JsDiffer: Refactoring Detection in JavaScript

Mosabbir Khan Shiblu

A Thesis
in
The Department
of
Computer Science and Software Engineering

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

November 2022

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Mosabbir Khan Shiblu**
Entitled: **JsDiffer: Refactoring Detection in JavaScript**

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

Master of Computer Science (Computer Science)

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

Signed by the final examining committee:

_____ Chair
Dr. Joey Paquet

_____ External

_____ Examiner
Dr. Joey Paquet

_____ Examiner
Dr. Weiyi(Ian) Shang

_____ Thesis Supervisor
Dr. Nikolaos Tsantalis

Approved by _____
Dr. LEILA KOSSEIM, Graduate Program Director

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

Abstract

JsDiffer: Refactoring Detection in JavaScript

Mosabbir Khan Shiblu

Refactoring refers to any code changes that improve the maintainability of the software system. Identifying such activities helps to understand the evolution and the relationship between two versions of a system. Therefore, automatic detection of refactorings applied in a system by comparing the source code between two snapshots has been an active research topic. Current state-of-the-art refactoring detection tools RefactoringMiner 2.0, however only supports programs written in Java language. On the other hand, JavaScript, despite being the most popular language, is supported by only one refactoring detection tool - RefDiff 2.0 which cannot detect variable level refactorings such as rename variable, rename parameter, etc.

In this study, we present JsDiffer, which supports **18** different refactoring operations including several variable related refactorings in JavaScript projects. Although the tool is inspired by RefactoringMiner, it differs quite a lot from refactoring miner in terms of structural mapping. We evaluated JsDiffer by constructing an oracle of **341** refactoring instances mined from 18 open-source JavaScript projects and compared it with RefDiff 2.0. Our results indicate that JsDiffer can achieve a precision and recall of 96% and 44% respectively. Although RefDiff 2.0 turned out to be the better of the two tools for JavaScript projects, our approach shows promising results in detection on Rename Variable refactorings where it achieves a precision of 88%.

Acknowledgments

I would like to express my gratitude and thanks to my supervisor, Dr. Nikolaos Tsantalis. His invaluable guidance and continuous support opened a new horizon of knowledge for me.

I would also like to thank my colleagues, Mohammad Sadegh Aalizadeh, Mehran Jodavi, and Ameya Ketkar who shared their experiences and were amazing in teamwork and helped me to learn a lot in my journey at Concordia.

Thank you.

Mosabbir Khan Shiblu

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Refactoring means restructuring source code to improve its maintainability without altering its functionality. It plays an important role in the modern software development life cycle. In addition to improving software maintainability, refactoring is frequently used to denote changes that improve software performance, software security, and even the energy consumption of a system [1], [2], [3]. In an agile environment, it enables the limited upfront design of the software to advance [4]. On the other hand, in test-driven development, it is regarded as a necessary activity in keeping the code-base compliant for further development [5].

A recent survey paper [6] found over 3,000 papers on refactoring topics which attests its popularity in modern research. Many researchers empirically investigated the benefits of refactorings by studying how the renaming of identifiers affects code readability [7], how and why developers rename identifiers [8], the impact of refactoring on code naturalness [9], the impact of refactoring on code smells [10], the co-occurrence of refactoring and self-admitted technical debt removal [11], and how the introduction of Lambda expressions affects program comprehension [12].

Therefore, by detecting refactorings in software projects, researchers can better understand software evolution. Earlier studies used such information to investigate the usage of refactoring tools [13], [14], the motivations behind refactoring [15], [16], [17], the risks associated with refactoring [15], [16], [18], [19], [20], and the effect of refactoring on code quality metrics [15], [16]. Additionally, the accuracy of source code evolution analysis can be improved by keeping track of refactorings, because files, classes, or functions may have their histories split by Move or Rename [21] refactorings. Lastly, according to a survey spanning 86 articles [22], the most desirable application for detecting changes that occurred between two program versions is extracting patterns of change and re-performing changes in different contexts. Knowing the applied refactoring operations in the version history of a system not only can help advance software evaluation research, but also can help developers in their practice. First, such information can be used to help resolve merge conflicts and improve code review time as many developers face difficulties when reviewing or integrating code changes with large refactoring operations [15]. It has been reported that refactoring activities can cause merge conflicts when merging development branches [23]. Therefore, if a tool can identify applied refactorings at commit level, it can possibly be used to resolve merge conflicts automatically. Additionally, various source code diff tools can use this information to match textually different code elements between two versions. Second, identified refactoring instances can be automatically appended in the commit message to let code reviewers know about the refactored components upfront. Third, if an API is refactored, corresponding refactorings could be applied to the client code automatically [24] [25]. Fourth, detected refactorings can be used to distinguish new lines of code representing a feature in a software development sprint from behavior-preserving changes, which can potentially help a project manager to monitor the progress of the current milestone. Last but not the least, refactoring detection tools can potentially be used to increase the accuracy of source code plagiarism detection tools.

Detecting refactorings is a challenging task due to the fact that developers rarely document such activities [26] and refactorings are often intertwined with other code changes making them even harder to distinguish. However, given the practical importance of refactoring in software development as well as its research potential, it is unsurprising that we have seen many automatic refactoring detection tools over the past few decades. RefactoringMiner 2.0 by Tsantalis et. al [27] currently represents the state of the art and is capable of detecting 40 refactoring types with a precision and recall of 99.6% and 94% respectively. Unfortunately, like the majority of the detection tools, it supports only the Java programming language. On the other hand, RefDiff 2.0 by Silva et. al [28] is the only tool capable of detecting refactorings in JavaScript projects. RefactoringMiner structurally matches statements of code thus capable of detecting lower level refactorings (such as rename /merge variable). On the other hand, the body of a function is represented as a bag of tokens by RefDiff and thus such lower-level structural information is never present after the tokenization.

Given the fact that JavaScript is currently the most popular programming language [1], a refactoring detection tool that can detect lower-level refactorings in JavaScript may provide more insight into its ecosystem, as developers tend to apply such refactorings more frequently than high-level refactorings [13]. Besides, there are significant differences between Java and JavaScript besides language grammar. For example, in JavaScript, functions are first-class citizen and can be stored and used as a variable. JavaScript code can reside outside of the body of a function. Moreover, unlike Java which is object-oriented and class-based, JavaScript projects can follow styles of OOP, functional programming, or a mix of both. Lastly, there are many super sets and extensions of JavaScript language (e.g., Typescript JSX) which are often intermixed with vanilla JavaScript code further complicating the task of parsing and modeling the source code.

[1] https://survey.stackoverflow.co/2022/

In this thesis, we present JsDiffer - the first tool that allows the detection of variable-related refactorings in JavaScript projects. Our tool is heavily inspired by RefactoringMiner 2.0; however, it has its own unique approach to detecting refactorings. JsDiffer takes two source code versions and outputs the refactorings performed between them. Currently, JsDiffer supports 18 types of refactorings including 1 JavaScript specific refactoring (Change Variable Kind refactoring). To evaluate the performance of JsDiffer, we ran it on 608 commits from 19 open-source JavaScript projects, and manually validated a portion of the detected refactoring instances. Overall, JsDiffer achieved a precision of 96% and a recall of 44%. Although the precision is higher than the current state-of-the-art, JsDiffer has performed poorly on recall. This result showed the potential of JsDiffer for detecting some specific refactoring types in JavaScript projects. Additionally, we compared JsDiffer with the only publicly available refactoring detection tool for JavaScript, namely RefDiff 2.0, and found that JsDiffer was able to find a few unique refactoring instances, which were not detected by the state-of-the-art.

1.2 Objectives and Contributions

The goal of our study is to develop a refactoring detection tool that does not rely on textual similarity, but rather on the structural similarity of two code elements. This can potentially help us with plagiarism detection when two source codes significantly changed between two versions. Besides, structural-based detection is the only approach that can detect low-level refactorings, such as Rename Variable.

Therefore, we employed a similar approach by RefactoringMiner 2.0 [27] which uses structural-based matching between two source codes and it is the current state-of-the-art for Java projects with precision and recall of 99.6% and 94%, respectively. In contrast to RefDiff 2.0, our approach does not rely on any similarity threshold, which makes it suitable for detecting refactorings between any two source codes that could have significant textual

differences. Moreover, our tool is the only tool that can detect variable-related refactorings such as Rename Variable and Rename Parameter.

This thesis makes the following contributions:

(1) We present the first tool, which is able to detect variable level refactorings between two JavaScript source codes.

(2) We support a total of 18 refactoring types including Change Variable Kind refactoring, which is the first JavaScript-specific refactoring in contrast to object-oriented refactorings, which are supported by most tools.

(3) We compared JsDiffer with the current state-of-the-art refactoring detection tool: RefDiff 2.0, where an empirical study on 608 commits from 19 open source JavaScript repositories were performed to create an oracle of 341 refactorings. This oracle is also publicly available to enable the replication of our experiments as well as future research on refactoring mining[2].

1.3 Outline

The rest of the thesis is structured as follows. Chapter 2 provides an overview and discussion of related works. Our approach to automatic detection of the refactorings that occurred between two program versions is presented in Chapter 3. The correctness and completeness of our approach are evaluated in Chapter 4, including the limitations of the proposed approach and threats to the validity of our study. Finally, in Chapter 5, we provide our conclusions and discuss possible related future research.

[2] https://docs.google.com/spreadsheets/d/1CQg7FO30GresGrHSGh0lxn5cWdrd_Y3EXEO98MPrQqE

Chapter 2

Related Work

In this section, we start by briefly talking about the spectrum of research exploring the practice of refactoring and later go into details about modern refactoring detection tools.

Leo Brodie [29] first mentioned the word "Refactoring" in his book "Thinking Forth", originally published in 1984. As per the author, "Factoring" and "Refactoring" were interchangeably used in the Forth community back then and he defined refactoring activities as identifying useful fragments that could be pulled out to make code more generally useful and more maintainable, or to eliminate duplication. In addition, the author also discussed many software development principles and practices that are still applicable to date.

However, it was Opdyke [30] who generalized refactorings as any source code transformations that improve the understandability and reusability of source code.

In an early work, by Mens and Tourwe [31], an overview of the existing research was provided in terms of refactoring practices and techniques, refactoring tools, and the effect of refactorings on the software process. Further studies studied the impact of refactoring on code quality [32] [33] [34] [35] [36] [37], detecting refactoring opportunities [38] [39], refactoring recommendations, [40] [41] [42], and automated refactoring tools [43] [44] [45] [46] [28] [47].

2.1 Refactoring Detection Approach

2.1.1 Detection Using Meta Data

Several studies have used repository metadata, such as commit messages, from version control systems to detect refactorings. Ratzinger et al. [48] searched for a predefined set of terms (e.g. "refactor") in commit messages to classify them as refactoring changes. Kim et al. [16] reported that in some cases, a branch may be created exclusively to refactor the code. Soares et al. [49] proposed an approach that can detect behavior-preserving changes by automatically generating and running test cases and can also be employed to classify behavior-preserving commits. Recently, Krasniqi and Cleland-Huang [50] implemented CMMiner which is capable of detecting 12 refactoring types based on analyzing commit logs provided by developers.

