. The authors discuss *non-essential changes*, i.e., minor code changes which can affect the accuracy of the existing change-based differencing approaches. For example, the authors consider *Rename-Induced Modifications* as non-essential changes, e.g., the renamings applied on method invocations due to renaming the invoked method. Other non-essential changes include *Trivial Type Updates*, *Local Variable Extractions*, *Trivial Keyword Modifications*, *Whitespace and Documentation-Related Updates*, and *Local Variable Renames*. Such changes decrease the accuracy of the change-based differencing approaches.

To help in alleviating the effect of non-essential approaches in differencing approaches, Kawrykow and Robillard introduce DiffCat to detect non-essential changes in the repository of a software system. DiffCat uses the infrastructure provided by SemDiff [BR09] (a change analysis tool for studying framework evolution) to get a set of files changed in two software revisions. DiffCat uses a Partial Program Analyzer to resolve the type bindings of the expressions in the ASTs of the changed files. It then uses ChangeDistiller [FWPG07] to identify structural changes across the ASTs and augments the identified changes with various methods. For example, ChangeDistiller does not identify all the instances of the *rename field* refactoring, particularly when the textual similarity between the fields after refactoring is not high, and DiffCat attempts to improve this by iterating over all the reported insert-delete pairs in each class and checking whether all the references to a field are consistently replaced with the renamed identifier. The authors, unfortunately, do not explain how exactly DiffCat detects the instances of the *Rename Local Variable* refactoring in this step.

After detecting the renamed identifiers, DiffCat rolls them back (i.e., it replaces them with the original names), with the intuition that renames affect the detection of other non-essential changes. DiffCat finally runs ChangeDistiller again on the rolled-back version to find non-essential changes.

The authors conducted an empirical study on seven open source systems and showed that between 2.8% and 22.9% of the changed lines contain non-essential changes, and most of the non-essential differences were induced by rename refactorings or trivial updates involving `this` keyword.

2.3.3 REPENT (Arnaoudova et al. [AEP+14])

Arnaoudova et al. [AEP+14] argue that identifier renaming can improve the understandability of the source code, and therefore, they conduct a survey on identifier renaming. The authors ask various questions regarding renaming identifiers from 71 developers, out of which some were the developers of five Java software systems. The study finds that:

1. 39% of the developers perform renaming activities at least few times a week.
2. Developers rename identifiers along with other development activities such as changing the functionality, performing other refactoring, and adding new functionality.
3. 35% of the developers believe that renaming has a cost and requires time and effort. Also, 32% of the participants consider that the renaming is a costly activity depending on a particular case.
4. Developers postpone the renaming activity due to different reasons (e.g, its potential impact on other systems or the possibility of introducing bugs).

Based on the results of this study and the fact that a small percentage of identifier renamings are documented, the authors propose an approach, namely REPENT, to identify the instances of identifier renaming across the revisions of a software system. REPENT automatically classifies the detected renaming based on a taxonomy that comprises four dimensions:

Entity kinds what entity is being renamed, e.g., a type or a local variable,

Form of renaming whether the renaming is *simple* (i.e., change applies to one word), *complex* (i.e., changes apply on more than one parts in the identifier name), *formatting only* (i.e., changes in the letter cases or letter separators), and *term reordering* (i.e, changes in the positions of the terms in the identifiers).

Semantic change e.g., whether the renaming changes the meaning of the previous name or preserves it,

Grammar change e.g., part of speech changes, like change the word from singular form to plural.

To find the instances of identifier renaming in the source code, REPENT compares each modified file in the consecutive software revisions by applying Unix context *diff* algorithm. The result of this comparison is a mapping between the source lines of code in the old and new files. REPENT then maps the entities in these files using the Abstract Syntax Trees and the line mapping from the previous step. This leads to creating an initial set of candidate renamings for each file.

Next, REPENT tries to eliminate false positives from this list of candidates. To do so, it calculates a similarity score for each candidate pair using their def-uses, and removes the candidates for which the similarity score is below a threshold. For each of the candidates, all the def-use statements corresponding to the first identifier (i.e., in the old file) are compared with all the def-use statements of the second one, using the normalized Levenshtein edit distance. A mapping between the def-use statements of the first and second identifiers that maximizes the sum of the similarity scores between the statements is chosen, and this sum reflects the similarity score for the candidate rename refactoring.

After filtering out false positives and forming the final set of candidates, REPENT classifies the renames using the aforementioned taxonomy. It splits the names of camel case candidate identifiers (e.g, `isValid` is spitted into two parts: `is` and `valid`). Then, REPENT tries to map each part of the first and second identifiers, using WordNet and the Stanford Part-of-Speech Analyzer to fulfill the classification.

To evaluate REPENT, the authors test it against five Java open-source projects. The results show that REPENT can find the rename identifier instances cases with a precision and recall of 88% and 92%, respectively.

2.4 Limitations of Current Approaches

The mentioned approaches for detecting refactorings in the IDE have the advantage that they can find refactorings that do not end up to the version control systems, i.e., *shadowed* refactorings. A study showed that a large amount of changes are in fact shadowed by other changes [NVC+12]. As mentioned, several studies also showed that the refactorings can also interleave other changes, which will negatively affect the performance of the approaches that rely on the history of systems to detect refactoring activities [MHPB12, NCV+13, STV16].

At the same time, all these techniques need special plug-ins for each IDE to be applicable. This drastically limits the scenarios where they are usable. Any approach that uses the history of the software for detecting instances of refactorings, including our approach, does not rely on any specific IDE, which makes it useful in a broader range of applications.

The approaches that were discussed in Section 2.2 and find refactorings in the change history of a software system, all use some kind of similarity thresholds, which are usually tedious to calibrate, and might not be generalizable. Unlike these techniques, our proposed approach does not require similarity thresholds.

In addition, some of these techniques (e.g., Ref-Finder [PRSK10, KGLR10]) require the software systems under analysis to be fully-built. As mentioned, it has been shown that in practice

only a small portion of commits can be built in the history of software systems [TPB+17]. Some other approaches rely on Partial Program Analyzers (e.g., DiffCat [KR11]), of which the accuracy is affected in the absence of external dependencies. Our approach, in contrast, does not need the system to be built or the type bindings of the expressions and works only based on information collected from the Abstract Syntax Trees of the system versions under analysis.

Since REPENT is considered the state-of-the-art technique in detecting instances of Rename Local Variable refactorings in the history of a software system, we evaluated the accuracy of our approach against that of REPENT, as it will be discussed in Chapter 4.

2.5 Chapter Summary

In this chapter, we looked at the existing approaches for detecting refactorings in the evolving code, and discussed some of the problems that they have. In the next chapter, we describe our approach in identifying the instances of Rename Local Variable refactorings in the change history of systems.

Chapter 3

Approach

Our approach in detecting instances of Rename Local Variable refactorings has been implemented on top of RefactoringMiner [TME+18], a tool that allows to detect various types of refactorings in the history of software systems (excluding Rename Local Variable refactorings). There are several reasons justifying the selection of RefactoringMiner as the basis of our approach, including:

1. RefactoringMiner provides a complete infrastructure for cloning Git-based repositories and includes a rich API to work with consecutive revisions of the repository under analysis.

2. It provides an abstract model of the system under analysis and detects high and low-level changes in that model. The refactorings are inferred by analyzing the detected changes. The model frees us from dealing directly with abstract syntax trees, while it still allows us to have access to AST nodes where needed. This means that it is rather easy to define new detection rules for refactoring types which are not yet supported.

3. It does not depend on any specific IDE to be executed, and can be used as a standalone program, in contrast to several other refactoring construction approaches, such as Ref-Finder that depends on the Eclipse IDE.

4. The model generation and differencing are fast and scalable. We will discuss the scalability of our approach in Chapter 4.

5. RefactoringMiner has been successfully used in previous research for conducting a large-scale empirical study to understand why developers perform refactorings in their code [STV16], the impact of refactoring on code smells [CGM+17], and the impact of refactoring on quality attributes [CFF+17].

We have done several enhancements on RefactoringMiner to adapt it with our needs for detecting instances of Rename Local Variable refactorings, which we will discuss in the following subsections. First, we will briefly show how the core of RefactoringMiner works.

3.1 Background

3.1.1 RefactoringMiner

RefactoringMiner accepts a Git-based repository of a Java software system and checks out[1] consecutive revisions of its source code. In each of these two consecutive revisions (i.e., a commit and its parent in the directed acyclic graph that models the commit history in Git repositories), RefactoringMiner inspects only the added/removed/changed files to detect the refactoring operations that occurred. This is in contrast to other existing refactoring detection approaches, such as RefFinder [KGLR10], RefactoringCrawler [DCMJ06], and JDEvAn [XS08], where the analysis includes all the files in two snapshots/revisions of a Java project. This improves significantly the efficiency of RefactoringMiner.

Each of the added/removed/changed files across the two revisions are then parsed to their ASTs using Eclipse's JDT parser. JDT uses a Partial Program Analyzer and is able to parse Java files that might even have compiler errors. In addition, JDT parser is able to resolve type bindings for each of the expressions in the generated ASTs. Type bindings can drastically empower any static source code analysis technique. For example, in the context of refactoring identification in software repositories, we need to understand which entities in the source code (e.g., source code statements) are added or removed, and which ones have remained intact or slightly changed, i.e., the *mapped* entities. The *mapped statements*, for example, might be found much easier by looking at the type bindings of the composing expressions first. In other words, if two statements are composed of expressions with different type bindings, they are most likely non-matching statements, rather than corresponding to the same statement that has been modified between the two revisions.

However, to completely resolve type bindings within the source code, the parser should be aware of the location of all the external dependencies of the project, in addition to the Java runtime library files (i.e., it should have a *complete environment*). Particularly for external dependencies, they should be compiled (or downloaded pre-compiled as `jar` files). This is usually done automatically in the presence of a build system (e.g., Maven) when compiling the source code, which is usually a time-consuming task.

When analyzing the history of software systems, compiling each revision to have a complete

[1]Since check-out is an expensive operation, we are currently working on improving the efficiency of RefactoringMiner by using Git's internal representation of the files, i.e., the Blobs.

environment for binding-aware parsing is not always feasible. Recent studies have shown that only a small portion of commits can be successfully built [TPB+17], as mentioned before.

As a result, RefactoringMiner is designed to operate without resolved type bindings. This extends the applicability of RefactoringMiner to a much wider spectrum, such as the detection of refactorings on partial code, the detection of refactorings at commit time, etc.

In each revision r, RefactoringMiner extracts the following entities from the source code:

- TD_r: The set of type declarations (i.e., classes, interfaces, enums) which are changed in r. For a child commit, this set includes the type declarations inside the changed/added Java files, while for a parent commit, this set includes the type declarations inside the changed/removed Java files. Each element td of the set is a tuple of the form (p, n, F, M), where p is the parent of td, n is the name of td, F is the set of fields declared inside td, and M is the set of methods declared inside td. For a top-level type declaration p corresponds to the package of the compilation unit td belongs to, while for a nested/inner type declaration p corresponds to the package of the compilation unit td belongs to concatenated with the name of the type declaration td is nested under.

- F_r: The set of fields inside the type declarations of TD_r. It contains tuples of the form (c, t, n), where c is the fully qualified name of the type declaration the field belongs to (constructed by concatenating the package name p with the type declaration name n), t is the type of the field, and n is the name of the field.

- M_r: The set of methods inside the type declarations of TD_r. It contains tuples of the form (c, t, n, P, b), where c is the fully qualified name of the type declaration the method belongs to, t is the return type of the method, n is the name of the method, P is the ordered parameter list of the method, and b is the body of the method (could be `null` if the method is abstract or native).

- D_r: The set of all directories in r as returned by command `git ls-tree`. Each directory is represented by its path p.

The body of a method is represented as a tree capturing the nesting structure of the code, where each node corresponds to a statement, similar to the representation used by Fluri et al. [FWPG07]. The statements are divided into two categories: *composite statements* and *leaf statements*.

Composite statements are the statements that contain other statements within their body, such as `for`, `while`, `do-while`, `if`, `switch`, `try`, `catch`, `synchronized block`, and `label`. Composite statements are represented as parent nodes in the tree representing the body of a method,

and contain the statement's type and the expression(s) appearing within parenthesis before the statement's body.

Leaf statements are statements that do not have a body. Leaf statements are represented as leaf nodes in the tree representing the body of a method, and contain only the statement itself.

RefactoringMiner avoids keeping ASTs in the memory, since ASTs can be quite large. Instead, for each statement/expression RefactoringMiner keeps its string representation in a pretty-printed format where all redundant whitespace characters are removed. In addition, an AST Visitor is used to extract all variable identifiers, method invocations, class instantiations, variable declarations, types, literals, and operators appearing within each statement/expression. The extracted expressions are stored in a pretty-printed format within the corresponding statement node.

Figure 2 shows the tree-like representation of the body of method `createAddresses`, along with the information extracted by the AST Visitor for two of its statements. Note that `try` and `catch` blocks are treated as sibling nodes, while in the actual AST representation `catch` is a property of `try` (i.e., `catch` belongs to `try`).