2.1.2 Detection By Tracking IDE Activity

Murphy-Hill et al. [13] tracked the usage history of refactoring commands available in Eclipse IDE using a plugin and found that developers had performed about 90% of their refactorings manually instead of opting for the refactoring tool. Additionally, developers often interleave refactorings with other behavior-modifying programming activities. Furthermore, developers rarely explicitly report their refactoring activities in commit messages.

Negara et al. [14] developed CodingTracker, which infers refactorings from continuous code changes with the help of a refactoring inference plugin. Using their tool, they constructed a large corpus of 5,371 refactoring instances performed by 23 developers working in their IDEs. Their approach reported precision and recall of 93% and 100%, respectively, for a sample of both manually and automatically performed refactorings.

Similar to CodingTracker [14], GhostFactor [51] and ReviewFactor [52] infer fully

completed refactorings by monitoring the fine-grained code changes in real-time inside the IDE. On the other hand, BeneFactor [53] and WitchDoctor [54] offer code completion by detecting ongoing manual refactorings.

2.1.3 Detection by Static Source Code Analysis

Static analysis is a widely popular and modern approach for finding differences between two versions of a software system. It has the advantage of being able to detect applied refactoring from software version histories. As our tool falls into this category, in this section, we will go in-depth about existing static refactoring detection tools.

Demeyer et al. [55] introduced the first strategy for identifying the refactored elements between two system snapshots. They defined four heuristics based on the changes of object-oriented source code metrics such as method size, class size, and the number of inherited or overwritten methods to identify refactorings of three general categories (Split/Merge Class, Move Method, and Split Method). For example, to detect Extract Superclass refactoring, they start by inspecting the increase in the inheritance hierarchy of a class to detect newly added classes. Then, they observed whether the number of methods and fields in the hierarchy has been decreased, but increased in the newly added class. To validate their technique, they applied it to different versions of three software systems. However, the precision of their approach was seemingly on the lower side, for example, for Move Method refactorings (limited to super, sub, and sibling classes) the reported average precision was 23%. One of the reasons for low precision is due to the partial overlapping of heuristics causing some false negative refactorings to be reported as false positives for other refactorings. On the other hand, the paper concluded that from the perspective of reverse engineering, the proposed heuristics were extremely useful to uncover where, how, and maybe why implementation had drifted from its original design.

Antoniol et al. [56] used an automatic technique based on Vector Space cosine similarity to compare identifiers in different classes in order to detect the renaming and splitting of classes. Since it's based on a similarity threshold, it does not perform very well for classes with many changes and may require threshold adjustment on a case basis.

Weißgerber and Diehl [57] developed the first technique for identifying class-level and locally-scoped refactorings i.e., refactorings that occur within one class, and thus, within the same file (e.g. Rename Method). Their approach first extracts and identifies added and deleted refactoring candidates (fields, methods, and classes) by parsing deltas and then comparing each pair's name similarity from a version control system. For ambiguous candidate pairs, it uses a clone detection tool CCFinder [58] to compare their bodies and then rank them. CCFinder is also configured to ignore whitespaces and comments and to match consistently renamed variables, method names, and references to members. Finally, they used random sampling to estimate the precision, whereas commit messages were inspected manually to find documented refactorings in order to compute the recall. Although they have achieved a good recall of 89%, it has been proven that commit messages are not reliable indicators of refactoring activity [50], [13]. Lastly, the authors stated that their technique is susceptible to multiple refactorings performed on the same entity.

Dig et al. [59] developed an Eclipse plug-in named RefactoringCrawler which initially uses a computationally inexpensive text-based similarity metric, Shingles encoding [60], to find possible refactoring candidates. Shingles act as "fingerprints" for texts (e.g., method bodies) and reduce the ramification of small textual changes like renaming and minor edits. This enables RefactoringCrawler to detect similar pairs of high-level code elements (methods, classes, and packages) between two versions of a project much more robustly than existing string matching techniques that are vulnerable to minor changes. To detect actual refactorings, it then refines the candidates by employing a more expensive and precise

semantic analysis based on reference graphs. To evaluate the performance of RefactoringCrawler, it was applied to three open-source Java projects and archived high values for both precision(95%) and recall(90%). However, similar to the work of Weißgerber and Diehl [57] the authors manually discovered the applied refactorings by inspecting their commit messages and release notes for computing recall, while they inspected the source code to compute precision. Later, Biegel et al. [61] replicated Weißgerber's approach using three different similarity metrics: CCFinder (text-based), JCCD [62] (ast-based), and Shingles (token-based). It was concluded that the three metrics performed with a comparable quality even though they can affect the ranking of refactoring candidates.

Xing and Stroulia [63] used both textual and structural similarity to detect refactorings between two versions of a system in their Eclipse plug-in named JDEVAN [64] [25]. JDEVAN initially constructs two UML logical design models from the source code corresponding to two versions of a Java system. Next, using UMLDiff [65], the two models are compared and the differences between them are reported as removal, addition, moving, and renaming of UML entities (e.g., class, package). Finally, JDEVAN's refactoring-detection module defines a suite of queries [63] assisted by a set of similarity metrics that attempt to categorize detected differences as refactoring instances through a hierarchical pairwise comparison between the two models' packages, classes, methods, and fields. As an example of an implemented query, an *Extract Operation* refactoring is inferred when the set of usage relations (read, write, call, instantiate) inside a newly added method proved to be a subset of the removed usage relations from the original method or their intersection set is greater than a user-specific threshold. JDEVAN found all the documented refactorings when applied to two systems and proved to be useful in detecting different types of refactorings in several studies. However, the authors confirmed that its rename and move refactorings detection is vulnerable to cases where there are not enough relations between the refactored entities and other parts of the program.

Ref-Finder [45] by Prete et al. [66] is capable of detecting 63 of Fowlers' catalog [67] 72 refactoring types, which contains the most comprehensive list of refactoring types to that time. It first describes classes, methods, and fields as a set of logic predicates along with their content (e.g., method body) and structural dependencies (i.e., field access, method calls, subtyping, and overriding) to represent the versions of a program as a database of logic facts. Additionally, supported refactorings are encoded as logic rules where the antecedent defines the constraints (i.e., change facts) and the consequent holds the refactoring type to be inferred. Next, it converts the antecedent of these logic rules as logic queries and then invokes them against the database of logic facts to identify program differences that match the constraints of each refactoring type under focus. Besides, by tracking the dependencies among refactoring types, lower-level refactorings were queried to identify higher-level, composite refactorings making Ref-Finder the first tool capable of detecting composite refactorings, where each refactoring consists of a set of atomic refactorings. For example, Extract Superclass refactoring is inferred by checking if a new superclass was created and a number of Pull Up Method/Field refactorings were identified that had moved fields and methods to the newly created class. For the detection of some types of refactoring, their rules require a special logic predicate that indicates if the word-level similarity between two candidate methods is above a threshold. This was implemented as a block-level clone detection technique that trims parenthesis and removes escape characters, returns keywords, and computes the world-level similarity between the two code fragments using the longest common sub-sequence algorithm. The tool was tested on three open-source Java programs and precision of 74% and recall of 96% were reported. However, later studies reported lower precision and recall for Ref-Finder [68] [69] [70]. Considering refactorings applied in isolation (root canal refactorings [13]) and ignoring refactorings with overlapping changes (i.e., floss refactorings [13]) was the reason behind higher precision and recall in the evaluation conducted by the authors.

Now we are going to discuss the current state-of-the-art commit-based refactoring detection tools RefDiff [28], RefactoringMiner 2.0 [27] and RefDetect [47], which are closely related to our work.

RefactoringMiner

Tsantalis et al. [71] proposed a tool based on an extended, lightweight variation of the UMLDiff [65] which is an algorithm for differencing object-oriented models. It is capable of identifying 14 high-level refactoring types: Rename Package/Class/Method, Move Class/Method/Field, Pull Up Method/Field, Push Down Method/-Field, Extract Method, Inline Method, and Extract Superclass/Interface. The process identifies refactorings between two models in several rounds. First, it compares the names or signatures of classes, methods, and fields in a top-down fashion and determines whether they have been matched, removed from the first model, or added to the second model. Next, removed elements are compared against the added elements by the equality of their names and parameter count to identify the changes in signatures of fields and methods. Third, the leftover removed/added classes are matched based on the similarity of signatures of members from the previous step thus this step can endure type changes. Finally, a set of refactoring detection rules defined by Biegel et al. [72] was extended and employed to infer actual refactoring instances. To evaluate their approach, the authors applied their technique in the version histories of three projects and reported 96.4% precision for Extract Method refactoring with 8 false positives and 97.6% precision for Rename Class refactoring with 4 false-positive instances. No false positives were found for the remaining refactorings. Later, Silva et al. [17] extended and re-introduced the tool as RefactoringMiner and used it to mine refactorings on large scale in git repositories. In their evaluation of the tool, a precision of 63% with 1,030 false positives out of 2,441 refactorings was reported. On the other hand, RefactoringMiner achieved precision and recall of 93% and 98% respectively when the authors evaluated it

as a benchmark on the dataset created by Chaparro et al. [73].

RefDiff 1.0

In their tool RefDiff, Silva and Valente [69] introduced the concept of analyzing only the changed, added, or deleted files between two versions of a program to detect refactorings. It is capable of detecting 13 high-level refactoring types through static analysis and code similarity comparison. As a first step, RefDiff represents the body of classes and methods as a multiset (or bag) of tokens, whereas for each field it considers tokens of all the statements that use that field. Next, to find similarities between code entities, a variation of the TF-IDF weighting scheme [74] is used to assign more weight to tokens that are less frequent, and thus have finer distinctive importance than other tokens. Additionally, the similarity threshold for different kinds of code elements is calibrated by using a set of ten commits from ten different open-source projects for which the project developers themselves have confirmed the applied refactorings [17]. Finally, similar to the evaluation performed by Prete et al. [66], they evaluated RefDiff based on an oracle of refactorings applied by graduate students in 20 open-source projects. The evaluation suggested that RefDiff surpassed RMiner [17], RefactoringCrawler [59] and Ref-Finder [66] in terms of performance and accuracy.

RefactoringMiner 1.0 /RMiner

Later, Tsantalis et al. [46] proposed a major evolution of their existing RefactoringMiner [17] [71] tool and renamed it to RMiner (RefactoringMiner version 1.0) which is the first refactoring detection tool that does not rely on code similarity thresholds. Similar to RefDiff, RMiner also processes only the changed, added, or deleted files of a commit; however, unlike its competitors such as, Ref-Finder, UMLDIFF, and RefactoringCrawler; RMiner

does not depend on building the two compared project versions. Consequently, this technique can support well commit-based refactoring detection, as the majority of the change history of software systems cannot be successfully compiled [75]. RMiner employs an AST-based statement matching algorithm and a set of detection rules to detect 15 representative refactoring types. It matches statements in a round-based fashion where textually identical statements are matched first. Then, the algorithm employs two novel techniques: abstraction, to facilitate the matching of statements having a different AST node type, and argumentation, which deals with changes in sub-expressions within statements due to the replacement of expression with method parameters, and vice-versa. To deal with overlapping refactorings (e.g., variable renames), while matching two statements, RMiner performs a syntax-aware replacement of the compatible AST nodes to make them identical. For evaluation, the authors created a dataset with 3,188 real refactorings instances from 185 open-source Java projects. Using this oracle, the authors reported a precision of 98% and recall of 87%, which was the best result at the time, surpassing RefDiff [69], the previous state-of-the-art, which achieved a precision of 75.7% and a recall of 85.8% on the same dataset. The superiority of RMiner is also confirmed by Tan and Bockisch [70] where it emerged as the winner among its competitors: RefactoringCrawler [59], Ref-Finder [66] and RefDiff [69].

RefDiff 2.0

In continuation to their previous work, Silva et al. [28] evolved RefDiff [69] to RefDiff 2.0 [1], which is the first multi-language refactoring detection tool. The tool is capable of detecting refactorings in Java, C, and JavaScript programs and remains the only known tool capable of detecting performed refactorings in JavaScript. It employs a two-phase approach where in the first phase source codes are represented as a Code Structure Tree (CST)

[1] https://github.com/aserg-ufmg/RefDiff

that abstracts away the detail of a particular language. Each node in CST is represented by higher-level entities, such as classes and functions. Since code can be written outside of a class in JavaScript and C, files are also considered as CST nodes. In the second phase, RefDiff 2.0, uses the same approach as its predecessor and determines the similarity between the CST nodes by tokenizing the body of CST nodes and then computing their weight using a variation of the TF-IDF weighting scheme. However, in contrast to its predecessor, RefDiff 2.0 uses a single default similarity threshold of 0.5 for all kinds of code element relationships. For Java projects, RefDiff 2.0, was evaluated on the same oracle [46] [2] that was used to evaluate RMiner [46] and precision of 96% and a recall of 80% were reported. On the other hand, for C projects, RefDiff 2.0 achieved a precision of 88% and a recall of 91% based on a small-scale experiment. For JavaScript, the computed precision was 91% using 87 refactorings, and recall was 88% using 65 refactoring instances. Later Brito and Valente [76] created a GO language plugin of RefDiff 2.0 named RefDiff4Go and reported similar precision (92%) and recall (80%) based on six GO projects.

RefactoringMiner 2.0

Recently, Tsantalis et al. extended RMiner [46] and introduced RefactoringMiner 2.0[3] [27]. The tool is capable of detecting over 80 different types of refactoring operations including low-level ones that occur within the method body (e.g., Inline/Extract/Split/Rename Variable) in Java projects. The main improvement is in the matching function, where new replacement types and heuristics are added. In the evaluation, the authors compare their tool with existing tools including its predecessor RefactoringMiner 1.0/ RMiner [46], and RefDiff 2.0 [28] by using a dataset containing 7,226 true instances for 40 different refactoring types, which are validated by experts. The results proved the superiority of the new version of RefactoringMiner 2.0 by achieving the best precision (99.6%) and recall (94%)

[2] http://refactoring.encs.concordia.ca/oracle
[3] https://github.com/tsantalis/RefactoringMiner

from the tools evaluated. Zarina Kurbatova from JetBrains-Research[4] leads and maintains kotlinRMiner, which is essentially an extension for RefactoringMiner 2.0 supporting Kotlin. Additionally, Python extensions are also created by Atwi et al. [77] (PyRef[5]) and Dilhara et al. (Py-RefactoringMiner[6]).

RefDetect

In contrast to RefDiff 2.0 [28] which uses token similarity and RefactoringMiner 2.0 [27] which structurally matches the code fragments, RefDetect by Moghadam et al. [47] employs a completely different approach where the whole program is represented as a sequence of characters that abstracts away the specifics of that particular language. Their approach detects refactorings in three steps. First, each entity is represented by 7 different types of characters in a specific order: class (C), interface (I), generalization/inheritance relationship (G), attribute (A), method (M), method parameter (P), and a property access/-coupling relationship between two classes as (R). As an example, if a class B has two methods, inherits class A, and accesses a field of another class C, B is represented as CGMMR. These entities representing sub-strings are sorted by the original name of the corresponding entity and form a single string that represents one version of the input program. Second, a sequence alignment algorithm (FOGSAA [78]) is used to identify the changes existing between the two input program versions. For each pair of characters in the input strings, the alignment algorithm considers three possibilities: match, mismatch, or gap and returns the initial list of unmatched entities. In the third and the final step, a threshold-based refactoring detection algorithm is used to identify the set of applied refactorings that resulted in the evolution from the older version to the newer one. RefDetect was evaluated and compared with the current state-of-the-art refactoring detection tool RefactoringMiner 2.0

[4]https://github.com/JetBrains-Research/kotlinRMiner
[5]https://github.com/PyRef/PyRef
[6]https://github.com/maldil/RefactoringMiner

on the same aforementioned publicly available refactoring oracle. The authors found that while RefactoringMiner 2.0 clearly outperforms RefDetect in terms of precision (98.5% vs 91.2%), RefDetect achieved a better recall (84.5% vs 78.9%).

2.2 Limitations of Existing Approaches

We will now discuss some of the key limitations of existing approaches that we tried to overcome in our proposed approach.

Dependence on similarity thresholds: Most of the refactoring detection tools use similarity metrics to compute the resemblance between a pair of code elements originating from two different snapshots of a software system. These metrics typically require user-calibrated thresholds to determine whether the elements should be considered matched. Since developers often perform other maintenance activities (e.g. bug fixes, performance) during refactoring [13], similarity thresholds often help endure such overlapping changes. Modern refactoring detection tools come pre-configured with default thresholds that are empirically derived through experimentation on a relatively small number of project (one for UMLDiff, three for Ref-Finder, and RefactoringCrawler, and ten for RefDiff 2.0). Therefore such thresholds risk being over-fitted to the test projects, and thus cannot be general enough to handle all the possible ways refactorings are applied in projects from different domains. Consequently, it may require a manual inspection of the reported refactorings against the source code to find false positives in order to re-calculate the thresholds.

Several studies in the field of software measurement and metric-based code smell detection, extensively investigated the problem of deriving the holy grail value for a particular threshold by applying various statistical methods and machine learning techniques on a large number of software projects [79], [80], [81], [82], [83]. Dig et al. [84] reported that precision and recall can vary significantly for the same software system based on different values of thresholds. Moreover, Aniche et al. [85] have shown that different threshold

values are required for source code metrics for software systems using different architectural styles and frameworks. This is especially true for JavaScript programs, where unlike the object-oriented style of Java, developers can opt for either functional or object-oriented fashion or a mix of both, and thus the threshold yielding better results in object-oriented programming may need adjustment for projects written in a functional approach. Therefore, based on experience, it can be concluded that it is very difficult to derive universal threshold values that can work well for all projects, regardless of their architecture, domain, and development practices [27].

Dependence on built project versions: Tools like RefactoringCrawler, Ref-Finder, and UMLDiff require building the project versions in order to retrieve structural information, such as field access, method call, subtyping and overriding. To accurately obtain such information in Java projects, a compiler such as the Eclipse JDT plugin is used to resolve type, variable, and method binding information. However, this approach is severely limited by the fact that most commits are not compilable [75]. As a result, these tools may not be suitable for performing large-scale refactoring detection in the entire commit history of a project.

Incomplete oracle (ground truth): Calculating the true recall for refactoring detection studies has always been very challenging and debatable, since it is hard to identify all the refactorings that have been performed in a commit. Inspecting commit messages has been proved to be an unreliable indicator of presence refactorings as developers do not always mention them in the commit message [13]. Moreover, Moreno et al. [86] found out that only 21% of the release notes include information about refactoring operations by manually inspecting 990 release notes from 55 open-source projects. Last but not the least, analyzing each file manually for identifying performed refactorings is time-intensive, requires expertise, and does not guarantee finding all refactoring instances.

Programming language specificity: The majority of the existing approaches support

only Java systems. In fact, to the best of our knowledge, no other tool can detect refactorings in the JavaScript ecosystem except for RefDiff 2.0. Even so, because of the token based nature of its detection methodology, RefDiff 2.0 cannot detect low-level refactorings, such as renaming variables, renaming parameters. Lastly, the token-based approach cannot be successfully used by source code diff tools to cross-match code elements inside containers (e.g. Functions, Classes) as this approach cannot match the corresponding token in the next version.

Chapter 3

Approach

In this chapter, we present our approach for detecting refactorings between any two JavaScript source code versions. Before going into the details of our implementation, we illustrate some of the key differences in programs written in Java vs. JavaScript.

3.1 Java vs. JavaScript Program Structure

3.1.1 Function Expression

Function expressions are similar to Java lambda and can be assigned to a variable, invoked, passed as arguments, and can be declared almost anywhere in the program. Unlike Java

```
1   const add = (function () {
2       let count = 0;
3       return function () { count += 1; return count; }
4   })();
5
6   add();
```

Figure 1: A self-invoking function expression is assigned to the variable `add`.

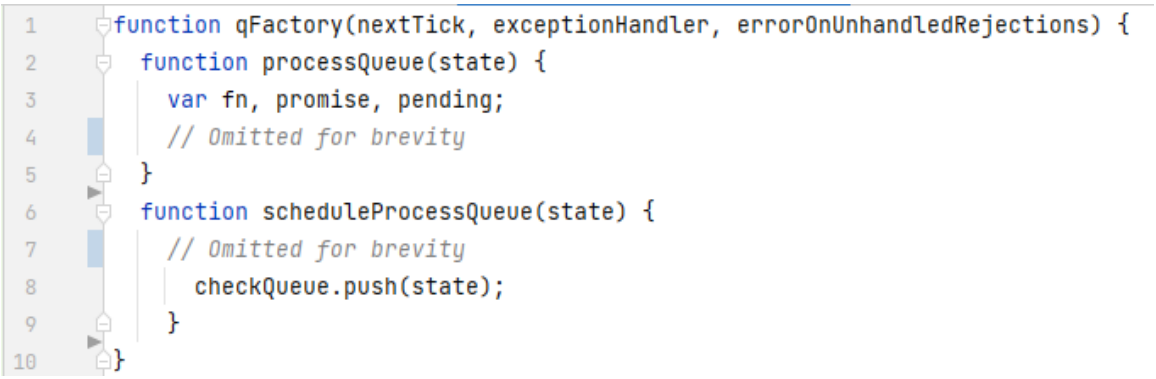

Figure 2: Function `processQueue` and `scheduleProcessQueue` are directly declared inside of the function `qFactory`.

lambdas, which are typically short in size, in JavaScript, we have found instances of function expression which cover the whole program and often tend to be very large.

Figure 1 shows an example of a self-invoking function expression that is invoked immediately after it's declared. This basically creates a private scope for the variable `count`. The returned function in line 3 by the function expression is assigned to the variable `add` which is invoked in line 6.

Being a functional language, JavaScript programs typically make heavy usage of function declaration and function expression. This contributes to the creation of syntactically very large statements. Structurally matching large statements without breaking down large expressions/statements can be a challenging task.

3.1.2 Nested Function Declaration

In JavaScript, functions or classes can be declared almost anywhere and we have found it is a common practice to declare functions inside of a function directly. This is different than Java where we cannot declare a named method directly inside of another method. In Java, methods can only be declared directly under a class.