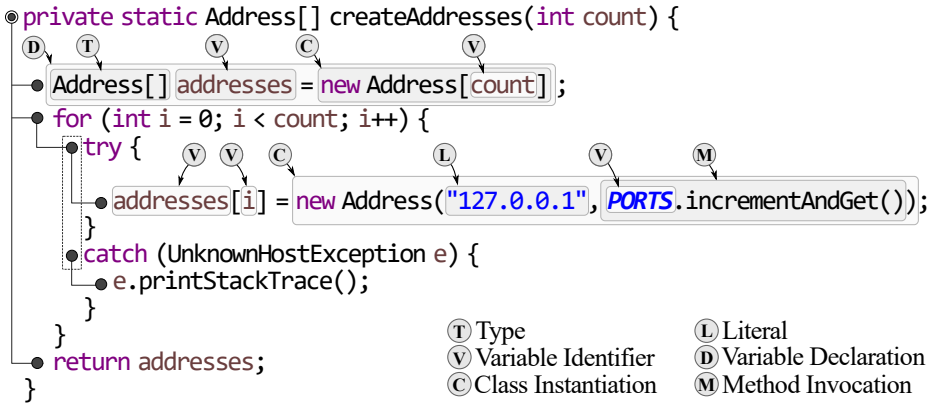

Figure 2: Representation of a method body as a tree.

Statement Matching

RefactoringMiner needs to find out which statements across the two revisions are possibly *matching*, and which ones are added in the new revisions or removed from the old revision, in order to be able to detect refactorings. In the context of Rename Local Variable refactoring, for instance, the renamed variables should be searched within the matched statements.

The statement matching algorithm employed by RefactoringMiner has been inspired by Fluri et al. [FWPG07], in the sense that the statements are matched in a bottom-up fashion, starting from the matching of leaf statements and then proceeding to composite statements. However, in contrast

to Fluri et al.'s approach, no similarity measure is used for matching the statements, and thus there is no need to define any similarity thresholds.

The Core Matching Function

Given two trees, T_1 and T_2, that represent the bodies of two methods in two revisions of the system under analysis, RefactoringMiner tries to find matching nodes between them. The core matching function used in RefactoringMiner (i.e., the function `matchNodes`) is depicted in Algorithm 1. This function accepts a set of nodes N_1 and N_2 from the two trees T_1 and T_2 (i.e., $N_1 \subseteq T_1.nodes$ and $N_2 \subseteq T_2.nodes$). In addition, the method accepts a criterion (i.e., the *matching condition*) according to which the node matching should be done. This is a function $\mathsf{matchCondition} : N_1 \times N_2 \rightarrow \{\texttt{true}, \texttt{false}\}$, which determines whether the two given nodes are matching or not. For each given node $n_1 \in N_1$, function `matchNodes` finds all possible nodes $n_2 \in N_2$ that match n_1 using the matching condition, and stores the matching pairs (n_1, n_2) into the set P.

To facilitate the matching of nodes that have undergone more radical changes across the two revisions (e.g., due to overlapping refactorings, or other maintetance tasks, such as bug fixing), RefactoringMiner applies two forms of preprocessing on each of the nodes, namely *abstraction* and *argumentation* (i.e., the `preprocessNodes(`n_1`,`n_2`)` function in line 5 of Algorithm 1). The preprocessing deals with specific changes taking place in the code when applying Extract, Inline, and Move Method refactorings and helps to increase the textual similarity of statements modified from the mechanics of a refactoring transformation [Fow99].

Abstraction: In a nutshell, *abstraction* deals with the cases in which types of the AST nodes of the statements that should be matched are different due to the refactoring, while the expressions within these statements are still more or less the same. For example, when an expression is extracted from a given method, it appears as a `return` statement in the extracted method.

To facilitate the matching of statements having a different AST node type, we *abstract* the statements that wrap expressions. When both statements being compared follow one of the following patterns, they are abstracted to `expression` before their comparison.

- `return expression;` i.e., returned expression
- `Type var = expression;` i.e., initializer of a variable declaration
- `var = expression;` i.e., right hand side of an assignment
- `call(expression);` i.e., single argument of a method invocation
- `if(expression)` i.e., condition of a composite statement

Algorithm 1: Statement Matching Function

Input : Nodes N_1 and N_2 from T_1 and T_2, respectively

Output: Set M of matched node pairs, Sets N_1 and N_2 with the matched nodes removed (i.e., the unmatched nodes from T_1 and T_2, respectively)

```
1  Function matchNodes(N1, N2, matchCondition)
2      foreach n1 ∈ N1 do
3          P ← ∅
4          foreach n2 ∈ N2 do
5              pn1, pn2 ← preprocessNodes(n1,n2)
6              if matchCondition(pn1,pn2) then
7                  P ← P ∪ (n1, n2)
8              end
9          end
10         if |P| > 0 then
11             bestMatch ← findBestMatch(P)
12             M ← M ∪ bestMatch
13             N1 ← N1 \ bestMatch.n1
14             N2 ← N2 \ bestMatch.n2
15         end
16     end
17     return M, N1, N2
18 end
```

Consider, for example, an instance of Extract Method Refactoring found in the `hazelcast` project, illustrated in Figure 3. Here, to detect this refactoring instance, we need to match statement (D) in the old revision of the code with statement (5) in the extracted method in the new revision of the code. As it is observed, the types of the AST nodes for these statements are different, one being an Assignment statement, while the other is a Return statement. The two statements are abstracted into two Class Instance Creation expressions:

```
new Address("127.0.0.1", PORTS.incrementAndGet())
```

and

```
new Address(host, port)
```

so that they are more similar. However, as it is observed, the arguments passed to the class constructors are still different, and thus the matching is still not possible.

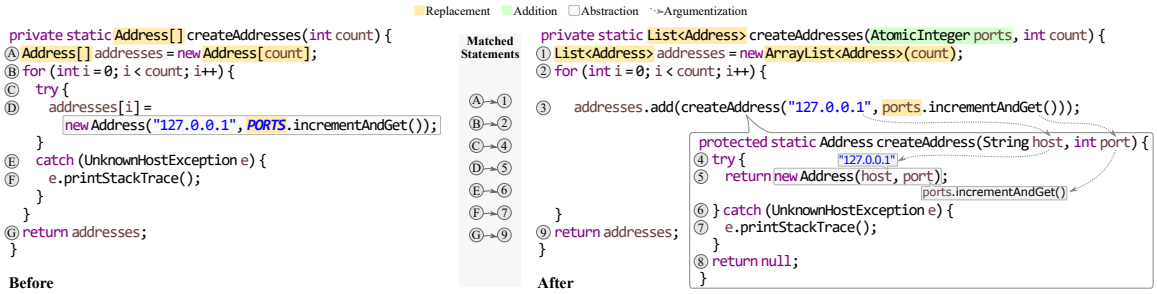

Figure 3: Statement matching for an EXTRACT METHOD refactoring in project hazelcast.

Argumentation: This technique deals with the cases where a refactoring replaces expressions with parameters, and vice versa. For instance, when duplicated code is extracted into a common method, all expressions being different among the duplicated code fragments are *parameterized* (i.e., they are replaced with parameters in the extracted method). The duplicated code fragments are replaced with calls to the extracted method, where each expression being different is passed as an argument. In many cases, the arguments may differ substantially from the corresponding parameter names, leading to a low textual similarity of the code before and after refactoring. Argumentization is the process of replacing parameter names with the corresponding arguments in the code after refactoring.

In the example of Figure 3, we mentioned that abstraction is not enough for matching the two statements ⑤ and Ⓓ. However, if the parameters `host` and `port` used in statement ⑤ are replaced with arguments `"127.0.0.1"` and `ports.incrementAndGet()`, respectively, the resulting statement becomes identical with statement Ⓓ (after applying abstraction).

The same process is applied to the statements of inlined and moved methods. In particular, when an instance method is moved to a target class, we might have a parameter (or a source class field access) of target type that is removed from the original method, or a parameter of source type that is added to the original method. In the case of removal, the removed parameter (or field access) might be replaced with `this` reference in the moved method, while in the case of addition, `this` reference might be replaced with the added parameter in the moved method.

Algorithm 1 applies the matching condition on the preprocessed (i.e., abstracted and argumentized) nodes to populate the set P. This set, as mentioned, contains all pairs in the form of (n_1, n_2) for each given $n_1 \in N_1$, where $n_2 \in N_2$ matches n_1 based on the matching condition. The algorithm then tries to find the best match for each $n_1 \in N_1$. Function `findBestMatch(`P`)` (line 11), sorts the node pairs in P and selects the top-sorted one, i.e., the best match for each given $n_1 \in N_1$. *Leaf node pairs* are sorted based on 3 criteria:

1. Based on the string edit distance [Lev66] of the nodes in ascending order (i.e., more textually

similar node pairs rank higher).

2. Based on the absolute difference of the nodes' depth in ascending order (i.e., node pairs with more similar depth rank higher).

3. Based on the absolute difference of the nodes' index in their parent's list of children in ascending order (i.e., node pairs with more similar position in their parent's list of children rank higher).

Composite node pairs are sorted with an additional criterion, which is applied right after the first criterion: based on the ratio of the nodes' matched children in descending order (i.e., node pairs with more matched children rank higher).

As mentioned, RefactoringMiner uses the function `matchNodes` that we described above to find matching nodes between two given trees, T_1 and T_2, representing the bodies of methods appearing in two revisions of the code, respectively. To reduce the chances of erroneous matches, a conservative approach is followed, in which the statements are matched in rounds, where each subsequent round has a less strict *matching condition* than the previous round. Thus, the statements matched in earlier rounds are "safer" matches, and are excluded from being matched in the next rounds. In this way, the next round, which has a more relaxed match condition, has fewer statement combinations to check. In the next two subsections, we describe how `matchNodes` is used in action, when matching the nodes between T_1 and T_2.

Matching Leaf Nodes

RefactoringMiner first tries to match the leaf statements, using the `matchNodes` function. The matching of leaf statements is done in three rounds. In the first round, it matches the statements with identical string representation and nesting depth. In other words, the function matchCondition for round one is defined as:

$$\mathsf{matchCondition}_{leaves-round1}(n_1, n_2) = n_1.\mathsf{text} == n_2.\mathsf{text} \wedge n_1.\mathsf{depth} == n_2.\mathsf{depth}$$

In the second round, RefactoringMiner matches the statements with identical string representation regardless of their nesting depth. In other words:

$$\mathsf{matchCondition}_{leaves-round2}(n_1, n_2) = n_1.\mathsf{text} == n_2.\mathsf{text}$$

In the third round, RefactoringMiner matches the statements that become identical after replacing the AST nodes being different between the two statements, i.e:

$$\mathsf{matchCondition}_{leaves-round3}(n_1, n_2) = |\mathtt{replacements}(n_1.\mathsf{text}, n_2.\mathsf{text})| > 0$$

where function `replacements` determines whether there exists a set of AST nodes in the second statement that can replace some AST nodes in the first statement so that the statements become

textually identical, and returns all such replacements. For example, for statements Ⓐ and ① from Figure 3:

```
Ⓐ Address[] addresses = new Address[count];
```

and

```
① List<Address> addresses = new ArrayList<Address>(count);
```

replacing `Address[]` and `Address[count]` from Ⓐ with `List<Address>` and `ArrayList<Address>(count)` from ①, respectively, makes the two statements textually identical. In this case, the function `replacements` (Ⓐ`.text`,①`.text`) returns the following set of replacements:

$$\{\texttt{Address[]} \rightarrow \texttt{List<Address>}, \texttt{Address[count]} \rightarrow \texttt{ArrayList<Address>(count)}\}$$

Trying to find replacements for the AST nodes in the statements for matching has two main advantages over existing methods that rely on textual similarity.

First, there is no need to define a similarity threshold. There is empirical evidence that developers interleave refactoring with other types of programming activity (e.g., bug fixes, feature additions, or other refactoring operations) [MHPB12, STV16, NCV+13]. In many cases, the changes caused by these different activities may overlap [NVC+12]. Some of these changes may even change substantially the original code being part of a refactoring operation. For example, a code fragment is originally extracted, and then some temporary variables are inlined in the extracted method. The longer the right-hand-side expressions assigned to the temporary variables, the more textually different the original statements will be after refactoring. Therefore, it is impossible to define a *universal* similarity threshold value that can cover any possible scenario of overlapping changes. This approach does not pose any restriction on the replacements of AST nodes, as long as these replacements are syntactically valid.

Second, the replacements found within two matched statements can help to infer other edit operations taking place on the refactored code (a phenomenon called *refactoring masking* [SPDZ15]), such as renaming of variables, generalization of types, and merging of parameters. On the other hand, similarity-based approaches lose this kind of valuable information. Indeed, we used this information to detect the instances of Rename Local Variable refactoring.