In figure 2, the topmost function `qFactory` in line 1 contains only two functions but no other statements. Therefore, if `qFactory` is renamed in the next version, its

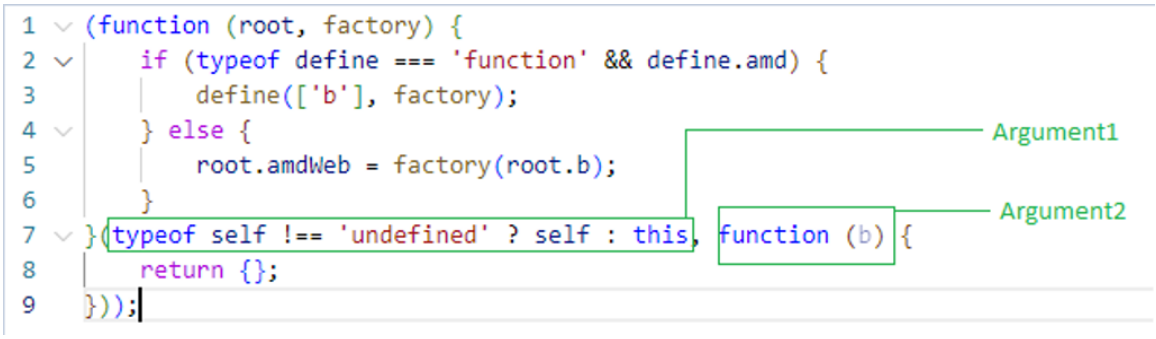

Figure 3: A pattern of Universal Module Definition in JavaScript where the return statement on line 8 can contain a program that can be run on the client or server-side.

child functions `processQueue` and `scheduleProcessQueue` can be used to identify their parent `qFactory`. By using functional expressions or nested function declarations, JavaScript programs often becomes highly nested structurally. Whereas in java, the nested depth is usually a class and its methods and sometimes methods with anonymous classes.

3.1.3 Script

In JavaScript, code can be directly written inside of a file. These files are typically executed as scripts. Figure 3 shows a frequent JavaScript pattern which is typically written inside of a file. In Java, however, no code can be written outside of a class or method.

Such structure is significant for a diff tool because, in Java, the classes must be declared inside of a package that is explicitly written (except for the default package) therefore combining the file path, class name, and method information can be used to estimate the location of a particular statement. However, since statements can be written directly inside of a file in JavaScript, if they are moved between files, we cannot presume such information. Moreover, without the containing classes/functions, the combination of statements pair to be matched among different files can be very high.

```
1   var handle = function(req, res, done) {
2       var self = this;
3   };
4
5   handle(req, res);
```

Figure 4: A function expression with 3 parameters is assigned to a variable `handle`. The variable `handle` is invoked like a regular function in line 5.

3.1.4 Functions as Variables

Function can be used as variable in JavaScript and can be passed as the argument of a return expression. Figure 4 shows how the variable `handle` is invoked after it was initialized.

At first glance, this may seem like a Java lambda expression. However, in Java the lambda expressions need a named method to invoke which is typically declared inside of an interface. On the other hand, during invocation, the first parenthesis is directly written after the variable `handle` at line 5.

This increases challenges for a structural matching tool to match an invocation with its declaration during Extract Function refactoring where a code is extracted to a new function and the original function invokes the newly added function in lieu of the removed code.

3.1.5 No Static Types

In JavaScript, there are no strong types such as `long`, `int` etc. like in Java. Therefore variables are declared without any static types. This makes it potentially difficult to identify renamed variables between versions because there is no hint of the type of the variable. Furthermore, the absence of types negatively affects the matching of parameters of functions and therefore their signatures.

Difference	Example
Semicolon is optional to end Statement	`let x = 1`
Returned Function Invocation	`config.queue.shift()();`
Self-Invoking Function	`(function (a, b) {})(10, 20);`
Default Value in Parameter	`(function (a, b = 20) {})(10);`
Destructuring Assignment	`[v1, v2, ...rest] = [1, 2, 30, 40]`

Object Expression	`const rectangle = {`
	`  w: 20, h: 30`
	`};`

Object Destructuring	`const { h } = rectangle;`
Different Types of Array Elements	`let array = [10, "John", function () {}];` `array[2]();`

Figure 5: Some of the supported features of JavaScript which are not available to Java.

3.1.6 Other Differences

Figure 5 illustrates some of the other differences in syntax between Java and JavaScript code. We have found that in many projects developers do not put semicolons at the end of a JavaScript statement. Moreover, since functions can be assigned to a variable, in JavaScript they are often returned from a function and then invoked immediately after the call. The optional default value of a parameter often contains a function or empty object literals to be used inside of the function body in case the caller does not provide an argument to that parameter during invocation. Object expressions are similar to class declarations in java however, the values of properties are assigned using a colon (:) instead of an equal sign (=). The destructirng pattern basically selects specified elements or properties from an array or object and can assign them to variables in a single expression. Lastly, the array elements can be of any type thus sometimes a function element can be obtained to immediately invoke from an array.

3.2 Detection Methodology

Similar to RefactoringMiner, JsDiffer can take a commit as input. A commit contains the changed, added, or removed files between the parent and child revisions of the source code of a project. Additionally, JsDiffer can take any two JavaScript source directories or code snippets as input and returns the detected refactorings between the two sources. The whole process can be divided into 4 general steps: 1) Source Code Extraction 2) Modeling Code Elements, 3) Matching Code Elements, and 4) Applying Refactoring Heuristics.

3.2.1 Source Code Extraction

Extracting the source code efficiently and then representing it by a useful source model is itself often a challenging task. Supersets of JavaScript such as TypeScript, JSXElements can be transpiled[1] into JavaScript code and are often found intermixed in vanilla JavaScript source files, which further complicates the task of parsing.

We have chosen Babel[2] as our source code parser. Babel is the most popular JavaScript compiler/transpiler that can handle any modern JavaScript syntaxes (e.g., class declaration). Babel library is written in JavaScript therefore we needed a way to execute Babel in a JavaScript environment (i.e., NodeJs[3]).

Since RefactoringMiner is written in Java, for potential reusability, we chose to develop JsDiffer in Java as well. Therefore we have used J2V8[4] - a JavaScript engine /executor for Java. We had chosen J2V8 because it is one of the most efficient ones and has been successfully used by RefDiff 2.0[28]. Moreover, we have also tried Google Closure Compiler[5] which is written in Java and thus much faster than executing Babel in a virtual environment

[1]Transpilation: Conversion of source code written in one language to another language
[2]https://babeljs.io/
[3]nodejs.org
[4]https://github.com/eclipsesource/J2V8
[5]https://github.com/google/closure-compiler

from a Java project. Unfortunately, it cannot correctly parse JavaScript files that have intermixed supersets of JavaScript or modern JavaScript syntaxes such as arrow functions and class declarations. Therefore, we opted for Babel as our chosen JavaScript parser.

For detecting refactoring between two directories, we consider all the JavaScript files of each revision. On the other hand, for git repository-based comparison, we only process the added, modified, or removed files in the specific input commit. Analyzing only the changed files instead of the whole project improves accuracy and performance, since it reduces the number of program statement comparisons to be performed. In both cases, we only consider files with .js extension to keep only vanilla JavaScript files. In practice though, these files may contain TypeScript or even HTML-based source code, and using Babel, JsDiffer processes only the portion of the code that contains pure JavaScript code.

This phase finishes by returning a list of JavaScript source files along with their contents that will be processed in the subsequent steps.

3.2.2 Source Model Creation

In this phase, JsDiffer parses source code files and returns two corresponding Java source code models which represent the first and second versions of the input.

Each program model contains a list of SourceFile models representing the input JavaScript files from the first step. For each source file, we extract the following information by traversing its Abstract Syntex Tree (AST) using Babel.

- **Function:** In JavaScript, a function can be either a declaration or an expression (when used in a statement). The function expressions are treated in a similar way as RefactoringMiner models anonymous class declarations in Java code. In addition to the name, we extract the name of the parameters and the statements inside the body of a function.

- **Class:** Class declarations are a syntactic sugar of the legacy JavaScript functional constructor. It contains field declarations and methods. In addition, it can have an inheritance relationship to another class. Class declarations can be also used in a statement, in which case it is considered as a class expression similar to the Anonymous Class declaration of Java-based Refactoring Miner.

- **Object:** In JavaScript, object expression or declarations are structurally very similar to class expressions or declarations and therefore treated as such.

```
var product =  {
    id: "xxx",
    price: 10
};
```

Listing 3.1: Object expression assigned to a variable

- **Single Statement:** Single Statements or Leaf Statements contain no nested statements (e.g. let x = 1;). We extract and categorize all the elements such as variable names, literals, array accesses, function invocations, and anonymous classes/function expressions that appear in a single statement. Moreover, we record additional metadata, such as its parent method (if any), its original AST in a text representation, as well as its depth and its index in the AST.

- **Block Statement:** A Block statement is a composite statement that contains the same attributes as a single statement. However, a Block Statement may contain a body that consists of other statements. For example, `try`, `for`, `if` are all Block Statements. In addition, in JavaScript, a BlockStatement may contain function or class declarations.

In JsDiffer, we introduced an additional layer of abstraction for functions, classes, and objects named *Container* which keeps a reference to its parent container (if any) and its *statements*, and *declarations*. Because, unlike Java, in JavaScript, a function declaration

can contain other function declarations, class declarations, object declarations, and vice versa. Besides, in JavaScript, these Containers can be assigned and used as variables. Note that RefactoringMiner does not consider these two frequent patterns of JavaScript programs, and thus JsDiffer has limited support for refactorings performed inside nested function declarations.

```
1  var math;
2  math.sum = function (a, b) {
3      return a + b;
4  };
5  function main() {
6      const result = math.sum(10, 20);
7      print(result);
8      function print(text) {
9          console.log(text);
10     }
11 }
12
13 class AtomBackend {
14     const buffer = editor.getBuffer()
15     stop() {
16         this.subs && this.subs.dispose()
17     }
18 }
```

Listing 3.2: Example JavaScript Code Snippet

The example code snippet shown in Listing 3.2 is modeled as a *File Container* with the following properties:

1. List of Function Declarations **FDs** = {main}
2. List of Single Statements **LSs** = {var math, math.sum = function...}

3. List of Block Statements **BSs** = {}

4. List of Function Expressions **FEs** = {}

5. List of Class Declarations **CDs** = {AtomBackend}

Here, in line 2, even though the "sum" is a function variable of "math", by RefactoringMiner's concept, it is an anonymous class/function declaration as it is part of a leaf statement. However, for JsDiffer, if a function is assigned to a variable and the variable has a simple one-word identifier (e.g. sum), it is treated as a function declaration.

In line 13, "AtomBackend" is modeled as a class declaration with a function declaration named "stop". Line 14 is modeled as an attribute of this class where the name of the attribute is "buffer". Note that since RefactoringMiner does not take into account that statements can be written directly inside a class. Therefore, JsDiffer supports variable declaration statements inside a class that can be converted to an attribute inside a class.

Contents inside a function declaration are modeled identically to RefactoringMiner except we also support direct declaration of a function. Therefore, the function "main" declared in line 5 has the following properties:

1. List of Single Statements **LSs** = {const result =..., print(result);}

2. List of Function Declarations **FDs** = {print(text)}

By introducing this new property to the function declaration model, we supported JavaScripts' potential infinite nesting depth of Function or class declarations. Note that this is in contrast to Java where a function cannot be declared outside of a class.

Since both functions and file containers are identical in source code structure for JavaScript, we were able to abstract them into a general container. However, to re-use RefactoringMiner's code and improve the accuracy of the detection process, we had to keep class declaration as a different container to file and function declarations.

For each leaf statement, we keep track of all the tokens such as the identifiers, function calls, object creations, literals, operators, etc. They are used to compare the differences between two leaf statements in the matching phase.

3.2.3 Matching Code Elements

This phase takes the two program models from the previous step and identifies the matched and unmatched elements between the two versions. Each program model consists of a list of source file models, which contain various code elements.

Source File Models Matching: Between the versions, source files can either be moved to another folder, renamed, moved and renamed at the same time, or remain with the same name and file path.

Algorithm 1: Source File Models Matching

Input : Two Lists of Source File Models S_1 and S_2.
Output: Set S_m of matched source file models, Sets of unmatched source file models S_{u1}, S_{u2} from S_1 and S_2, respectively and detected refactorings R

1 $S_m \leftarrow identicalFilepath(S_1, S_2)$
2 $S_{u1} \leftarrow S_1 \backslash S_m$
3 $S_{u2} \leftarrow S_2 \backslash S_m$
4 $checkForMovedFiles(S_{u1}, S_{u2}, EXACT\_MATCH)$
5 $checkForRenamedFiles(S_{u1}, S_{u2}, EXACT\_MATCH)$
6 $findRefactoringsInCommonNamedFiles(S_m)$
7 $checkForMovedFiles(S_{u1}, S_{u2}, RELAXED\_MATCH)$
8 $checkForRenamedFiles(S_{u1}, S_{u2}, RELAXED\_MATCH)$
9 $findRefactoringsInUnmachedFiles(S_m)$

Algorithm 1 shows the high-level overview of our approach. In line 1, it first tries to match each source file model of version one with source file models from version two by their qualified name (i.e., directory path and filename without extension). If the corresponding file is found in the second version, they are added to a list of common files. If they are not found, it is added to a list of removed files. Inversely, if a file from the second version cannot be matched, it is added to a list of added files. Thus, this step returns lists

of common and unmatched files.

Next, it tries to match the moved files with equal filenames but in different directories. It also utilizes any rename file hint available by the input commit. The algorithm tries to exactly match the name of classes and functions inside the pair of candidate files up to a nesting depth of 3. Then, a similar approach is used for Rename File refactoring detection where the file name is different in the same directory.

Lastly, for each pair of common source files, the declared container elements and directly written statements (i.e., scripts) are matched. Note that this is also something that has to be improvised as Refactoring Miner does not support statements written outside of a method body and its matching heuristics significantly rely on the information of the containing method.

Container Matching: Initially, the container elements are matched in a round-based fashion. In the first round, the class and function declarations are considered for matching by their qualified name (i.e., parent container name and class/function declaration name). Similar to the previous step, they are added to the initial lists of matched, removed, or added containers based on the comparison of their names. Note that functions are matched with functions only and classes are matched with classes only. Since Class Expressions and Function Expressions are part of a single statement or expression they are matched during the statement matching process.

For each matched container element (i.e., function or class declaration), we match their code elements, such as statements or other container declarations and container expressions.

The body of each container element can contain Leaf Statements, Block Statements, class or function declarations, or even function or class expressions. Thus, we consider all of these to determine if two functions are equivalent or not.

Leaf Matching: Leaf Statement and expressions consist the backbone of our algorithm

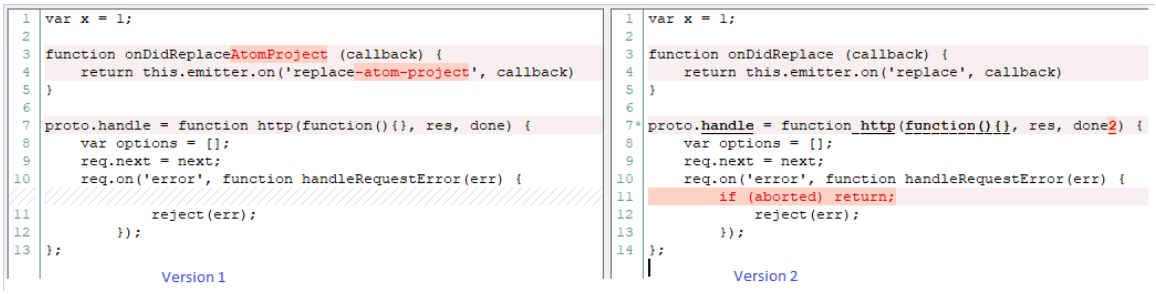

Figure 6: An example of leaf statement (line 7) with multiple functional expression.

as they help match other elements such as block statements or the body of a container. We followed the same technique described in RefactoringMiner 2.0 [27]. We encourage curious readers to find in-depth information in that paper.

A notable difference from RefactoringMiner is that due to the Levenshtein Distance computation cost, classes and functions are matched differently when they are part of a statement. For example, RefactoringMiner applies various string replacements on the text of a single statement containing anonymous classes. In Java, text representing the anonymous classes tends not to be huge and the Levenshtein Distance calculation can be performed without any significant performance cost for the majority of the cases. However, we have found that function/class expressions tend to be very long in JavaScript and sometimes contain the whole program therefore we employ a round-based technique to pair container expressions in a pair of leaves. This significantly reduces the accuracy of JsDiffer, as RefactormingMiner could potentially find exact matches when comparing two anonymous classes by using the statements inside their declared methods for matching the original pair of leaf statements.

In addition to exact matching, a set of heuristics is finally used to determine if two leaf statements can be matched if identical matching fails.

Figure 6 shows an example of two versions of a code snippet. The "`proto.handle`..." is considered a leaf statement in RefactoringMiner's concept as it is not a straightforward function declaration which is a method of a class. Moreover, this statement contains two

functional expressions. As we cannot identically match them by simple textual comparison because of the added statement in line 11 of version 2 and the renamed parameter "done2", RefactoringMiner would perform a Levenshtein Distance calculation by replacing "done" with "done2". This is not suitable for JavaScript as we have found frequent cases where the size of code related to the leaf statement is very large. Thus in some cases, it took 3 hours to match just one pair of statements.

Therefore, in JsDiffer, to match the statement pairs, we employed a round-based technique. First, if any of the statements contain functional expressions, we do not perform any string replacement and thus do not calculate the expensive edit distance between the text of the two leaf statements. For this example, we first check if there is a name for the functional expressions provided. In this case, both statements have a functional expression with the identical name "http". Therefore we go inside the body of "http" functional expression to match the statements inside.

However, a name for a function expression is not always available therefore in the second round, we try to match functional expressions which have common child function declarations inside of them. This increases the confidence in matching two functional expressions as JavaScript is a popular functional programming language, thus if a pair of functions contain common child function declarations, they are likely to be the same function.

In the third round, we check for functional expressions with the same-named parameters. Thus, having step-by-step matching helps JsDiffer relax the matching rule gradually to allow the detection of complex leaf statements that are textually different.

Expression Matching: The condition of an `if` statement or `switch` statement, `for` loop, `catch block`, etc. is considered as an expression in JsDiffer. An expression is handled in a very similar fashion to a leaf statement and holds the same information, such as variables, and literal and container expressions that appear inside of an expression.

The expressions from two pairs of composite statements help JsDiffer to determine whether they are considered matched or unmatched.

Composite Matching: Since a composite statement contains an expression, statements, and declarations, we use all of them to determine whether a pair of candidate composite statements can be considered as matched. At first, like RefactoringMiner, JsDiffer tries to match them by expression and statements. In case the two pairs of composites cannot be matched, we try to match them by matching their child function declarations. This is an additional step over RefactoringMiner, which does not support the concept of having declarations inside of a composite statement.

This step essentially generates the diff between pairs of containers from which the applied refactorings can be determined.

3.2.4 Applying Refactoring Heuristics

This step is applied in every container diff elements to determine applied refactorings.

Renamed Containers Detection: This is determined by observing the changes in the signatures of the container diff. For example, if two function expressions have been matched but there is a difference in the names of the functions, JsDiffer adds a Rename Function refactoring to the global list of detected refactorings.

Variable Related Refactorings Detection: Rename Variable, Rename Parameter, etc. refactorings are supported by JsDiffer. These are determined during container matching by observing the difference between their matched leaf statements and expression pairs. For example, if two leaf statements are matched identically by replacing one of the variables of version 1 with an added variable in version 2, chances are those two are the same but renamed variables.

We were able to re-use a significant portion of variable replacement techniques used by RefactoringMiner for this portion. This process applies several consistency checks to make

sure that the detected rename instance is a valid refactoring. For example, if a variable is renamed, all the references of that variable are also renamed in the same commit. We check for this consistency to make sure that the detected refactoring is correct. On the other hand, multiple variables declared in the same container are harder for JsDiffer to match. Since, in JavaScript, there are no named strong types (e.g. string, int, etc.), we used variable kinds (e.g let, var and const) for identifying the correct variable pairs. However, we have found that this is significantly under performing because of not having strong types as the 3 variable kinds of JavaScript are often interchangeable.

Extract/Inline Function Detection: JsDiffer first tries to employ the same techniques described in the RefactoringMiner paper to determine whether a method has been extracted or inlined. It tries to check if some removed statements from a function can be matched with the statements inside of an unmatched newly added function and a call from the original function is made to the newly added function. Conversely, for inlined function refactorings, we check for removed functions where the call to the function is also removed and replaced with the statements that were inside of the removed function.

Moved Containers Detection: Moved Source Files are detected in two rounds. In the first-round, the contents of two files with the same name but in a different directory are identically matched to determine if they are indeed the same file but in a different directory. This is referred to as exact or strict matching by JsDiffer. After matching the common named files, one additional round of relaxed move detection rules is used to determine if an unmatched file is actually moved. This detection phase is more relaxed than the previous stricter exact matching rule, because in this phase if the number of matched function declarations is more than half of the total number of function declarations, we consider the source files as matched i.e., moved. Because of this multi-phase matching, containers with more similar contents get matched first. This reduces the chance of incorrectly matching containers in further steps.

Chapter 4

Evaluation

Evaluating the accuracy of a refactoring detection tool is often challenging due to the lack of a proper dataset. Developers often do not mention the applied refactorings, and thus it is not possible to find and confirm all the refactorings performed in a commit. Moreover, the validity of some refactoring instances is subjective in nature, and depending on the validator, one particular instance can be marked as valid or invalid.

To evaluate the accuracy of our approach, we ran our tool on 608 commits which potentially include refactoring instances. To the best of our knowledge, RefDiff 2.0 [28] is currently the only other tool capable of detecting refactorings performed in JavaScript projects. Therefore, we evaluated JsDiffer's precision and recall by comparing it with RefDiff 2.0 on an oracle of 341 JavaScript refactorings.

In our evaluation we investigate the following research questions:

RQ1. What is the accuracy of JsDiffer in refactoring detection and how does it compare to that of RefDiff 2.0?

RQ2. What is the accuracy of JsDiffer in variable-related refactoring detection?

RQ3. How does the execution time of JsDiffer compare to that of RefDiff 2.0?

4.1 Oracle Creation

4.1.1 Dataset Creation

In their paper, Silva et al. [28] mentioned a dataset consisting of 3,481 reported JavaScript refactoring instances out of which 87 instances were manually validated to calculate the precision and recall of their tool.

Interestingly, we discovered that not all the refactorings that were found in the dataset provided in the paper were reported when executing the latest version of RefDiff[1]. Additionally, we have discovered a different file containing 4,365 refactoring instances in their evaluation project on GitHub. Moreover, Silva et al. also provided 65 instances of documented refactorings based on the commit message. Combining all these sources, we ended up with a total of 608 unique commits each containing at least one reported refactoring across 19 unique projects.