Initially, the `replacements` function computes the intersection between the sets of variable identifiers, method invocations, class instantiations, types, literals, and operators extracted from each

statement, respectively, in order to exclude from replacements the AST nodes being common in both statements, and include only those that are different between the statements. AST nodes that cover the entire statement (e.g., a method invocation followed by semicolon) are also excluded from replacements in order to avoid having an excessive number of matching statements. All attempted replacements are *syntax-aware*, in the sense that only *compatible* AST nodes are allowed to be replaced, i.e., types can be replaced only by types, operators can be replaced only by operators, while all remaining expression types can be replaced by any of the remaining expression types (e.g., a variable can be replaced by a method invocation). Out of all possible replacements for a given node from the first statement that decrease the original edit distance of the input statements, we select the replacement corresponding to the *smallest edit distance*.

Matching Composite Nodes

The composite statements from the two given trees T_1 and T_2 are also matched in three rounds, using exactly the same match conditions as those used for leaf statements combined with an additional condition that requires at least one pair of their children to be matched, assuming that both composite statements have children. In other words:

$$\mathsf{condition4}\ (n_1,\ n_2) = \exists\ (k_1, k_2) \in M \mid k_1 \in n_1.\mathsf{children} \wedge k_2 \in n_2.\mathsf{children}$$

$$\mathsf{matchCondition}_{composites-round1}(n_1, n_2) = \mathsf{matchCondition}_{leaves-round1}(n_1, n_2) \wedge \mathsf{condition4}\ (n_1,\ n_2)$$

$$\mathsf{matchCondition}_{composites-round2}(n_1, n_2) = \mathsf{matchCondition}_{leaves-round2}(n_1, n_2) \wedge \mathsf{condition4}\ (n_1,\ n_2)$$

$$\mathsf{matchCondition}_{composites-round3}(n_1, n_2) = \mathsf{matchCondition}_{leaves-round3}(n_1, n_2) \wedge \mathsf{condition4}\ (n_1,\ n_2)$$

3.1.2 Refactoring Detection in RefactoringMiner

RefactoringMiner detects refactorings in two phases:

1. In the first phase, it matches code elements in a top-down fashion, starting from classes and continuing to methods and fields. Two code elements are matched only if they have an identical *signature*. Assuming a and b are two revisions of a project:

   - Two type declarations td_a and td_b have an identical signature, if $td_a.p = td_b.p\ \wedge td_a.n = td_b.n$
   - Two fields f_a and f_b have an identical signature, if $f_a.c = f_b.c\ \wedge f_a.t = f_b.t\ \wedge f_a.n = f_b.n$
   - Two methods m_a and m_b have an identical signature, if $m_a.c = m_b.c\ \wedge m_a.t = m_b.t\ \wedge m_a.n = m_b.n\ \wedge m_a.P = m_b.P$

- Two directories d_a and d_b are identical, if $d_a.p = d_b.p$

The first phase is less computationally expensive, since the code elements are matched only based on their signatures. Our assumption is that two code elements having an identical signature in two revisions correspond to the same code entity, regardless of the changes that might have occurred within their bodies.

After the end of the first phase, we consider the unmatched code elements from revision a as *potentially deleted*, and store them in sets TD^-, F^-, M^-, and D^-, respectively. We consider the unmatched code elements from revision b as *potentially added*, and store them in sets TD^+, F^+, M^+, and D^+, respectively. Finally, we store the pairs of matched code elements between revisions a and b in sets $TD^=$, $F^=$, $M^=$, and $D^=$, respectively.

2. The second phase is more computationally expensive, since the remaining code elements are matched based on the statements they have in common within their bodies. In this phase, our algorithm matches the remaining code elements (i.e., the *potentially deleted* code elements with the *potentially added* ones) in a bottom-up fashion, starting from methods and continuing to classes, to find code elements with signature changes or code elements involved in refactoring operations.

RefactoringMiner applies a set of rules in the second phase to detect refactorings. For example, to detect the instances of Extract Method refactoring, where in the new revision of the code method m_b is extracted from the body of method m_a that appeared in the old revision of the code, the following rule is applied:

$$\exists\ (M, U_{T_1}, U_{T_2}) = \texttt{statementMatching}(m_a.body, m_b.body)\ |\ (m_a, m_{a'}) \in M^=$$
$$\wedge\ m_b \in M^+ \wedge\ m_a.c = m_b.c\ \wedge \neg\texttt{calls}(m_a, m_b) \wedge \texttt{calls}(m_{a'}, m_b) \wedge |M| > |U_{T_2}|$$

In this rule,

- statementMatching is a function that essentially performs what we described in the previous section: it accepts the two given trees, T_1 and T_2, representing the bodies of the two methods in the two revisions, and returns a tuple containing 3 sets (M, U_{T_1}, U_{T_2}) where M is the set of matched nodes in T_1 and T_2, and U_{T_1} and U_{T_2} correspond to the sets of unmatched nodes in T_1 and T_2, respectively. Recall that in the heart of statementMatching, the `matchNodes` function is used in several rounds for the leaf and composite nodes of T_1 and T_2.

- Function $\texttt{calls}(m_a, m_b)$ returns `true` if method m_a calls m_b.

In plain English, if there exists a matching between the statements appearing within the bodies of two methods m_a and m_b such that:

- There exists a method $m_{a'}$ in the new revision of the code that has been matched with m_a, i.e., $(m_a, m_{a'}) \in M^{=}$, and
- There exists a new method m_b which has been added in the new revision of the code, i.e., $m_b \in M^{+}$, and
- Both methods m_a and m_b belong to the same class, i.e., $m_a.c = m_b.c$, and
- m_a did not call m_b in the previous revision, and
- $m_{a'}$ calls m_b in the new revision, and
- The number of matched nodes in the bodies of m_a and m_b is greater than the number of unmatched nodes in the body of m_b, i.e., $|M| > |U_{T_2}|$.

then RefactoringMiner reports that m_b has been extracted from m_a.

Apart from Extract Method refactoring, RefactoringMiner implements the necessary rules for detecting instances of 14 other refactoring types: Inline Method, Move Field, Move Class, Extract Interface, Push Down Method, Push Down Field, Change Package, Pull Up Method, Pull Up Field, Move Method, Rename Method, Extract Superclass, Rename Class, and Extract & Move Method. As it is observed, there is no rule for detecting the instances of the Rename Local Variable refactoring.

3.2 Detecting Rename Local Variable Refactoring Instances

Our approach for detecting instances of Rename Local Variable refactorings is similar to other refactoring types, but there are several challenges associated with it. The rule that we have proposed for this type of refactoring is as follows:

$$\exists\ (M, U_{T_1}, U_{T_2}, R_M) = \mathsf{statementMatching}(m_a, m_{a'}) \mid (m_a, m_{a'}) \in M^{=}$$
$$\wedge\ \exists\ r \in R_M = l \rightarrow l' \mid\ l \in \texttt{localVariables(}m_a\texttt{)} \wedge l' \in \texttt{localVariables(}m_{a'}\texttt{)}$$
$$\wedge\ l \notin \texttt{variables(}m_{a'}\texttt{)} \wedge l' \notin \texttt{variables(}m_a\texttt{)}$$
$$\wedge\ l \notin E_{m_a} \wedge l' \notin I_{m_a}$$
$$\wedge\ \nexists\ r_2 \in R_M \cup R_E = a \rightarrow b \mid a = l$$
$$\wedge\ \nexists\ r_3 \in R_M \cup R_I = c \rightarrow d \mid d = l'$$

In this rule:

- R_M is the set containing all AST node replacements (as computed in the `replacements` function that was explained in Section 3.1.1) extracted from the matched statements between methods m_a and $m_{a'}$. Note that $l \rightarrow l'$ means that l is replaced by l'. In addition, R_E (R_I) correspond to the set of replacements that RefactoringMiner has found in the matched statements of methods extracted from (inlined to) m_a. We will shortly show how looking at the extract method/inline method refactorings can help in detecting more advanced types of Rename Local Variable refactorings.

- `localVariables(`m_a`)` returns all local variables defined in the body of m_a.

- `variables(`m_a`)` returns all variables defined or used in the body of m_a, including local variables, parameters, and fields.

- E_{m_a} (I_{m_a}) is the set of all variables which appear in the body of the methods extracted from (inlined to) the method m_a, as detected by RefactoringMiner.

In plain English, an instance of a Local Variable Renaming is detected when:

1. There exists a method $m_{a'}$ in the new revision of the code that was matched with m_a, i.e., $(m_a, m_{a'}) \in M^{=}$.

2. There exists a replacement r which replaces the local variable l in the old revision with l' in the new revision, i.e., $\exists\ r \in R_M = l \rightarrow l'$.

3. l should belong to the declared local variables in m_a, and l' should belong to the declared local variables in $m_{a'}$, i.e., $l \in$ `localVariables(`m_a`)` $\wedge\ l' \in$ `localVariables(`$m_{a'}$`)`.

4. l should not exist in the new revision of the code, and l' should not exist in the old revision of the code, i.e., $l \notin$ `variables(`$m_{a'}$`)` $\wedge\ l' \notin$ `variables(`m_a`)`.

5. l should not appear in the body of the methods extracted from m_a, as detected by RefactoringMiner, i.e., $l \notin E_{m_a}$. We explain the reason why the renamed local variable should not appear in the body of the extracted methods using an example.

```
void m() {
 String c = "some value";
 consume(c);
 ...
}
```

(a) Before

```
void m() {
 String d = "some other value";
 consume(d);
 extracted();
 ...
}
```

```
void extracted() {
  String c = "some value";
  consume(c);
 }
```

(b) After

Figure 4: Renaming Ambiguity (Same Variable in the Extracted Method)

In Figure 4a, the local variable `c` is defined and used in the body of method `m()`. In the next revision (Figure 4b), there is a local variable d defined in the body of `m()`, and also a method `extracted()` that has been extracted from `m()`. Without considering the extracted method, one might assume that the local variable `c` has been renamed to `d`. However, it is much more probable that the developer has extracted the variable `c` and the surrounding code into the extracted method, and variable `d` is just a new one added to method `m()`.

In general, we try to avoid reporting instances of Rename Local Variable in similar cases, where there is not enough evidence, or there is ambiguity. In this way, we try to keep the *precision* of our approach as high as possible while the *recall* can be negatively affected. This is because there is evidence showing that developers are bothered much more with a high number of false positives rather than missing refactorings (i.e., false negatives) [CB16].

Similarly, l' should not appear in the body of the methods inlined into m_a, i.e., $l' \notin I_{m_a}$. The reasoning behind this is similar to the extract method case.

6. A local variable cannot be renamed to two local variables. As a result, while having a replacement $r = l \rightarrow l'$, there should not be another replacement $r_2 = l \rightarrow b$, i.e., a replacement that allows to match two nodes by replacing l in the old revision with b in the new revision.

```
Throwable throwable = null;
...
try {
...
} catch (IOException e) {
  ...
  throwable = e;
} catch (UnavailableException e) {
  ...
  throwable = e;
}
```

(a) Before

```
IOException ioException = null;
ServletException servletException = null;
...
try {
...
} catch (IOException e) {
  ...
  ioException = e;
} catch (UnavailableException e) {
  ...
  servletException = e;
}
```

(b) After

Figure 5: Variable Splitting

An example of a local variable reported to be replaced by two other local variables is illustrated in Figure 5. This code has been found in the `Tomcat` project[2]. Here, the variable `throwable` in the old revision (i.e., Figure 5a) was used to store different values, depending on the raised exception. In the next revision (i.e., Figure 5b), the variable is actually *split* into two variables, `ioException` and `servletException`, and each of them is assigned depending on the raised exception. Note that, when matching the bodies of the two methods in the two revisions, two

[2] https://github.com/MatinMan/tomcat-code/commit/4a80db9b2d96148b42047a8c066f9e5e4390220b#diff-e8c6ad4c90e1f58006925472c3558ea2L467

replacements are reported: `throwable` → `ioException` and `throwable` → `servletException`. Both replacements are correct, as they make the corresponding statements textually identical. However, by looking at these two replacements, it is not really possible to certainly say whether `throwable` is renamed to `ioException` or `servletException`.

In addition, if a local variable l is renamed in a method m_a, and at the same time a method is extracted from m_a, there should not be a local variable b in the extracted method that l could be possibly renamed to it. This is because in this case we cannot decide which renaming has actually happened in the code. Figure 6 depicts such example. This example is very similar to Figure 4, but the variable `c` in Figure 4b has been renamed to `z` in Figure 6b.

```
void m() {
 String c = "some value";
 consume(c);
 ...
}
```

(a) Before

```
void m() {
 String d = "some other value";
 consume(d);
 extracted();
 ...
}
```

```
void extracted() {
  String z = "some value";
  consume(z);
 }
```

(b) After

Figure 6: Renaming Ambiguity (Renaming in the Extracted Method)

Note again that the local variable `c` can be possibly reported to be renamed to `d` in the body of `m()`. However, the same variable can also be extracted to method `extracted()` as a part of an extract method refactoring, and then be renamed to `z`. As a result, we cannot decide which renaming has actually happened here.

To avoid reporting such cases, we argue that there should not be a replacement computed when detecting extract method refactorings from m_a that replaces l with b in the body of the extracted method.