The inconsistency in the reported refactorings in the paper and those actually detected by RefDiff 2.0 could result in a noisy or biased precision and recall. Therefore, we decided to run the latest versions of both tools. RefDiff 2.0 and JsDiffer reported 3708 and 2365 refactorings, respectively, from these 608 commits which constitute our dataset.

4.1.2 Commit Selection for Oracle

Due to the difficult nature and time-consuming manual process of validation, it was not feasible to validate all refactorings from the dataset to create an oracle. One way to calculate the precision and recall of both tools would be to validate a specific number of random refactoring instances. This approach was taken by RefDiff 2.0 when evaluating its accuracy on JavaScript projects by taking 10 random refactoring instances for each of the different refactoring types.

[1] https://github.com/aserg-ufmg/RefDiff/commit/889b0bf

Repository	Description
react	A declarative, efficient, and flexible JavaScript library for building user interfaces.
vue	Vue.js is a progressive, incrementally-adoptable JavaScript framework for building UI on the web.
d3	Bring data to life with SVG, Canvas and HTML.
react-native	A framework for building native apps with React.
angular.js	AngularJS - HTML enhanced for web apps.
create-react-app	Set up a modern web app by running one command.
jquery	A fast, small, and feature-rich JavaScript library.
atom	The hackable text editor.
axios	Promise based HTTP client for the browser and node.js.
three.js	JavaScript 3D library.
socket.io	Realtime application framework (Node.JS server).
redux	Predictable state container for JavaScript apps.
webpack	A bundler for javascript and friends.
Semantic-UI	Semantic is a UI component framework based around useful principles from natural language.
reveal.js	The HTML Presentation Framework.
meteor	Meteor, the JavaScript App Platform.
express	Fast, unopinionated, minimalist web framework for node.
material-ui	React components for faster and easier web development.
Chart.js	Simple HTML5 Charts using the canvas tag.

Figure 7: Open-source JavaScript projects used for the evaluation.

However, we opted for validating all instances in a commit versus validating random refactoring instances, mainly because of the increased chance of discovering other refactorings undetected by one of the tools. This potentially helps us to calculate recall more accurately. Furthermore, validating all refactorings in a commit could provide a better estimation of accuracy because ultimately both tools detect refactorings between two source codes that could be represented by a commit. Therefore, we felt this approach for validating precision and recall is better than validating random instances. It's also important to note that, all the commits come from modern popular JavaScript library projects such as react js, angular, etc. Figure 7 provides short descriptions of the projects that were selected for evaluation. These were already carefully curated and sorted by RefDiff 2.0 paper [17] based on popularity, duration, etc. More details of the selection criteria for these projects can be found in their paper.

To create an oracle from the dataset, we selected 5 random commits from each of the projects which fulfill the following criteria:

First, the commit must contain at least one common refactoring type that was reported

by either of the tools. This helps us compare both tools as we don't have to validate a commit that does not contain a commonly supported refactoring instance. Second, the number of reported refactorings that is supported by both tools in that commit should not exceed more than 10, because JsDiffer and RefDiff 2.0 reported more than 10 refactorings in only 29 and 44 commits respectively out of these 5 commits. Therefore we felt confident that choosing these commits represents a more general population. For example, in one of the commits[2], JsDiffer and RefDiff 2.0 reported 408 and 362 refactorings, respectively. Both tools reported 362 Move File refactoring in this commit which is a significant portion of their total reported refactorings. From Table 1 it can be seen that this huge number of refactorings in a single commit can be considered an anomaly and thus could adversely affect the practical precision or recall.

Table 1: Distribution of reported refactoring instances among commits when both tools were applied in 608 commits.

	# Commits	
# Refactorings	JsDiffer	RefDiff 2.0
1	118	218
2	44	95
3	15	34
4	11	24
5	4	15
06-10	18	38
11-20	10	15
more than 20	19	28
Total Commits	239	467

Furthermore, a large number of refactoring instances in a commit not only makes it difficult to understand the source code during manual inspection, but it also increases the chance of making errors while validating refactorings. Last but not the least, having a smaller number of refactorings in a single commit makes it easier for the evaluator to fully validate a commit while increasing the chance of noticing refactorings missed by

[2]https://github.com/mui/material-ui/commit/3e4854b39b17511d9b312071ca93061eeb503f5d

the evaluated tools.

4.1.3 Validation of Refactorings

To validate the refactorings, we employed a semi-automated approach which involves identifying commonly reported refactorings and manual inspection of source code.

First, the reported instances from both JsDiffer and RefDiff 2.0 were stored in a common format. This helped us to compare and identify refactorings reported by both tools. If a refactoring is reported by both tools, in many cases we were able to automatically mark them as a true positive instance. For the rest, the author of this thesis manually inspected the commit, and each reported refactoring was marked as true positive, false positive, true negative, or false negative for JsDiffer and RefDiff 2.0, respectively. If one of the tools reports a false positive instance and the other tool does not report it, we put a true negative for the other tool, but keep the false positive for the first tool. Moreover, in case of the same refactorings reported by both tools, we keep only one refactoring instance in our oracle as it is considered a duplicate.

In case of difficult or complex refactorings, either the author asks for validation from the thesis supervisor who has much more expertise in validating refactorings or keeps them as invalidated. This way we feel confident that the oracle we have created is quite accurate. In addition to validating reported refactorings, during the inspection, if a refactoring is discovered which is not reported anywhere, it is also added to the oracle as a false negative for both JsDiffer and RefDiff 2.0.

Internal_Move_Function and Internal_Move_Rename_Function are only supported by RefDiff 2.0 which represents the change of the parent container of a function. It also reports internal move functions for cases where a function is moved to another block statement i.e., moved to another pair of curly braces, and JsDiffer does not report such refactorings. Moreover, sometimes all the child functions are reported as individual internal moves by

RefDiff 2.0 when the parent function is actually moved to another container and JsDiffer reports only the parent container move. In such cases, manual validation is necessary to consider multiple internal moves as one single Move Function refactoring. On the other hand, we did validate refactorings that are not supported by RefDiff 2.0 to find the precision of JsDiffer as an individual tool.

4.2 Result and Discussion

In this section, we will compare our tool with the current state-of-the-art RefDiff 2.0 based on their precision and recall of our carefully constructed Oracle. We will also compare their execution time and finally discuss cases where JsDiffer outperforms RefDiff 2.0 and vice versa.

4.2.1 RQ1: Refactoring Detection Accuracy

Table 2 shows the precision and recall for each refactoring type for RefDiff 2.0 and JsDiffer, respectively.

Table 2: Precision and Recall per commonly supported refactoring types. Hyphenated (-) cells represent cases where there was not enough data to calculate the precision or recall.

SL#	RefactoringType	#Validated	JsDiffer Precision	RefDiff 2.0 Precision	JsDiffer Recall	RefDiff 2.0 Recall
1	EXTRACT_FUNCTION	65	1	0.93	0.43	0.82
2	EXTRACT_MOVE_FUNCTION	14	-	0.83	-	0.38
3	INLINE_FUNCTION	10	1	1	0.14	0.71
4	MOVE_CLASS	1	1	1	1	1
5	MOVE_FILE	36	1	1	1	1
6	MOVE_FUNCTION	42	1	0.97	0.3	0.94
7	MOVE_RENAME_FILE	10	1	1	0.5	1
8	MOVE_RENAME_FUNCTION	13	-	1	-	1
9	RENAME_CLASS	4	0.5	1	0.33	1
10	RENAME_FILE	17	0.92	0.94	0.73	1
11	RENAME_FUNCTION	67	0.96	0.92	0.39	0.92
	Total	**279**	**0.97**	**0.95**	**0.45**	**0.89**

For each tool, we calculated precision and recall for each category of refactorings based

on the following formulas:

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

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

Where **TP** = Number of True Positives and **FP** = Number of False Positives and **FN** = Number of False Negatives.

The total precision and recall are also calculated by using the total number of true positives, false positives, and true negative instances. Table 3 shows the overall precision and recall as well as precision and recall for only the common refactorings of both types. Note that this does not include 73 instances of Rename Variable refactorings which are discussed separately.

Table 3: Oracle precision and recall for 341 validated instances overall and out of 279 instances were a refactoring type supported by both tools (Common Types).

Tool	Precision (Overall)	Precision (Common Types)	Recall (Overall)	Recall (Common Types)
JsDiffer	0.96	0.97	0.44	0.45
RefDiff 2.0	0.86	0.95	0.9	0.89

The result shows that RefDiff 2.0 has significantly outperformed JsDiffer in terms of recall. JsDiffer seems to have better precision in every category except for Rename Class refactoring. JsDiffer obtained a very poor recall on Inline Function and Move Function refactorings. We inspected 37 false negative instances of JsDiffer to categorize the root cause of missing them and present them in Figure 8.

We can generalize the moving or extracting outside of a container categories into a single one and can say that it is the most prevalent reason for the missing refactorings. This happens when a piece of code or a function is moved from a file to another file or in the

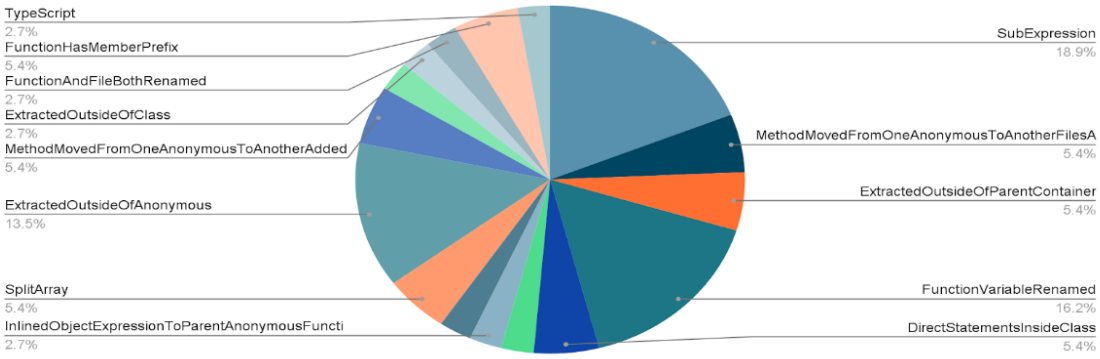

Figure 8: Top reasons behind the low recall of JsDiffer

```
function extractArgs(fn) {
  var fnText = Function.prototype.toString.call(fn).replace(STRIP_COMMENTS, ''),
      args = fnText.match(ARROW_ARG) || fnText.match(FN_ARGS);
  return args;
}
```

```
function stringifyFn(fn) {
  return Function.prototype.toString.call(fn);
}

function extractArgs(fn) {
  var fnText = stringifyFn(fn).replace(STRIP_COMMENTS, ''),
      args = fnText.match(ARROW_ARG) || fnText.match(FN_ARGS);
  return args;
}
```

Figure 9: The expression `Function.prototype.toString.call(fn)` has been extracted to `stringifyFn` function.

same container. Therefore it's a limitation of RefactoringMiner's approach.

Figure 9 represents a sub expression base matching problem where an expression becomes a statement in a extracted function. Even though it is textually identical it is not currently possible to match by RefactoringMiner's approach.

On the other hand, JsDiffer missed Inline Function refactorings where the removed function resides outside of its destination container or has its statements become part of an expression of the destination container. Figure 10 however, describes a different scenario where JsDiffer fails to support the addition of named argument passing (i.e. configurable:

```
    this.browserWindow = new BrowserWindow(options)

    if (this.atomApplication.projectSettings != null) {

      this.projectSettings = this.atomApplication.projectSettings

    }


    this.loadDataOverProcessBoundary()
    this.handleEvents()
```

V1

```
    this.browserWindow = new BrowserWindow(options)

    Object.defineProperty(this.browserWindow, 'loadSettingsJSON', {
      get: () => JSON.stringify(Object.assign({
        userSettings: this.atomApplication.configFile.get(),
        projectSettings: this.projectSettings
      }, this.loadSettings))
    })


    this.handleEvents()
```

V2

Figure 10: The function `loadDataOverProcessBoundary` has been inlined to the parent container class. This is not currently detected by JsDiffer.

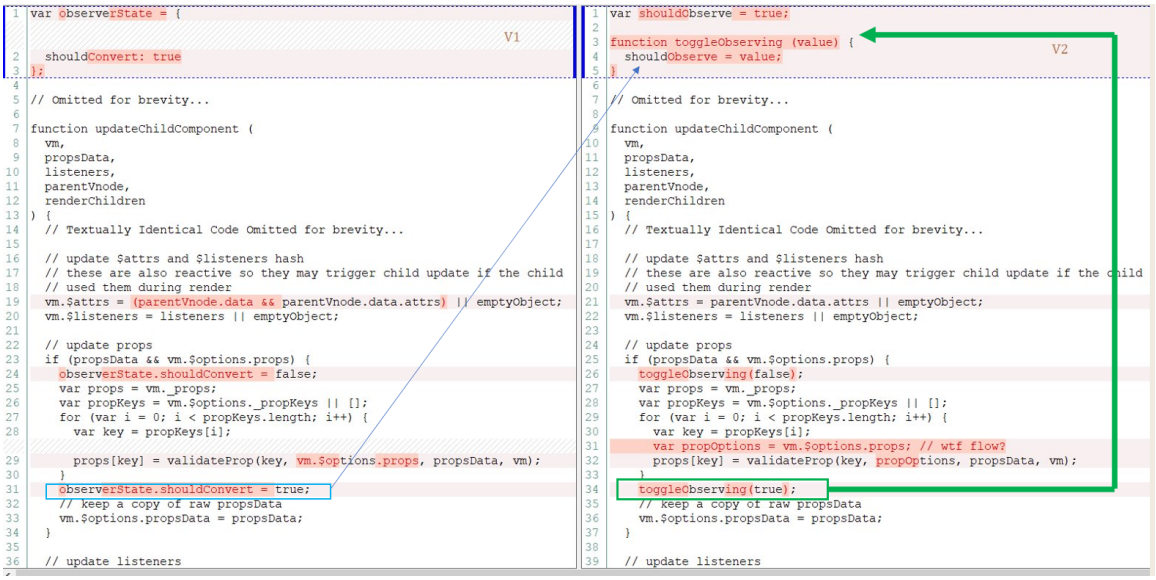

Figure 11: Extract Function `toggleObserving` from `updateChildComponent`

true) and thus cannot match the statement of 155 to 57. In practice, we have found cases where the whole program is passed to a function call as an argument. Therefore, a more sophisticated approach is needed to handle such a part. ReDiff 2.0, however, was able to successfully report this refactoring because it has many textually similar tokens that were matched.

For Rename Class refactorings, the missed cases by JsDiffer are cases where a class contains a single functional expression-based statement. Since, during class matching, RefactoringMiner's approach does not consider any directly written statement, we cannot match if they are renamed.

However, JsDiffer has found 10 unique instances which are not detected by RefDiff 2.0. For example, in Figure 12, although the `toggleOserving` function has been extracted from the `updateChildComponent` method it was not reported by RefDiff but 2.0 JsDiffer was able to successfully identify that refactoring.

JsDiffer is able to keep track of the arguments passed to a function, when detecting extract method refactoring, it can identify when a variable has been parameterized. In this case, JsDiffer considered the parameter "value" of `toggleObserving` function as "true"

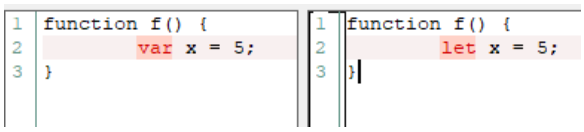

Figure 12: Change Variable Kind refactoring from `var` to `let`

when it encountered the function invoking statement - `toggleObserving(true)`. Therefore it can exactly match the `shouldObserve = value` statement to the statement `observerState.shouldConvert = true;` This is very similar to the "argumentization" process used by RefactoringMiner 2.0 for mapping function invocation with added operations. On the other hand, we assume that because of the low textual similarity between the removed statement and the newly added function, RefDiff 2.0 was not able to detect this refactoring.

4.2.2 RQ2: Accuracy on Detecting Variable Related Refactorings

From 608 commits, JsDiffer reported 355 Rename Variable Refactorings. We validated 73 instances and calculated a precision of 88%. We could not calculate recall as it is an extremely challenging and subjective task to find all the rename variable refactorings performed in a commit. However, our assumption is that most of the missed rename variable refactorings would be inside of a container expression which is currently poorly supported by JsDiffer.

In Addition, to Rename Variable refactorings, we also found Rename Parameter, and Add/Remove Parameter refactoring instances. Only one Change Variable Kind refactoring was reported which turned out to be a true positive. Modern JavaScript allows 3 kinds of named variable kinds - var, let, and const which can be used to declare variables. A Change Variable Kind refactoring occurred when the declaration kind is changed. For example, var can be changed to let. This is somewhat similar to Change Type Refactoring of Java (such as int to long).

After taking a closer look at some of the false positives of Rename Variable refactorings, we have found that without static types, the syntax-aware replacement technique of RefactoringMiner does not perform very well in detecting variable level refactorings. This limitation was also mentioned in their paper. RefactoringMiner 2.0 relies heavily on types while detecting the signature of a method which helps find a pair of methods for comparison even after it has been renamed. This caused many false positives.

4.2.3 RQ3: Performance Comparison

RefDiff 2.0 outperforms JsDiffer in terms of execution time. Table 4 shows the execution time range for both tools. It took less than a second to detect refactorings in more than half of the commits for RefDiff 2.0. On the other hand, for the majority of the commits, JsDiffer took 1 to 9 seconds to process.

Table 4: Execution Time Distribution over 608 commits.

	Commit Count		Percentage of Commits	
Time Range (seconds)	JsDiffer	RefDiff 2.0	JsDiffer	RefDiff 2.0
Less Than 1	202	340	33.22	**55.92**
1 - 9	333	217	**54.77**	35.69
10 - 19	33	13	5.43	2.14
20 - 29	10	16	1.64	2.63
30 -59	23	17	3.78	2.80
Greater Than 60	7	5	1.15	0.82

We narrowed down the performance issue of JsDiffer into two phases: the parsing of source code and generation of source models, and the refactoring detection.

Both JsDIffer and RefDiff 2.0 are written in Java language. Both of them use Babel as the JavaScript source code parser in a virtual node.js environment. However, RefDiff 2.0 gets the source code back to the Java side in a tokenized format as a string where JsDiffer traverses the source code and passes back data from the JavaScript side to the Java Side for each node in the Abstract Syntax Tree. These multiple calls back and forth from Java to

the JavaScript side turned out to be very costly and on average it took about 3 seconds to parse and create a model from a commit.

On the other hand, the performance of detection of refactorings was degraded by Levensthein string distance calculation even though we have removed it for matching statements with container expressions. This occurs when having a large block of code inside an array access (e.g., arr[]) and the overall number of functions and statements is large.

4.3 Limitations and Threats to Validity

The reason behind such a low recall for JsDiffer is because RefactoringMiner employs a top-down approach when matching container elements where it checks the signature of two containers are similar, so that they are potential candidates for matching.

For example, if a class has more than half of its attributes matched with the same types, then our tool performs a diff between the classes similar to RefactoringMiner 2.0. However for JavaScript, since there are no types, the only indicator is the name. So, if the name changes between two versions, the probability of finding a potential candidate to be matched with is reduced and since we do not employ a brute force approach, JsDiffer misses many refactorings.

To remedy this problem, we currently check for child function names up to a nesting depth of 3. That means if two containers have similarly named functions inside their body or in their children/grandchildren's (up to depth 3) they are considered for matching.

Unfortunately, this has proven to be highly inadequate for cases where the two containers do not contain any function declarations. In such a case, we cannot do the statement mapping because two differently named containers have no common function declarations and without static types, it's not possible to properly match the signature of the containers. This essentially became a Chicken and Egg problem where we want to do the statement mapping between two potential container elements but without the statement mappings

sometimes it is not possible to find such candidate container pairs.

We do consider the name and number of the parameters during matching two function signatures. However, due to the style of JavaScript code, we found that in many cases the whole program was written as a functional expression and assigned as a default value to the parameter. Therefore our approach can handle only simple cases and cannot generalize the structure yet.

As for the evaluation, even though we were careful when constructing our oracle, it is possible that we might have wrongly but unintentionally put the wrong validation. For example, whenever both tools reported the same refactoring, we considered it automatically as a True positive for both tools. However, we have found that in at least one case, both tools reported an Extract Function refactoring as Rename Function refactoring.

In addition, JsDiffer by default processes files with .js extensions from a commit. It is very likely that a huge portion of the source code is left unprocessed which is typically found in .html, .jsx, or .ts file extensions. These files may contain vanilla JavaScript, TypeScript, JSX, or a mix of them. To reduce the scope we opted to not process other file extensions at this moment.

Additionally, because of the low number of validated instances for some refactoring, the accuracy of the tools might not reflect their correctness in practice. For example, we validated only one instance of Move Class refactoring and both tools were able to detect that. But only a single instance validation is not enough to judge the strength of the tools.

Moreover, estimating recall for both tools suffers from the lack of information on all the actually applied refactorings in the commit history. However, we do believe precision is more important than recall in a refactoring detection tool. Because having a tool with higher precision produces a less noisy result for the end users. This increases the trust in the tool put in by the developer. Since recall is difficult to calculate and validate fully, precision represents the effectiveness of the tool more explicitly. Especially for empirical

study, it is preferred to have high precision tool because we don't want wrong instances that can adversely affect the findings.

Chapter 5

Conclusion and Future Work

In this thesis, we represent our tool JsDiffer which can detect Refactorings in JavaScript projects. We described our methodology and thoroughly discussed the difference between our tool and compared it with the current state-of-the-art RefDiff 2.0.

Our result showed that even though RefDiff 2.0 performed significantly better than JsDiffer in Recall, our approach was able to find some interesting refactorings that were missed by RefDiff 2.0. Especially our tool was able to find refactorings that were significantly structurally different.

In summary, we learned the followings from this thesis:

1. Though JsDiffer achieved a decent precision of 96% in the oracle it has a low recall of 44% for detecting JavaScript refactorings.

2. JsDiffer achieved a precision of 88% on detecting Rename Variable refactorings.

3. RefDiff 2.0 also outperformed JsDiffer in execution as the majority of the commits were processed in under one second by RefDiff, while JsDiffer took 1-9 seconds to process the majority of the commits in the oracle.