The mentioned two rules correspond to the following formula $\nexists\, r_2 \in R_M \cup R_E = a \to b \mid a = l$.

7. Similar to the previous rule, we cannot have two local variables being renamed to one local variable. Moreover, when a local variable l in m_a is reported to be replaced by (i.e., renamed to) l' and at the same time there is another variable c belonging to a method that is being inlined to m_a and c is also reported to be replaced by l' in it, we cannot decide which renaming has actually happened.

   These two rules are implemented by the following formula: $\nexists\, r_3 \in R_M \cup R_I = c \to d \mid d = l'$.

3.2.1 Exceptional Cases

There are two exceptions for rules #6 and #7, where we improve the accuracy of detecting instances of Rename Local Variable refactorings by trying to correct the shortcomings of the matching algorithm. In particular, if besides having the replacement $r = l \rightarrow l'$ there is another replacement $r' = l \rightarrow l''$ in two matched statements in m_a and $m_{a'}$, different situations might have occurred:

- The local variable l in r could be defined in a different lexical scope than l in r' (while they have the same name). For instance, consider the example depicted in Figure 7. This code has been found in the `dnsjava` project[3].

```
while (...) {
 RRset rrset = (RRset)e.nextElement();
 addRRset(response, rrset);
}
...
if (...)
 RRset rrset = zone.findRecords(name, type);
 if (rrset != null)
  addRRset(response, rrset);
}
```

(a) Before

```
while (...) {
 Record cname = (Record)e.nextElement();;
 response.addRecord(cname, Section.ANSWER);
}
...
if (...)
 RRset[] rrsets = cr.answers();
 for (int i = 0; i < rrsets.length; i++)
  addRRset(response, rrsets[i]);
}
```

(b) After

Figure 7: Renamed Variables in Different Lexical Scopes

  As it can be observed, the `rrset` variable declared in the `while` loop in Figure 7a is potentially renamed to variable `cname` in the `while` loop in Figure 7a. In other words, to match the first statements in the two `while` loops, one replacement is `rrset` $\rightarrow$ `cname`. At the same time, the variable `rrset` inside the `if` statement in Figure 7a could be also possibly replaced with `rrsets` in Figure 7b, i.e., there is a replacement `rrset` $\rightarrow$ `rrsets`.

  Note that the function `replacements` computes the possible replacements in the bodies of two matched methods without considering the lexical scope in which the variables appear within the body of the matched methods. In such cases, the variable replacements are treated individually, and a separate instance of Rename Local Variable is reported for each variable replacement (given that the other mentioned conditions hold). In the example of Figure 7, both instances of the Rename Local Variable refactoring are reported: `rrset` to `cname` and `rrset` to `rrsets`.

- In contrast, if in the two replacements $l \rightarrow l'$ and $l \rightarrow l''$ the two variables named l belong to the same lexical scope, it might be that the set of replacements reported by `replacements` is not *optimal*.

[3] https://github.com/MatinMan/RefactoringDatasets/commit/fe3d0590ecf026ebd94b6e71f8c779c5cbedfac5#diff-9665c59364c6022679c637d56c09ae01L216

For example, consider the code snippets in Figure 8, found in the `Tomcat` project[4].

```
if (!resourceInfo.exists) {
...
}
...
if (ostream != null) {
 copy(resourceInfo, ostream);
} else {
 copy(resourceInfo, writer);
}
...
```

(a) Before

```
if (!cacheEntry.exists) {
...
}
...
if (ostream != null) {
 copy(cacheEntry, renderResult, ostream);
} else {
 copy(cacheEntry, renderResult, writer);
}
...
```

(b) After

Figure 8: Method Invocations with Possible Non-Optimal Argument Replacement

In this example, to match the two method invocations:

i_1: `copy(resourceInfo, ostream)`

i_2: `copy(cacheEntry, renderResult, ostream)`

RefactoringMiner needs to find a replacement that minimizes the difference between the two invocations. Applying the replacement `cacheEntry` $\rightarrow$ `resourceInfo` to i_2 will result to the string `copy(resourceInfo, renderResult, ostream)`, which has the Levenestein distance of 13 with i_2, or the normalized similarity of $1 - 13/39 = 0.66$, where 39 is the length of the longer string in the comparison. However, there is another possible replacement, `renderResult` $\rightarrow$ `resourceInfo`, which converts i_2 to `copy(cacheEntry, resourceInfo, ostream)`. In this case, the normalized similarity between the resulting string and i_2 is $1 - 11/39 = 0.71$, which is a higher similarity than the previous replacement. As a result, the second replacement is preferred over the previous one, and the two method invocations are reported to be matching using this replacement. Note that, although the replacement is not optimal, yet the result is fine with RefactoringMiner, since the two method invocations are matched anyway.

However, such non-optimal replacements can make detecting Rename Local Variable instances erroneous. In other words, if there are two replacements $l \rightarrow l'$ and $l \rightarrow l''$ in the same lexical scope, any of these two replacements could be reported just because the algorithm infers the replacements in a non-optimal way.

To solve this issue, for the matching statements corresponding to the originally-reported replacements (which might be non-optimal), we consider *all other possible replacements* (i.e., not only the

[4] `https://github.com/MatinMan/tomcat-code/commit/d6576bb2b3895ca717c3b4d22afae5d5bcc90608#diff-9ce171d3985c65aea7b23cd950aed11fL984`

best replacement that minimizes the textual dissimilarity between the statements) to see whether there exists another replacement consistent with the other reported replacements.

To clarify this, suppose that there are two pairs of matched statements, (s_1, s_1') and (s_2, s_2'), across th two revisions under analysis. Let's say that s_1 and s_1' are matched because there exists a best replacement, $r_1 = l \rightarrow l'$, that makes s_1 and s_1' highly textually similar. At the same time, suppose that for statements s_2 and s_2', the best reported replacement that makes the matching possible is $r_2 = l \rightarrow l''$. Now imagine that the same replacement $r_1 = l \rightarrow l'$ could be possibly applied on s_2 to make it textually similar to s_2', but the similarity in this case is less than when we apply $r_2 = l \rightarrow l''$. In this case, we ignore r_2 (i.e., the best replacement), since it is not the optimal replacement for detecting the instances of Rename Local Variable refactoring, and our approach would report that the local variable l is renamed to l' (given that all other necessary conditions are met).

3.3 Chapter Summary

In this chapter, we described how RefactoringMiner works, and our method (created on top of RefactoringMiner) for detecting instances of Rename Local Variable refactorings.

In the next chapter, we describe the design of our study to compare the results of our technique with the state-of-the-art rename detection technique, namely REPENT.

Chapter 4

Evaluation

In the previous chapter, we described our approach for detecting instances of Rename Local Variable refactorings in the history of software systems. In this chapter, we evaluate our approach. Particularly, we aim at answering the following research questions:

RQ1 *How accurate is our technique in detecting instances of Rename Local Variable refactorings?*
Particularly, we use the common measures adopted from the information retrieval literature to report the accuracy of our technique, namely in terms of its *precision* and *recall*.

RQ2 *How does our technique perform compared to* REPENT*?*
We compare the accuracy of our technique with the state-of-the-art tool for detecting instances of Rename Local Variable refactorings, namely REPENT. Again, we report the *precision* and *recall* of REPENT. We also discuss the advantages and disadvantages of each technique and illustrate some interesting examples found during our evaluations, where the two techniques perform differently.

RQ3 What is the efficiency of our technique in terms of the time taken for detecting instances of Rename Local Variable refactorings?

To answer RQ1, we need to have multiple software repositories on which we can apply the proposed technique, and a dataset of refactorings that we are certain they have occurred in those repositories (i.e., a *reference corpus*), so that we can compare the results of our technique against others and report their accuracy. Moreover, to answer RQ2, we need to run REPENT on the same reference corpus to make a fair comparison with our technique. In the following subsection, we will describe how we constructed a reference corpus for this study and how we limited bias in the validation process.

4.1 Reference Corpus Construction

There are two reasons why constructing a reference corpus for refactoring detection is a challenging task:

- Often, the number of refactorings applied on a software system is unknown. The developers usually do not document the refactoring activities. For example, we mentioned that it is not possible to identify all refactorings that occurred in the code by only looking at the commit messages [MHPB12]. As a result, it is not straightforward to assess the *completeness* of any given reference corpus.
- The correctness of the refactoring instances in the reference corpus is unknown, unless the developers themselves admit that they have applied them, or the detected refactorings are validated manually by (ideally multiple) independent people.

In general, there are two approaches for constructing a reference corpus of refactorings:

- Forming a *seeded* dataset. This might be considered as the easiest solution. In this approach, we consider the source code of a software system at a specific revision, and ask some developers to apply refactorings on it.
- The other solution is to run two or more refactoring detection tools on the same repository and compare all refactoring instances detected by the tools to assess their *agreement*. The instances for which there is a total agreement (i.e., all tools detect them), or a majority agreement (i.e., more than half the tools detect them) can be considered as true positives. We might also manually validate the detected refactoring instances to have more confidence about the correctness of the reported refactorings.

Each of the mentioned approaches has its own advantages and disadvantages. Knowing all the refactorings that have occurred in the source code and being certain that the reported refactorings are correct are the main advantages of the first approach (i.e., the seeded reference corpus). However, the applied refactorings by developers are not necessarily representative of real refactorings. Moreover, it is known that the refactoring activities are interleaved with other maintenance changes [STV16] on the source code, making the refactoring detection process much more difficult. In other words, the seeded refactorings are *artificial* and usually not *realistic*.

While the second approach (i.e, using the *tool agreement*) overcomes the mentioned problem, it has other limitations. First, it is not guaranteed that it contains all refactorings applied in the code as it is limited to the accuracy of the used tools. Moreover, the results of the tools might need to be manually validated to make sure that they are correct, which is a daunting task.

However, only by evaluating a refactoring detection approach against a reference corpus containing real refactorings applied by developers, we can assess its accuracy in a reliable manner. Thus, we choose to follow (an almost similar method to) the second mentioned approach, i.e., using tools' agreement, for evaluating our technique.

As mentioned in Chapter 2, there are a few refactoring detection approaches which support Rename Local Variable refactoring. Unfortunately, at the time of writing this thesis, none of the corresponding tools are available (this is is also mentioned in another study [LLLW15]). This includes the state-of-the-art tool for detecting instances of the Rename Local Variable refactorings, namely REPENT [AEP+14], which we have chosen to compare against the results of our technique.

Consequently, instead of running REPENT and our technique on any repository to seek their agreements, we use the datasetthat REPENT was evaluated on, which has been made publicly available by its authors [AEP+14].

4.1.1 Subject Systems

The authors or REPENT used five repositories in their study. As we will see, we are going to make sure that all reported instances from both tools are manually validated. This is a daunting task that needs several person-months to finish. As a result, we randomly selected two of these systems to feed our proposed technique, namely DNSJAVA and TOMCAT.

DNSJAVA[1] implements DNS protocol in Java. TOMCAT[2] is an implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies. The characteristics of the studied systems are depicted in Table 1(the table has been adopted from [AEP+14]).

Table 1: Characteristics of the Subject Systems

Software	Studied Revisions Period	Files	Total File Revisions	KLOC
DNSJAVA	1998-2011	365	1,415	9-35
TOMCAT	1999-2006	12,205	46,498	5-35

4.1.2 Automated Reference Corpus Construction

To find the agreement between our tool and REPENT, we developed another tool, called REFBENCHMARK, that automates the computation of agreement among multiple tools. In a nutshell, REFBENCHMARK creates abstract models of refactorings reported by different tools, and tries to map

[1] http://www.xbill.org/dnsjava/
[2] http://tomcat.apache.org/

the refactoring models created from the findings of one tool to the models created from the findings of another tool, in order to computed their agreements (and disagreements). RefBenchmark is not only designed for Rename Local Variable refactoring instances, in fact it supports 15 different types of refactorings and can be easily extended to support more types. It can be also extended to accept more refactoring detection tools as input. We describe the design of RefBenchmark in more detail in Chapter 5.

4.1.3 Systematic Manual Validation of the Detected Refactorings

To increase the certainty about the correctness of the results, we take a step further and make sure that all results reported by the two tools which are going to be inserted into the reference corpus are manually validated by at least one person.

The authors of REPENT used random sampling (with the confidence level of 95 percent) to validate the results of their tool, since validating all results was cumbersome. Then, two authors of the paper independently validated each refactoring instance in the sample.

We first extract all those validated instances from the reference corpus (the authors reported which instances are manually validated), since they should be correct given that two authors of REPENT have already validated them. Specifically, REPENT found 396 instances of Rename Local Variable refactoring in Tomcat, out of which 180 were manually validated. For dnsjava, these numbers are 144 and 32, respectively.

Then, we run our approach on the same commits of the two repositories, and use RefBenchmark to find the agreement between the results of both approaches. For the results which are detected by both approaches, it should be enough to be validated by one person. For the rest of the detected refactoring instances (i.e., the ones which are detected by only one tool), a researcher experienced enough with refactoring is asked to validate them. We accept the judgment made by the researcher if a case is deemed to be straightforward by her. For the other cases, where their complexity does not allow the researcher to reach a final decision, we asked a second researcher, again experienced enough with refactoring, to validate the case independently. The agreement between these two researchers will be used as the final decision about the case. If there is a disagreement for a case, a discussion is commenced between the two validators with the hope that the disagreement is resolved. Even in a very few cases, a third researcher was asked to resolve the disagreement between the two validators. In any case, there are a few cases about which both validators could not make a certain decision. These are the cases that only the original developer (or the person who applied the refactoring) might be able to judge about. We excluded such cases from the reference corpus. Eventually, our reference corpus contained 396 instances of Rename Local Variable refactorings from Tomcat and 128 instances from dnsjava.

4.2 Results

Having the reference corpus ready, we used well-known measures from the information retrieval literature to assess the accuracy of our technique and compare it with REPENT's accuracy. More specifically, for each tool, we first calculate the following raw measures:

True Positives (i.e., **TP**s) represent the instances of Rename Local Variable refactorings detected by the tool and exist in the reference corpus i.e., the correctly detected refactorings.

False Positives (i.e., **FP**s) represent the instances of Rename Local Variable refactorings reported by the tool, yet the reference corpus does not include them. In other words, False Positives are incorrectly detected refactorings.

False Negatives (i.e., **FN**s) represent the instances of Rename Local Variable refactorings that are not detected by the tool, yet the reference corpus includes them, i.e., the missed refactoring instances.

Then, we derive these two relative measures from the mentioned raw measures, which enable direct comparison of the results of the two tools:

- To estimate how accurate each tool is in the detection of Rename Local Variable refactoring instances, we calculate the *precision* of the tools using the following formula:

$$precision = \frac{|TP|}{|TP| + |FP|}$$

  Precision basically shows how many of the detected refactoring instances are correct.

- Measuring the *completeness* of the results of each approach is not straightforward, since one would need to know *all* refactoring instances that have actually occurred on the studied repositories. However, we can count the correct cases that one tool can detect as False Negatives for the other tool which is unable to identify them. The following formula is used to compute the *recall* of each tool as a measure representing the completeness of the results reported by each tool:

$$recall = \frac{|TP|}{|TP| + |FN|}$$

  Recall basically shows how many of the instances that a tool is supposed to report are actually found by the tool.

This section explains the results of both tools and gives an answer to the research questions.

4.2.1 RQ1: How accurate is our technique in detecting instances of Rename Local Variable refactorings?

Table 2 represents the results of running our technique on the subject systems.

Table 2: The accuracy of our technique

Software	#Refactorings	TP	FP	FN	Recall	Precision
TOMCAT	396	344	55	52	86.9%	86.2%
DNSJAVA	128	111	10	17	86.7%	91.7%
Total	524	455	65	69	86.8%	87.5%

Overall, we observe that the recall of our technique is over 86 percent on both repositories. In terms of precision, our technique performs better on DNSJAVA compared to TOMCAT. Our investigations showed that sometimes there are larger changes in the TOMCAT dataset, making our technique report incorrect statement matches that could lead to incorrectly identifying some Rename Local Variable refactoring instances.

4.2.2 RQ2: How does our technique perform compared to REPENT?

Table 3 reports the accuracy of REPENT in the studied subject systems. REPENT demonstrates its best recall in DNSJAVA, but as we observe its precision is higher in TOMCAT. This is predictable, since the thresholds used in REPENT are tuned using TOMCAT.

Table 3: The accuracy of REPENT

Software	#Refactorings	TP	FP	FN	Recall	Precision
TOMCAT	396	320	77	76	80.8%	80.6%
DNSJAVA	128	109	35	19	85.2%	75.7%
Total	524	429	112	95	81.8%	79.3%

To facilitate the comparison of the accuracy for both tools, we have depicted them side-by-side in Table 4. It is observed that our technique outperforms REPENT in both systems. The recall in DNSJAVA is the only case where the performance of both tools is very close.

Table 4: Side-by-side comparison of the accuracy of our approach and REPENT

		Our approach		REPENT	
Software	#Refactorings	Recall	Precision	Recall	Precision
TOMCAT	396	86.9%	86.2%	80.8%	80.6%
DNSJAVA	128	86.7%	91.7%	85.2%	75.7%
Total	524	86.8%	87.5%	81.8%	79.3%

4.2.3 RQ3: What is the efficiency of our technique?

To understand the efficiency of our technique, we compute the time taken for detecting the instances of Rename Local Variable refactoring. Unfortunately, since REPENT is not available, we are not able to make a comparison between our technique and REPENT.

Figure 9 illustrates the distribution of the time (in milliseconds, on the logarithmic scale), which was taken for each commit to be analyzed using our technique. Note that this time is the *entire time* taken by REFACTORINGMINER to detect all 15 supported refactoring types, plus the time needed by our approach for detecting the instances of Rename Local Variable refactoring. This is because detecting Rename Local Variable refactorings depends on identifying other refactoring types (e.g., Extract and Inline Method refactorings). As such, it is more fair to compute the entire time for the whole analysis instead of only showing the time taken by our algorithm.

The times are calculated using Java's built-in `System.currentTimeMillis()` method. We used a MacBook Pro with an Intel Core i5 CPU @ 2.6 GHz, 8 GB DDR3 @ 1600 MHz RAM, 256 GB SSD hard drive, macOS Sierra version 10.12.6, with Java version 1.8.0.66 64bit.

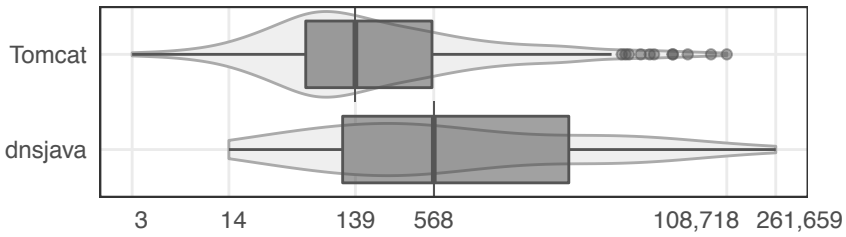

Figure 9: The distribution of the time taken for our technique

As it is observed, the median time taken for completing the process on DNSJAVA and TOMCAT are 139 and 568 milliseconds, respectively. This shows that the detection process is very fast. We can see that there are some outliers in the figure, particularly for the TOMCAT project. Looking at the commits corresponding to these outliers, we understood that these are the cases with a large number of changes on which REFACTORINGMINER needed to spend more time to detect the

refactoring instances, since the number of statements that needed to be matched were large, and for all of them there would be needed to perform different computations (including the computation of the Levenshtein Edit Distance). Note that, in none of these cases our part of the algorithm for detecting the instances of Rename Local Variable refactoring was the time-consuming part, and in all of them it was RefactoringMiner which needed more time to compute other types of refactorings. The maximum time spent on a commit in the dnsjava and Tomcat projects are about four and two minutes, respectively.

4.3 Discussion

In order to better understand the observed results from the two tools, in this section, we discuss some of the limitations of both tools in detecting instances of Rename Local Variable refactorings, together with the real-world examples where these limitations actually made the tools under-perform.

4.3.1 Limitations of REPENT

Relying on textual similarity

As mentioned in Section 2.3.3, the algorithm used in REPENT builds upon the output of a textual diff tool. We argue that this is the main issue with REPENT. One reason is that, textual diff tools might match lines containing totally irrelevant statements when the location of the statements is changed drastically. This can bring two issues for REPENT:

- It prevents REPENT from feeding its post-processing step (see Section 2.3.3) with right rename candidates, thus reducing its recall.
- The post-processing step fails to filter out the wrong candidates, thus reducing its precision.

Figure 10 depicts an example where drastic change in the location of statements can make REPENT to incorrectly report a Rename Local Variable refactoring. The change shown in this figure was found in Tomcat[3].

In this example, REPENT reports that the local variable `msg` in method `invoke()` was renamed to `imsg` in method `sendInvalidSessions()`. As it can be observed, the body of the method `sendInvalidSessions()` is completely extracted from the method `invoke()` with a minor change in the exception part. It is clear that the variable `imsg` is moved to method `sendInvalidSessions()` as a part of the extraction process. Moreover, the variable `msg` in the method `invoke()` is obviously

[3] `https://github.com/MatinMan/tomcat-code/commit/5b67c90bd4fe4b670f7244d4bc518c240a2b635a#diff-8fdbe4a7a9af89e447011e7b1fbdb337L162`

Figure 10: Drastic relocation of statements leads to a False Positive in REPENT

extracted to the method `sendSessionReplicationMessage()`. Thus, we cannot say that the variable `imsg` is renamed to `msg`. In this example, we have different clues pointing that this is a False Positive:

- First, the variable `imsg` already exists in the old revision and extracted to another method in the refactored revision.
- Second, the variable `msg` is also extracted to another method.

Notice that the main reason for this False Positive is the location of variable `msg` in the old revision and the location of the variable `imsg` in the refactored revision. As the textual diff tools try to find the similar text in the closest location, REPENT identifies it as a good candidate and its post-processing step cannot filter it out. By simply swapping the location of the methods `sendSessionReplicationMessage()` and `sendInvalidSessions()`, REPENT does not report this case as a Rename Local Variable, removing the False Positive.

Recall from Chapter 3 that we have defined special rules for avoiding such False Positives in our technique.

Distinguishing Merged and Split Local Variables

There are two possible scenarios where local variables can be marked as renamed incorrectly, while another change could have actually happened:

- In the new revision, one variable is responsible for doing the task that two variables used to do in the old revision, i.e., local variable *merging*,
- A variable used to do two tasks in the old revision, and a developer *splits* it into two variables in the new revision.

We found out that REPENT has problems in distinguishing merged and splitted local variables. An example of a local variable being split is illustrated in Figure 11. This change was found in the DNSJAVA project[4].

```
public void serveUDP(InetAddress addr, short port) {
    ...
    while (true) {
        byte[] in = new byte[udpLength];
        DatagramPacket dp = new DatagramPacket( in, in.length);

        try {
            sock.receive(dp);
        } catch (InterruptedIOException e) {...}



        byte[] out = response.toWire();
        dp = new DatagramPacket(out, out.length, dp.getAddress(),
                                                 dp.getPort());
        sock.send(dp);
    }
}
```

Before

```
public void serveUDP(InetAddress addr, short port){
    ...
    byte[] in = new byte[udpLength];
    DatagramPacket dpin = new DatagramPacket(in, in.length);
    DatagramPacket outdp;
    while (true) {
        indp.setLength(in.length);
        try {
            sock.receive(dpin);
        } catch (InterruptedIOException e) {...}


        byte[] out = response.toWire();
        outdp = new DatagramPacket(out, out.length,
                    indp.getAddress(), indp.getPort());
        sock.send(outdp);
    }
}
```

After

Figure 11: Split local variable creates a False Positive in REPENT

Here, the local variable `dp` in the old revision (i.e., in the left side of Figure 11) was used to either store the received `DatagramPacket` or the response `DatagramPacket`. In the new revision (i.e., in the right side of Figure 11) the variable is replaced by two variables, `indp` and `outdp`, the former storing the received `DatagramPacket`, and the latter storing the response `DatagramPacket`. By looking at both replacements, it is hard to decide whether the old `dp` variable was renamed to `indp` or `outdp`. Moreover, from the *semantics* point of view, the new names support the scenario of local variable splitting rather than renaming.

Failing in the Detection of Renamed Methods

Another reason that affects the precision and recall of REPENT is failing in correctly detecting renamed methods. There are cases where REPENT reports an instance of Rename Local Variable

[4] https://github.com/MatinMan/RefactoringDatasets/commit/c8e2ae0a58d0d494a13d7a2bcf9cc6d40bb2fc18#diff-a5423b0397f01cc1521ed2e866c9dd6eL622

refactoring in two methods which do not correspond to each other, while REPENT found the methods to be the same (one is renamed to the other). Although the Rename Local Variable instance seems to be detected correctly, we cannot accept it as a True Positive, since the methods in which the local variables are declared, are totally unrelated.

Note that, in general, any refactoring detection technique that fails to identify other renamed entities (methods, classes, etc.) will suffer from the same problem.

4.3.2 Limitations of Our Approach

The investigation of the False positives and False Negatives of our technique revealed interesting cases, which make room for future enhancements of our approach. In this section we discuss some of those cases in detail.

Chains of Extracted Methods

One of the cases where our approach is unable to correctly identify instances of a Rename Local Variable refactoring is when *a chain of extracted methods* occurs. The chain of extracted methods is a scenario in which method `c()` is extracted from method `b()`, and, at the same revision, method `b()` is extracted from method `a()`. As mentioned in Chapter 3, we build upon the extracted methods reported by RefactoringMiner. Since RefactoringMiner only looks for one level of method extraction, we miss such cases.

Figure 12 illustrates such scenario. The code shown here has been found in dnsjava[5].