4. JsDiffer was able to detect a single true positive Change Variable Kind refactoring

instance.

Additionally, we learned that JavaScript programs have very different structure compared to Java programs. For instance, typical JavaScript programs have more nesting depth of container elements, declare functions within other functions, and in many case the entire source code of a file is written in a single statement. RefactoringMiner was designed having the typical Java program structure in mind. It matches code at statement level and does not support sub-expression matching (i.e., part of a statement being matched with another statement). However, we found many scenarios where a sub-expression of a statement was extracted into a new function.

On the other hand, RefDiff completely ignores the code structure, and this makes it robust to structural changes. However, we should emphasize that RefDiff by design cannot support low-level refactorings within the body of a function (e.g., variable renames) as RefDiff cannot find token replacements, because the location of the tokens in the code structure is lost after extracting the bag of tokens.

Based on our experiments, we recommend that JavaScript projects may need a hybrid approach that combines both RefDiff and RefactoringMiner approaches, where at first, high-level program elements (functions, classes, files) are matched following RefDiff's approach, and then RefactoringMiner's statement mapping approach is applied to detect low-level refactorings.

Bibliography

[1] Abdullah Almogahed, Mazni Omar, and Nur Haryani Zakaria. Refactoring codes to improve software security requirements. *Procedia Computer Science*, 204:108–115, 2022.

[2] Wellisson GP da Silva, Lisane Brisolara, Ulisses B Corrêa, and Luigi Carro. Evaluation of the impact of code refactoring on embedded software efficiency. In *Proceedings of the 1st Workshop de Sistemas Embarcados*, pages 145–150, 2010.

[3] Luis Cruz, Rui Abreu, and Jean-Noël Rouvignac. Leafactor: Improving energy efficiency of android apps via automatic refactoring. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 205–206. IEEE, 2017.

[4] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In *IFIP Central and East European Conference on Software Engineering Techniques*, pages 252–266. Springer, 2007.

[5] Shaweta Kumar and Sanjeev Bansal. Comparative study of test driven development with traditional techniques. *Int. J. Soft Comp. & Eng*, 3(1):352–360, 2013.

[6] Chaima Abid, Vahid Alizadeh, Marouane Kessentini, Thiago do Nascimento Ferreira, and Danny Dig. 30 years of software refactoring research: a systematic literature review. *arXiv preprint arXiv:2007.02194*, 2020.

[7] Sarah Fakhoury, Devjeet Roy, Adnan Hassan, and Vernera Arnaoudova. Improving source code readability: theory and practice. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 2–12. IEEE, 2019.

[8] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J Decker, and Christian D Newman. An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 26–33, 2018.

[9] Bin Lin, Csaba Nagy, Gabriele Bavota, and Michele Lanza. On the impact of refactoring operations on code naturalness. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 594–598. IEEE, 2019.

[10] 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*, pages 465–475, 2017.

[11] Martina Iammarino, Fiorella Zampetti, Lerina Aversano, and Massimiliano Di Penta. Self-admitted technical debt removal and refactoring actions: Co-occurrence or more? In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 186–190. IEEE, 2019.

[12] Walter Lucas, Rodrigo Bonifácio, Edna Dias Canedo, Diego Marcílio, and Fernanda Lima. Does the introduction of lambda expressions improve the comprehension of java programs? In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, pages 187–196, 2019.

[13] 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, 2012.

[14] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph Johnson, and Danny Dig. A comparative study of manual and automated refactorings. volume 7920, pages 552–576, 07 2013.

[15] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[16] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, 2014.

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

[18] 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*, pages 151–160, 2011.

[19] Peter Weißgerber and Stephan Diehl. Are refactorings less error-prone than other changes? In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 112–118, 2006.

[20] 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 *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 104–113. IEEE, 2012.

[21] Andre Hora, Danilo Silva, Marco Tulio Valente, and Romain Robbes. Assessing the threat of untracked changes in software evolution. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1102–1113, 2018.

[22] Quinten David Soetens, Romain Robbes, and Serge Demeyer. Changes as first-class citizens: A research perspective on modern software tooling. *ACM Computing Surveys (CSUR)*, 50(2):1–38, 2017.

[23] Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. Are refactorings to blame? an empirical study of refactorings in merge conflicts. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 151–162. IEEE, 2019.

[24] Johannes Henkel and Amer Diwan. Catchup! capturing and replaying refactorings to support api evolution. In *Proceedings of the 27th international conference on Software engineering*, pages 274–283, 2005.

[25] 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, page 951–952, New York, NY, USA, 2008. Association for Computing Machinery.

[26] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 348–357. IEEE, 2021.

[27] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 2020.

[28] Danilo Silva, João Silva, Gustavo Jansen De Souza Santos, Ricardo Terra, and Marco Tulio O. Valente. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[29] Leo Brodie. *Thinking Forth*. Punchy Publishing, 2004.

[30] W. F. OPDYKE. Refactoring : An aid in designing application frameworks and evolving object-oriented systems. *Proc. SOOPPA '90 : Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990.

[31] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[32] Raimund Moser, Alberto Sillitti, Pekka Abrahamsson, and Giancarlo Succi. Does refactoring improve reusability? In *International conference on software reuse*, pages 287–297. Springer, 2006.

[33] Dirk Wilking, Umar Farooq Kahn, and Stefan Kowalewski. An empirical evaluation of refactoring. *e Informatica Softw. Eng. J.*, 1(1):27–42, 2007.

[34] 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, 2015.

[35] Diego Cedrim, Leonardo Sousa, Alessandro Garcia, and Rohit Gheyi. Does refactoring improve software structural quality? a longitudinal study of 25 projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering*, pages 73–82, 2016.

[36] Jevgenija Pantiuchina, Michele Lanza, and Gabriele Bavota. Improving code: The (mis) perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 80–91. IEEE, 2018.

[37] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.

[38] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.*, 11(2):5–1, 2012.

[39] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 268–278. IEEE, 2013.

[40] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. Many-objective software remodularization using nsga-iii. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):1–45, 2015.

[41] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. Recommending refactoring operations in large software systems. In *Recommendation Systems in Software Engineering*, pages 387–419. Springer, 2014.

[42] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(3):1–53, 2016.

[43] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object systems*, 3(4):253–263, 1997.

[44] Davood Mazinanian, Nikolaos Tsantalis, Raphael Stein, and Zackary Valenta. Jdeodorant: clone refactoring. In *Proceedings of the 38th international conference on software engineering companion*, pages 613–616, 2016.

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

[46] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 483–494, New York, NY, USA, 2018. ACM.

[47] Iman Hemati Moghadam, Mel Ó Cinnéide, Faezeh Zarepour, and Mohamad Aref Jahanmir. Refdetect: A multi-language refactoring detection tool based on string alignment. *IEEE Access*, 9:86698–86727, 2021.

[48] 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*, pages 35–38, 2008.

[49] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making program refactoring safer. *IEEE software*, 27(4):52–57, 2010.

[50] Rrezarta Krasniqi and Jane Cleland-Huang. Enhancing source code refactoring detection with explanations from commit messages. pages 512–516, 02 2020.

[51] Xi Ge and Emerson Murphy-Hill. Manual refactoring changes with automated refactoring validation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 1095–1105, New York, NY, USA, 5 2014. Association for Computing Machinery.

[52] Xi Ge, Saurabh Sarkar, Jim Witschey, and Emerson Murphy-Hill. Refactoring-aware code review. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 71–79, 2017.

[53] Xi Ge, Quinton DuBose, and Emerson Murphy-Hill. Reconciling manual and automatic refactoring. *Proceedings - International Conference on Software Engineering*, pages 211–221, 06 2012.

[54] 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, page 222–232, Zurich, Switzerland, 6 2012. IEEE Press.

[55] 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, page 166–177, New York, NY, USA, 10 2000. Association for Computing Machinery.

[56] G. Antoniol, M. Di Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.*, pages 31–40, 2004.

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

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

[59] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. 07 2006.

[60] A.Z. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, pages 21–29, 1997.

[61] Benjamin Biegel, Quinten David Soetens, Willi Hornig, Stephan Diehl, and Serge Demeyer. Comparison of similarity metrics for refactoring detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 53–62, New York, NY, USA, 2011. Association for Computing Machinery.

[62] Benjamin Biegel and Stephan Diehl. Highly configurable and extensible code clone detection. In *2010 17th Working Conference on Reverse Engineering*, pages 237–241, 2010.

[63] Zhenchang Xing and Eleni Stroulia. Refactoring detection based on umldiff change-facts queries. In *2006 13th Working Conference on Reverse Engineering*, pages 263–274, 2006.

[64] Zhenchang Xing and Eleni Stroulia. Differencing logical uml models. *Autom. Softw. Eng.*, 14:215–259, 07 2007.

[65] 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, page 54–65, New York, NY, USA, 2005. Association for Computing Machinery.

[66] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. Template-based reconstruction of complex refactorings. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

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

[68] Gustavo Soares, Rohit Gheyi, Emerson Murphy-Hill, and Brittany Johnson. Comparing approaches to analyze refactoring activity on software repositories. *J. Syst. Softw.*, 86(4):1006–1022, April 2013.

[69] Danilo Silva and Marco Tulio Valente. Refdiff: Detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 269–279, 2017.

[70] Liang Tan and Christoph Bockisch. A survey of refactoring detection tools. In *Software Engineering (Workshops)*, pages 100–105, 2019.

[71] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. A multidimensional empirical study on refactoring activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, page 132–146, USA, 2013. IBM Corp.

[72] Benjamin Biegel, Quinten David Soetens, Willi Hornig, Stephan Diehl, and Serge Demeyer. Comparison of similarity metrics for refactoring detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 53–62, New York, NY, USA, 2011. Association for Computing Machinery.

[73] Oscar Chaparro, Gabriele Bavota, Andrian Marcus, and Massimiliano Di Penta. On the impact of refactoring operations on code quality metrics. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 456–460, 2014.

[74] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., USA, 1986.

[75] 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, 2017.

[76] Rodrigo Brito and Marco Tulio Valente. Refdiff4go: Detecting refactorings in go. SBCARS '20, page 101–110, New York, NY, USA, 2020. Association for Computing Machinery.

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

[78] Angana Chakraborty and Sanghamitra Bandyopadhyay. Fogsaa: Fast optimal global sequence alignment algorithm. *Scientific reports*, 3(1):1–9, 2013.

[79] Tiago L Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[80] Kecia AM Ferreira, Mariza AS Bigonha, Roberto S Bigonha, Luiz FO Mendes, and Heitor C Almeida. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2):244–257, 2012.

[81] Paloma Oliveira, Marco Tulio Valente, and Fernando Paim Lima. Extracting relative thresholds for source code metrics. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 254–263. IEEE, 2014.

[82] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Aiko Yamashita. Automatic metric thresholds derivation for code smell detection. In *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*, pages 44–53. IEEE, 2015.

[83] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191, 2016.

[84] Danny Dig and Ralph Johnson. Automated upgrading of component-based applications. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 675–676, 2006.

[85] Maurício Aniche, Christoph Treude, Andy Zaidman, Arie Van Deursen, and Marco Aurélio Gerosa. Satt: Tailoring code metric thresholds for different software architectures. In *2016 IEEE 16th international working conference on source code analysis and manipulation (SCAM)*, pages 41–50. IEEE, 2016.

[86] 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, 2016.