Here, the variable `c` is renamed to `rrclass`, but the methods to which these variables belong have different signatures. The original method (labeled ①) is relocated in the refactored revision. It is observed that in the new revision, a method chain with a depth of two is extracted from the method `newRecord()` (in Figure 12, these methods are labeled ② and ③, respectively). The renamed variable is occurred in the deepest level of the chain (method ③). As mentioned, RefactoringMiner cannot detect method ③ as an extracted method. Thus, we miss to report this case. On the other hand, REPENT can detect it since the extracted method is located in the same location in the source code, compared to where the method `newRecord()` was located in the old revision.

Failing to Match Statements or to Report Replacements by RefactoringMiner

If one statement undergoes radical changes in the new revision, e.g., the type, name, and the assignment expression change all together, RefactoringMiner is unable to match the statement in the old revision with the corresponding statement in the new revision. In some cases, the `replacements`

[5] `https://github.com/MatinMan/RefactoringDatasets/commit/6d5572b768fcb347a02c356b837be30b65d64c5c#diff-8de9abcca25e02e169a7d555618174afL32`

Figure 12: Chain of extracted methods creates False Negatives in our approach

function (see Chapter 3) is unable to find any replacement to make the statements similar enough to match. Figure 13 depicts such a case from Tomcat[6].

```
public void invokeNext(Request request,
                       Response response){
    ...
    Integer current = (Integer) state.get();

    int subscript = current.intValue();
    state.set(new Integer(subscript + 1));
    ...
}
```

Before

```
public void invokeNext(Request request,
                       Response response){
    ...
    PipelineState pipelineState = (PipelineState)
                                  request.getNote(STATE);
    int subscript = pipelineState.stage;
    pipelineState.stage = pipelineState.stage + 1;
    ...
}
```

After

Figure 13: Radical changes in the code creates False Negatives in our approach

This case was reported by REPENT as a Rename Local Variable instance, where the variable `current` is renamed to `pipelineState`. The refactoring was also validated by the authors of REPENT. Here, it is not easy at all to tell that the renaming has really occurred in the code, unless we know that the class `PipelineState` is created in the new revision of the code and now wraps the

[6] https://github.com/MatinMan/tomcat-code/commit/585bda28cfb5f41f34e5085ab0d98d0f962ad450#diff-211599ed20cdc2697f40283a0da2f315L483

`current` variable. The uses of the two variables are more or less consistent throughout the code.

When trying to match the old statement

```
Integer current = (Integer) state.get();
```

with the new statement

```
PipelineState pipelineState = (PipelineState) request.getNote(STATE);
```

each part of the old statement has been changed in the new one, making the job for RefactoringMiner to find a replacement that can match the two statements very difficult, if not impossible.

Another source of False Positives in our technique is the methods that have undergone radical changes. For example, having a method expanded from five lines of code to 30 lines of code make it very difficult to find the best matches between the statements.

Failing to Identify Other Entity Renamings

Our approach (and as mentioned, any other refactoring detection technique) can suffer from the inability of correctly identifying other renamings that may have occurred on the code entities.

For example, when RefactoringMiner fails to identify renamed classes, it will be unable to detect Rename Local Variable instances occurring on the variables declared in the classes that are renamed.

Moreover, we observed a few cases where RefactoringMiner was unable to detect rename methods, or incorrectly reported renamed methods. Just like the case of type renaming, failing to detect renamed methods causes our approach to miss instances Rename Local Variable refactorings occurring in the methods that are being renamed. Wrong method rename detection, on the other hand, might cause our technique to report Rename Local Variable instances for two unrelated methods, i.e., more False Positives.

4.4 Threats to Validity

4.4.1 Internal Validity

Since we have built our approach on top of RefactoringMiner, its performance directly affects the performance of our technique. For example, the matching of statements and identification of replacements implemented in RefactoringMiner might cause missing Rename Local Variable instances. We tried to mitigate this by comparing the results of our approach with the state-of-the-art

tool, REPENT, and discovering the cases where RefactoringMiner missed to report the correct information, and designing algorithms that could compensate the inadequacies of RefactoringMiner. For instance, as discussed in Section 3.2.1, we do not limit ourselves to the replacements reported by RefactoringMiner and try to check other possible replacements that can lead to correctly identifying Rename Local Variable instances.

Moreover, the accuracy of the tools is computed based on the result of manual validation. As each validation is based on the human understanding of code changes, it can be different from human to human. Moreover, the developer who actually applied the refactorings might have a different opinion compared to our validation. To mitigate this threat, for the complicated cases, two researchers independently analyzed them. In the case of disagreements, the case was extensively discussed and even in a few cases, a third person was also involved in the justification.

Using the agreements between two tools for creating the reference corpus has this threat that there could be cases reported incorrectly by both tools (i.e., the same False Positives reported by both tools). However, since we systematically applied manual validation on all the cases and did not find any such case, we are sure about the validity of the cases in the constructed reference corpus.

As for the recall, we acknowledge that the results of two tools might not be enough to assess the completeness of the reference corpus. Yet, since we used two different refactoring detection approaches (i.e., a threshold-based approach and a non-threshold-based one), we can argue that they can complement each others' results. As it was observed, REPENT detected 429 rename local variable instances in the studied systems, and by running RefactoringMiner on the same systems, this number increased to 524. Of course, having more tools might help to expand the reference corpus even more.

Note that, we did not choose the cases to be inserted into the reference corpus, and we randomly selected the two projects to evaluate our approach. As a result, our study does not suffer from selection bias that can threaten the internal validity of the study.

4.4.2 External validity

To make sure that the results of the study are generalizable, we used two large-scale open-source projects, which were used in a previous study. In addition, to allow other researchers to replicate this study, we have made our tool[7] and the reference corpus[8] that we used to evaluate it available online.

As mentioned, the authors or REPENT performed random sampling on the results for manual validation. This can negatively impact the external validity. To mitigate this threat, instead of

[7] https://github.com/MatinMan/RefactoringBenchmark

[8] dnsjava: `https://github.com/MatinMan/RefactoringDatasets/blob/dnsjava/dnsjava-code/RLV-DNS.json` and Tomcat: `https://github.com/MatinMan/tomcat-code/blob/master/RLV-Tomcat.json`

sampling the results reported by the two approaches, all cases were manually validated.

4.5 Chapter Summary

In this chapter, we explained how we designed a study to evaluate the accuracy and performance of our approach. We compared the proposed approach to REPENT using an unbiased reference corpus, and showed that our approach is superior in terms of precision and recall. On average, our approach can detect instances of Rename Local Variable refactorings with a precision and recall of 87.5% and 86.6%, respectively, compared to REPENT which has a precision and recall of 81.8% and 79.3%.

In the next chapter, we briefly describe the internals of our automated accuracy computation tool, called RefBenchmark.

Chapter 5

RefBenchmark: Automated Accuracy Computation Framework for Refactoring Detection tools

In Chapter 2, we described several different tools that detect refactoring instances in the history of software systems [DCMJ06, SV17, DDN00, GZ05, FWPG07, PRSK10, AEP+14]. These tools apply various approaches to identify refactorings, and their accuracy is computed over different datasets. However, to understand the tools' strengths and weaknesses, and to make a fair comparison between them, it is necessary to run them on the same dataset. This would allow us to identify the best approach for the detection of refactorings in terms of precision and recall.

In this chapter, we introduce and describe RefBenchmark, an automatic approach to compare the accuracy of refactoring detection tools. RefBenchmark can compare the results of the tools against a given reference corpus. If a reference corpus is not available, RefBenchmark runs the tools and constructs a dynamically generated reference corpus based on the agreement between the tools.

The tool allows configuring the *agreement criterion*. For example, for a conservative agreement, it is required that all tools agree on a detected instance, while a more loose agreement would accept a detected refactoring instance if there is, say, at most one disagreement.

In the current implementation, RefBenchmark already includes the necessary functionality to automatically call RefDiff [SV17] and RefactoringMiner over a set of repositories. Users can easily extend RefBenchmark to include any other tool that detects refactorings. Moreover, one can implement *providers* (see Section 5.1.1) to import the results of other refactoring detection tools

given in any format (e.g., an `XML` or a `JSON` file) and compare it with the results of other tools. For example, we have implemented a provider for Ref-Finder. Also, since REPENT was not available, we could not implement the functionality to automatically call it for this thesis. Instead, the Rename Local Variable instances existing in the results provided by its authors [AEP+14] were converted to a `json` file and then imported into RefBenchmark.

RefBenchmark was successfully used in a previous study [TME+18] to build the most comprehensive reference corpus of refactorings to date (which, of course, excludes Rename Local Variable instances). In the following sections, we describe the internals of RefBenchmark in more details.

5.1 Design

Figure 14 shows an overview of RefBenchmark. As it is observed, RefBenchmark consists of two main components: the `Mediator` and the `Evaluator`.

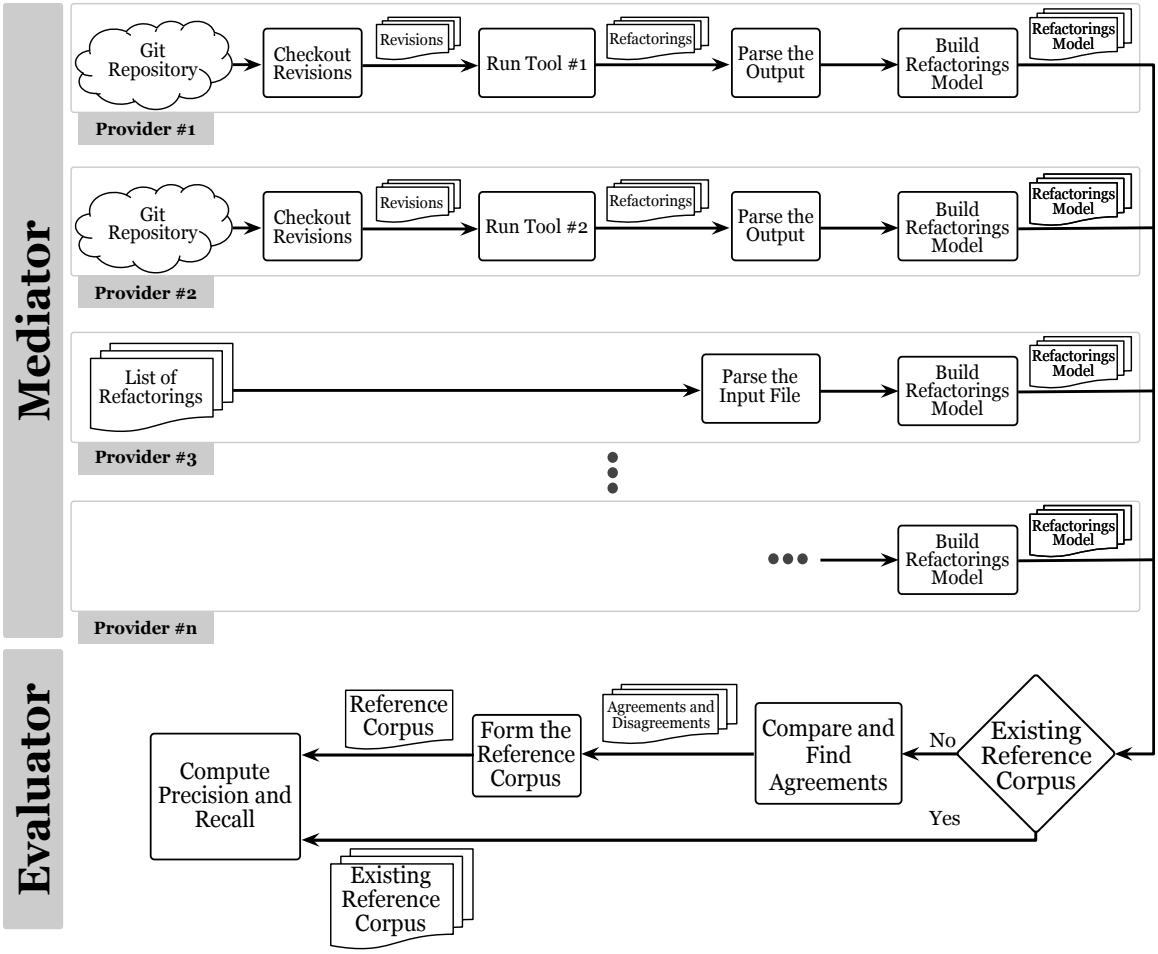

Figure 14: Design of RefBenchmark

The `Mediator` component is responsible to call a specific refactoring detection tool, parse its results and convert them to a unified *refactoring model*. The refactoring model is basically an abstract, object representation for refactoring operations. This model frees us from requiring to work with heterogeneous output formats from different refactoring detection tools.

Given different refactoring models, the `Evaluator` component compares these models against each other given an agreement criterion, or against a refactoring model that corresponds to an existing reference corpus. This component is responsible for computing the accuracy of the tools.

In the following, we explain the components of the tool in more details. First, we describe the structure of the `Mediator`. Next, we explain our approach for comparing the tools' results and calculating the accuracy measures using the `Evaluator`.

5.1.1 Mediator

The ultimate goal of the `Mediator` is to create a set of refactoring models for the results of the tools. It contains the `Provider` class that needs to be extended for each refactoring detection tool. As mentioned before, we have already extended the class `Provider` for four tools: RefFinder, RefDiff, REPENT, and RefactoringMiner. The subclasses of `Provider` implement the necessary methods for creating the refactoring models. In the following, we briefly explain the functionalities that a provider should implement for each new refactoring detection tool.

Cloning Repositories A provider should implement the necessary functionality to clone a repository from given URL. For example, one can use `JGit`[1] library, which is an implementation of the Git version control system's API to clone a repository and process it in Java. The cloned repositories will be fed to the tools for refactoring detection.

Collecting the Detected Refactorings After cloning a repository, the provider should specify how to invoke a specific tool to detect all the refactorings on the given commits.

Parsing the Output of the Tools After finding refactoring instances, the provider needs to specify a way to extract the necessary information from the output of the tool, for each of the identified refactoring instances. Each tool's output can have its own specific format. For instance, one tool can provide the information about the detected refactorings in a `JSON` format, the other tool might use `XML` files, or just plain text.

The provider extracts important information (e.g., the location of the occurred refactorings in terms of method and class names, and refactoring types) from the refactoring instances in the results. The provider can use regular expressions to extract information from the results. For

[1] `https://www.eclipse.org/jgit/`

example, for 14 types of refactorings supported by RefactoringMiner which are reported in particular string formats, the corresponding provider uses regular expressions to extract the necessary information. Moreover, since the provider invokes refactoring detection tools automatically, it might have complete access to the refactoring model of the corresponding tool, and thus it can simply adapt the tool-specific refactoring model to the RefBenchmark's refactoring model. This indicates that the provider could avoid the use of regular expressions. However, invoking a refactoring detection tool is a costly task (in terms of time and resources), and thus the regular expressions are useful when the detection results are already available in an output format.

Constructing the Refactoring Models After the provider extracts the information for each of the refactoring instances, it represents the results in a unified format, i.e., the refactoring models. As shown in Figure 15, each of the supported refactoring types by RefBenchmark is modeled by a designated class. Currently, we have implemented these classes for the following refactoring types: Extract, Extract-and-Move, Inline, Rename, and Move Method, Move and Rename Class, Extract Interface and Superclass, Move Attribute, Rename Local Variable, and a special class for the refactorings which are supported by a subset of tools, i.e., the `NotComparable` class. Note that, we have excluded classes related to Pull-Up and Push-Down Method and Attribute since their structure are similar to the corresponding Move Method/Attribute refactorings.

In each class, we only store important information that is useful for the comparison. For example, for Extract Method refactoring instances, RefBenchmark includes a class that contains the following properties: the signature of the extracted method (i.e., `extractedMethod` field), the signature of the method from which this method is extracted (i.e., `sourceMethod`), and the class that the refactoring was applied in (i.e., `sourceClass`). Currently, RefBenchmark supports those refactorings detected by at least two of the four aforementioned tools.

Next, RefBenchmark stores all the processed results for each refactoring in a new list. To keep track of the refactorings that are detected in each software revision, RefBenchmark stores, for each tool, all the unified refactorings in an instance of a class that represents a revision, i.e., `RevisionResult`. As shown in Figure 15, class `DetectionTool` which stores the information about the detection tools has a set of instances of the `RevisionResult` class. This way, RefBenchmark can differentiate refactorings that are detected in different revisions.

All the steps described above are mandatory only if a specific `Provider` needs to run a tool. Apart from this case, there is the scenario where we have the results of a tool and thus we can load and parse the refactoring and built the required refactoring model. In such case, method

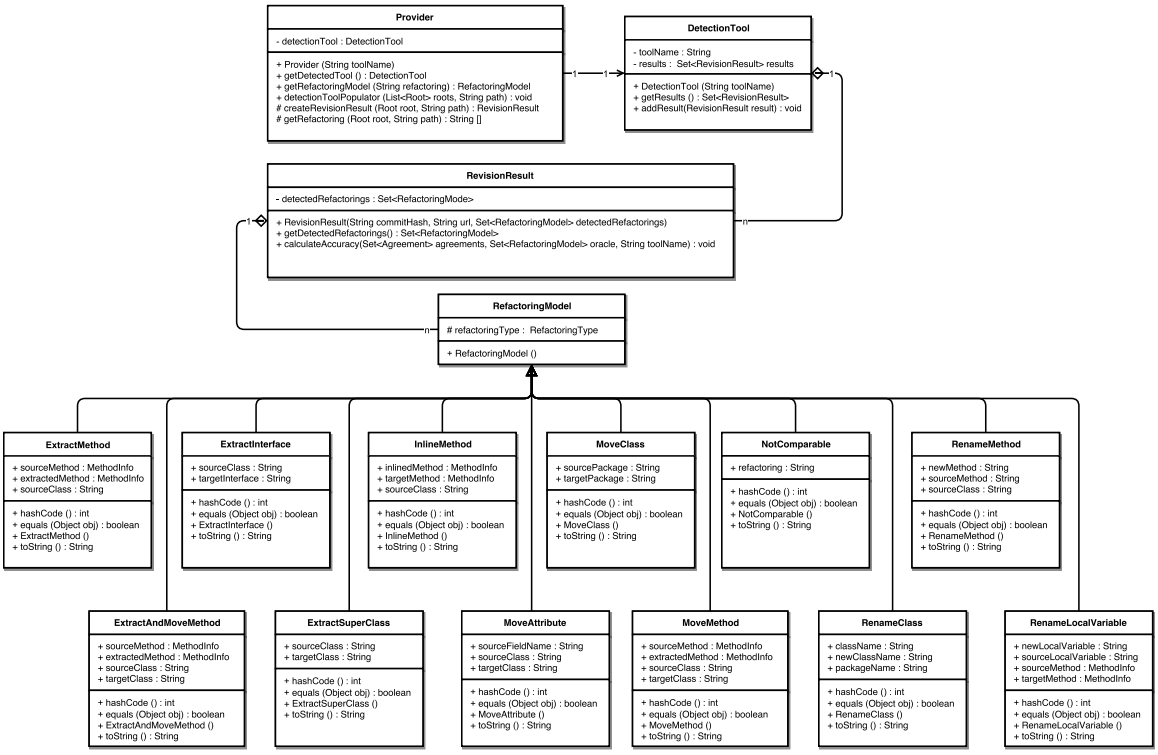

Figure 15: Class diagram for the Providers in RefBenchmark

`detectionToolPopulator()` needs to be implemented for populating the `results` in the class `DetectionTool`.

5.1.2 Evaluator

The input to the evaluator is a set of refactoring models that need to be compared for identifying common refactoring instances. The comparison is done by overriding the methods `hashCode()` and `equals()` in all the subtypes of `RefactoringModel`. In the `equals()` method, we compare all the corresponding fields of the same class instances. As an example, suppose that RefactoringMiner and RefDiff both detect an Extract Method Refactoring instance in the same revision of a system under analysis. After unifying their results as two different instances of the `ExtractMethod` class, our approach invokes the `obj1.equals(obj2)` to compare the two objects. If the values corresponding to the instance variables of the two classes are the same, we consider the two refactoring instances as identical.

Class `BenchmarkHandler` (Figure 16) is responsible for creating instances of the class `Agreement`. For each refactoring model detected by a tool, an instance of the `Agreement` class is created and the tool name is added to the `detectionToolsNames` instance variable. If another tool detects an already-detected refactoring, there is no need to create a new instance of class `Agreement`, only the

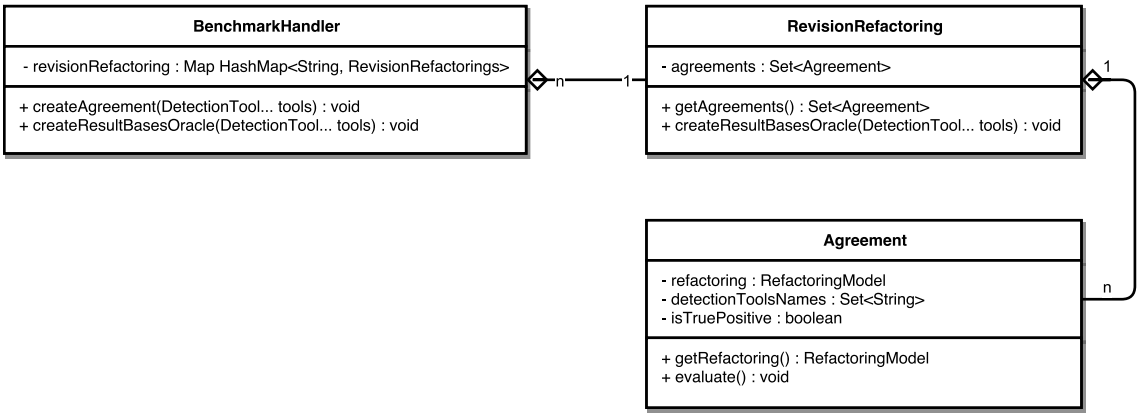

Figure 16: Class diagram for `BenchmarkHandler` in RefBenchmark

name of the tool will be added to the `detectionToolsNames` instance variable.

For an instance of the class `Agreement` (which indeed wraps a refactoring model), if the number of tools that detect the corresponding refactoring (i.e., the size of `detectionToolsNames`) is more than a user-defined threshold, the boolean instance variable `isTruePositive` is set to `true`. Class `RevisionRefactoring` stores a set of agreements for a specific revision, and the class `BenchmarkHandler` maps the commit hash to the `RevisionRefactoring`, therefore, for each given revision commit, the refactorings detected and the agreement between the tools can be reported.

5.2 Chapter Summary

We summarize the contributions of RefBenchmark as the following:

- It models 15 popular refactorings, and is easily extendable for other types of refactorings.
- It provides a mechanism to embed other detection tools by extending the `Provider` class.
- It allows comparing the tools based on configurable settings (e.g., using a threshold for tool agreement or by loading an existing reference corpus).
- It has been designed with a rich API and can be added as a library to any refactoring detection tool, so that it can provide the functionality for comparing its results with other refactoring detection tools.

The next chapter concludes this thesis.

Chapter 6

Conclusions and Future Work

In this thesis, we extended RefactoringMiner to identify the instances of Rename Local Variable refactorings applied between software revisions. Our approach has been built upon the statement matching algorithm of RefactoringMiner which, for each statement in the old revision of the code, identifies the *best* matching statement in the new revision. When the statements across the two revisions are not identical, RefactoringMiner attempts to match them by findings a set of *replacements* that can make the statements textually similar. We exploit the reported replacements to find the instances of Rename Local Variable refactoring. We provide the necessary rules for refactoring detection, and discuss the exceptional cases where we need to take further steps to improve the RefactoringMiner's identified replacements to accurately detect Rename Local Variable instances.

We evaluated our approach by conducting a study on two open source software systems, namely Tomcat and dnsjava. Our approach reported Rename Local Variable instances with precision and recall of 87.5% and 86.6%, respectively. Moreover, our approach outperformed the state-of-the-art tool, REPENT, in both precision and recall. It is important to notice that our approach does not require any pre-defined thresholds, does not require the systems under analysis to be fully built, and does not depend on any specific IDE, in contrast to all previous approaches.

To automate the construction of a refactoring reference corpus for the purpose of evaluation, we introduced RefBenchmark that can automate the entire refactoring identification workflow (i.e., it can invoke different refactoring detection tools, collect their results, and make a unified model of the detected refactorings) in order to compute the accuracy measures (i.e., precision and recall) based on a user-defined tool agreement criterion. RefBenchmark currently supports 15 different types of refactorings, can analyze the results of multiple state-of-the-art refactoring detection tools

namely, Ref-Finder [PRSK10, KGLR10], RefDiff [SV17], REPENT [AEP+14], and RefactoringMiner [TME+18], and is easily extensible to more refactoring types and refactoring detection tools.

There is a lot of room for improvement both for our Rename Local Variable refactoring detection technique and also for RefBenchmark, which we discuss in the followings.

- **Possible improvements on RefactoringMiner to improve the accuracy of our technique:**
  - As mentioned before, RefactoringMiner is unable to detect nested method extractions, and thus, our approach cannot detect the local variables renamed in a long sequence of nested extracted methods. By mitigating this issue in RefactoringMiner, we can increase the recall of our approach.
  - Another limitation of RefactoringMiner is related to missing renamed methods/types, and therefore, our technique will miss local variables being renamed inside them. We intend to improve RefactoringMiner by checking the statements that refer to the renamed types and methods. For example if method `a()` is renamed to `b()`, all its usages should be renamed too. We can use this knowledge to improve RefactoringMiner to detect renamings more accurately.
- **Identifying instances of other renaming refactorings:**
  - Rename Parameter and Rename Field refactorings can be potentially detected using a similar approach to detecting Rename Local Variable refactoring instances. We intend to support the detection of both refactorings in the future. To achieve this, the detection rules should be changed. Also, we believe there could be exceptional cases, like the ones that we discussed in Section 3.2.1 for Rename Local Variable refactoring, that need to be investigated for each of these refactoring types.
- **Expanding the reference corpus:**
  - In this thesis, we only evaluated our technique using two systems. While the selected case studies are representative of large systems, and potentially include a wide variety of possible Rename Local Variable instances, still there might be cases that we have missed. As mentioned before, the reason we did not use more systems for this thesis is that manual validation usually requires a lot of effort (several person-months). Thus, we are planning to extend gradually our reference corpus in the future. This will result to an even more complete reference corpus that can be used in other studies.

- **Possible improvements on RefBenchmark:**
  - RefBenchmark uses agreement to form a reference corpus in the current implementation. In our experiments, we observed several cases for which the validators were not completely sure about their decision (i.e., whether a case is really an instance of Rename Local Variable or not) and that's why there were sometimes extensive discussions among the validators, or a third validator was asked to give an opinion for a few cases. This can be true for all types of refactorings. Sometimes, the refactoring edits are interleaved with other maintenance changes, and the perception of a refactoring can differ from one person to another. We are going to add a *rating* feature to RefBenchmark, so that users can assign their confidence about each instance. Thus, instead of using only agreement, RefBenchmark can use the validators' feedback in order to form the reference corpus and sort the instances based on the confidence of the validators.
  - Each refactoring detection tool identifies some types of refactoring more accurately compared to other types. Giving *weight* to each type of refactoring identified by different tools is another improvement that can be done on RefBenchmark. For example, suppose that we observe the refactoring detection tool A identifies Extract Method refactoring instances much more reliably than tool B and C. If tool A detects an extract method refactoring, which is not detected by tools B and C, it is likely that this instance is a False Negative for tools B and C, and thus can be inserted to the reference corpus.

Bibliography

[AEP+14] V. Arnaoudova, L. M. Eshkevari, M. D. Penta, R. Oliveto, G. Antoniol, and Y. G. Guéhéneuc. REPENT: Analyzing the Nature of Identifier Renamings. *IEEE Transactions on Software Engineering*, 40(5):502–532, May 2014.

[APM04] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. An automatic approach to identify class evolution discontinuities. In *7th International Workshop on Principles of Software Evolution*, pages 31–40, 2004.

[ASK14] Everton L. G. Alves, Myoungkyu Song, and Miryung Kim. Refdistiller: A refactoring aware code review tool for inspecting manual refactoring edits. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 751–754, New York, NY, USA, 2014. ACM.

[BCG+10] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, and Nico Zazworka. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 47–52, New York, NY, USA, 2010. ACM.

[BCL+12] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. When does a refactoring induce bugs? an empirical study. In *Proceedings of the IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, SCAM '12, pages 104–113, Sept 2012.

[BDLMO14] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. *Recommending Refactoring Operations in Large Software Systems*, pages 387–419. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[BL76] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Syst. J.*, 15(3):225–252, September 1976.

[BLP+15] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, Sept 2015.

[BR09] Dagenais Barthélémy and Martin P. Robillard. Semdiff: Analysis and recommendation support for api evolution. In *2009 IEEE 31st International Conference on Software Engineering*, pages 599–602, May 2009.

[Bro97] A. Broder. On the Resemblance and Containment of Documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97, pages 21–, Washington, DC, USA, 1997. IEEE Computer Society.

[BTF05] Ittai Balaban, Frank Tip, and Robert M. Fuhrer. Refactoring support for class library migration. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 265–279, 2005.

[CB16] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 332–343, Sept 2016.

[CFF+17] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. How does refactoring affect internal quality attributes?: A multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, SBES'17, pages 74–83, New York, NY, USA, 2017. ACM.

[CGM+17] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 465–475, New York, NY, USA, 2017. ACM.

[CKO10] Jeromy Carriere, Rick Kazman, and Ipek Ozkaya. A cost-benefit framework for making architectural decisions in a business context. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 149–157, New York, NY, USA, 2010. ACM.

[CRGMW96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change Detection in Hierarchically Structured Information. *SIGMOD Rec.*, 25(2):493–504, June 1996.

[Dal15] Jehad Al Dallal. Identifying refactoring opportunities in object-oriented code: A systematic literature review. *Information & Software Technology*, 58:231–249, 2015.

[DBG+15] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. In *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering*, SANER '15, pages 341–350, March 2015.

[DCMJ06] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP '06, pages 404–428, Berlin, Heidelberg, 2006. Springer-Verlag.

[dCMS+16] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 2016.

[DDN00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 166–177, New York, NY, USA, 2000. ACM.

[DJ06] Danny Dig and Ralph Johnson. How Do APIs Evolve? A Story of Refactoring. *J. Softw. Maint. Evol.*, 18(2):83–107, March 2006.

[DMJN08] Danny Dig, Kashif Manzoor, Ralph E. Johnson, and Tien N. Nguyen. Effective software merging in the presence of object-oriented refactorings. *IEEE Transactions on Software Engineering*, 34(3):321–335, May 2008.

[DRW14] Steven Davies, Marc Roper, and Murray Wood. Comparing text-based and dependence-based approaches for determining the origins of bugs. *Journal of Software: Evolution and Process*, 26(1):107–139, 2014.

[FG06] Beat Fluri and Harald Gall. Classifying change types for qualifying change couplings. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 35–45, 2006.

[FGL12] Stephen R. Foster, William G. Griswold, and Sorin Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 222–232, June 2012.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[FWPG07] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, November 2007.

[GDMH12] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. Reconciling manual and automatic refactoring. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 211–221, Piscataway, NJ, USA, 2012. IEEE Press.

[GMH14] Xi Ge and Emerson Murphy-Hill. Manual refactoring changes with automated refactoring validation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 1095–1105, New York, NY, USA, 2014. ACM.

[GSMH14] Xi Ge, Saurabh Sarkar, and Emerson Murphy-Hill. Towards refactoring-aware code review. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '14, pages 99–102, New York, NY, USA, 2014. ACM.

[GSWMH17] Xi Ge, Saurabh Sarkar, Jim Witschey, and Emerson Murphy-Hill. Refactoring-aware code review. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '17, 2017.

[GW05] Carsten Görg and Peter Weissgerber. Detecting and visualizing refactorings from software archives. In *Proceedings of the 13th International Workshop on Program Comprehension*, IWPC '05, pages 205–214, May 2005.

[GZ05] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Software Eng.*, 31(2):166–181, 2005.

[HD05] Johannes Henkel and Amer Diwan. Catchup!: capturing and replaying refactorings to support API evolution. In *27th International Conference on Software Engineering*, pages 274–283, 2005.

[KCK11] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 151–160, New York, NY, USA, 2011. ACM.

[KGLR10] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. Ref-finder: A refactoring reconstruction tool based on logic query templates. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 371–372, New York, NY, USA, 2010. ACM.

[KHFG16] István Kádár, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy. A manually validated code refactoring dataset and its assessment regarding software maintainability. In *Proceedings of the 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE '16, pages 10:1–10:4, New York, NY, USA, 2016. ACM.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002.

[KMPY06] Ronny Kolb, Dirk Muthig, Thomas Patzke, and Kazuyuki Yamauchi. Refactoring a legacy component for reuse in a software product line: A case study: Practice articles. *J. Softw. Maint. Evol.*, 18(2):109–132, March 2006.

[KNG07] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society.

[KR11] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 351–360, New York, NY, USA, 2011. ACM.

[KZN14] M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, July 2014.

[KZPW06] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[LAK+17] Olaf Leßenich, Sven Apel, Christian Kästner, Georg Seibt, and Janet Siegmund. Renaming and shifted code in structured merging: Looking ahead for precision and performance. In *Proceedings of the 32Nd IEEE/ACM International Conference on*

Automated Software Engineering, ASE 2017, pages 543–553, Piscataway, NJ, USA, 2017. IEEE Press.

[Lev66] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[LLLW15] H. Liu, Q. Liu, Y. Liu, and Z. Wang. Identifying renaming opportunities by expanding conducted rename refactorings. *IEEE Transactions on Software Engineering*, 41(9):887–900, Sept 2015.

[MBP+17] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Arena: An approach for the automated generation of release notes. *IEEE Transactions on Software Engineering*, 43(2):106–127, Feb 2017.

[MHB08] Emerson Murphy-Hill and Andrew P. Black. Refactoring Tools: Fitness for Purpose. *IEEE Software*, 25(5):38–44, Sept 2008.

[MHPB12] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, Jan 2012.

[MHT00] G. Malpohl, J. J. Hunt, and W. F. Tichy. Renaming detection. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering*, ASE 2000, pages 73–80, Sept 2000.

[MHT03] Guido Malpohl, James J. Hunt, and Walter F. Tichy. Renaming detection. *Automated Software Engineering*, 10(2):183–202, Apr 2003.

[MN13] Anas Mahmoud and Nan Niu. Supporting requirements traceability through refactoring. In *Proceedings of the 21st IEEE International Requirements Engineering Conference*, RE '13, pages 32–41, July 2013.

[MN14] Anas Mahmoud and Nan Niu. Supporting requirements to code traceability through refactoring. *Requirements Engineering*, 19(3):309–329, Sept 2014.

[MSAS06] Raimund Moser, Alberto Sillitti, Pekka Abrahamsson, and Giancarlo Succi. Does refactoring improve reusability? In *Proceedings of the 9th International Conference on Reuse of Off-the-Shelf Components*, ICSR'06, pages 287–297, Berlin, Heidelberg, 2006. Springer-Verlag.

[NCV+13] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. A comparative study of manual and automated refactorings. In *Proceedings of the 27th*

European Conference on Object-Oriented Programming, ECOOP'13, pages 552–576, Berlin, Heidelberg, 2013. Springer-Verlag.

[NVC+12] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig. Is it dangerous to use version control histories to study source code evolution? In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 79–103, Berlin, Heidelberg, 2012. Springer-Verlag.

[Opd92] William F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, Champaign, IL, USA, 1992.

[PRK10] Kyle Prete, Napol Rachatasumrit, and Miryung Kim. Catalogue of Template Refactoring Rules. Technical report, The University of Texas at Austin, 04 2010.

[PRSK10] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. Template-based reconstruction of complex refactorings. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Sept 2010.

[PZODL17] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. An exploratory study on the relationship between changes and refactoring. In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, pages 176–185, Piscataway, NJ, USA, 2017. IEEE Press.

[RD03] F. Van Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In *Proceedings of the Sixth International Workshop on Principles of Software Evolution*, pages 126–130, Sept 2003.

[RK12] Napol Rachatasumrit and Miryung Kim. An empirical investigation into the impact of refactoring on regression testing. In *Proceedings of the 28th IEEE International Conference on Software Maintenance*, ICSM '12, pages 357–366, Sept 2012.

[RSG08] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. On the relation of refactorings and software defect prediction. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 35–38, New York, NY, USA, 2008. ACM.

[SGMHJ13] Gustavo Soares, Rohit Gheyi, Emerson Murphy-Hill, and Brittany Johnson. Comparing approaches to analyze refactoring activity on software repositories. *Journal of Systems and Software*, 86(4):1006–1022, Apr 2013.

[SPDZ15] Quinten David Soetens, Javier Pérez, Serge Demeyer, and Andy Zaidman. Circumventing refactoring masking using fine-grained change recording. In *Proceedings of the 14th International Workshop on Principles of Software Evolution*, IWPSE 2015, pages 9–18, New York, NY, USA, 2015. ACM.

[STV16] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '16, pages 858–870, New York, NY, USA, 2016. ACM.

[SV17] Danilo Silva and Marco Tulio Valente. Refdiff: Detecting refactorings in version histories. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 269–279, Piscataway, NJ, USA, 2017. IEEE Press.

[SZZ05] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[TDX07] Kunal Taneja, Danny Dig, and Tao Xie. Automated Detection of API Refactorings in Libraries. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 377–380, New York, NY, USA, 2007. ACM.

[TME+18] Nikolaos Tsantalis, Mohammad Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[TPB+17] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4), 2017.

[VCN+12] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 233–243, June 2012.

[WD06a] Peter Weissgerber and Stephan Diehl. Are refactorings less error-prone than other changes? In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 112–118, New York, NY, USA, 2006. ACM.

[WD06b] Peter Weissgerber and Stephan Diehl. Identifying refactorings from source-code changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 231–240, Sept 2006.

[WS08] Chadd Williams and Jaime Spacco. Szz revisited: Verifying when changes induce fixes. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, DEFECTS '08, pages 32–36, New York, NY, USA, 2008. ACM.

[XS05] Zhenchang Xing and Eleni Stroulia. Umldiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 54–65, New York, NY, USA, 2005. ACM.

[XS06a] Z. Xing and E. Stroulia. Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 458–468, Sept 2006.

[XS06b] Zhenchang Xing and Eleni Stroulia. Refactoring detection based on umldiff change-facts queries. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 263–274, Washington, DC, USA, 2006. IEEE Computer Society.

[XS07] Zhenchang Xing and Eleni Stroulia. Api-evolution support with diff-catchup. *IEEE Transactions on Software Engineering*, 33(12):818–836, Dec 2007.

[XS08] Zhenchang Xing and Eleni Stroulia. The JDEvAn tool suite in support of object-oriented evolutionary development. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 951–952, New York, NY, USA, 2008. ACM.