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 2021

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Mosabbir Khan Shiblu**
Entitled: **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. Chair

_____ External
Dr. ExternalToProgram

_____ Examiner
Dr. Examiner1

_____ Examiner
Dr. Examiner2

_____ 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

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 understanding 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 function level refactorings such as rename variable, rename parameter etc.

In this study we present JSRMiner which supports **16** different refactoring operations including several function level refactorings in JavaScript projects . Although the tool is inspired by RefactoringMiner it differs quite a lot from refactoring miner in terms of structure. We evualated JSRMiner against an oracle of **300** refactoring instances mined from 10 open source javascript projects and compared with RefDiff 2.0. Our results indicate that JSRMiner can achieve satisfactory precision although cannot beat RefDiff 2.0.

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 to 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 activities plays an important role in the modern software development life cycle. In addition to referring to source code changes that improve its maintainability, refactoring is frequently used to denote changes that improve software performance, software security, and even the energy consumption of a system. In an agile environment, it enables the limited upfront design of the software to advance. On the other hand, for test-driven environment, it is regarded as a necessary activity in keeping the code-base compliant for further development.

A recent survey paper [1] reported over 3,000 papers on refactoring topics which asserts its popularity in modern research. Many researchers empirically investigated the benefits of refactorings by studying how the renaming of identifiers affects code readability [23], how and why developers rename identifiers [56], the impact of refactoring on code naturalness [42], the impact of refactoring on code smells [16], the co-occurrence of refactoring and self-admitted technical debt removal [34], and how the introduction of Lambda expressions affects program comprehension [43].

Therefore, by detecting refactoring activity in software projects, researchers can better

understand software evolution. Earlier studies used such information to reveal the usage of refactoring tools [49], [50], the motivations behind refactoring [38], [39], [62], the risks associated with refactoring [38], [39], [36], [71], [7], and the effect of refactoring on code quality metrics [38], [39]. Additionally, the accuracy of source code evolution analysis can be improved by keeping track of recordings because files, classes, or functions may have their histories split by refactorings such as Move or Rename [33].

review

However, detecting refactoring is not a trivial task due to the fact that developers rarely document the refactoring operations they perform [2]. Besides, refactoring operations are often performed together with –or as a consequence of– other changes [3], which makes it even harder to distill them out of tangled code changes.

Several refactoring detection tools such as REFACTORINGMINER [4] and REFDIFF [5] have been proposed to mine refactoring operations from software projects.

Most of these tools mainly focus on the Java programming language [4], [6], [7], while REFDIFF [5] also detects refactoring for projects written in JavaScript and C.

None of these tools can be used to extract refactorings performed in Python. Given the fact that Python is currently one of the most popular programming languages1, a refactoring detection tool specifically designed for Python might unlock many new research opportunities. For example, Python is the dominant language in the fields of scientific computing, data science, and machine learning [8]. 1https://www.tiobe.com/tiobe-index/ Therefore, detecting refactoring in Python can allow to gain specific insights in these domains. Dilhara and Dig [9] have taken the first step to address this issue and developed PYTHON-ADAPTED REFACTORINGMINER2, which converts Python programs into Java and uses REFACTORINGMINER [4] to detect refactorings.

However, there are considerable differences between these two languages, let alone the language grammar. For example, Python checks types at runtime while Java is a statically

typed language. Moreover, Java is class-based and object-oriented, while Python projects can also follow other programming styles such as functional and imperative programming. Inspired by REFACTORINGMINER [4], we present PYREF, a tool that automatically detects mainly method-level refactoring operations from Python projects. To evaluate the performance of PYREF, we ran it on three real-world Python projects, and manually validated the refactoring detection. We also compared PYREF with the only publicly available refactoring detection tool for Python, namely PYTHON-ADAPTED REFACTORING MINER. On average, PYREF achieves a precision of 89.6than the current state-of-the-art. This results show the potential of PYREF for refactoring detection in Python projects. The remainder of the paper is structured as follows. Section II introduces current refactoring detection tools. The detailed techniques behind PYREF are described in Section III. Section IV reports the design and results of the study we performed to assess the performance of PYREF. The limitations of our tool are discussed in Section V. Finally, Section VI concludes the paper.

TODO

1.2 Thesis Statement

TODO

1.3 Objectives and Contributions

TODO

1.4 Outline

The rest of the thesis is organized as follows...

Chapter 2

Related Wok

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

Leo Brodie [15] 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 [52] who generalized refactorings as any source code transformations that improves the understandability and re-usability of source code.

In an early work, by Mens and Tourwe [45], 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 [48] [73] [8] [17] [55] [2], detecting refactoring opportunities [25] [54], refactoring recommendations, [46]; [9] [53], and automated refactoring tools [59] [44] [37] [69] [61] [47].

2.1 Refactoring Detection Approach

2.1.1 Detection Using Meta Data

Several studies have used metadata such as commit messages from version control systems to detect refactorings.

Ratzinger et al. [58] searched for a predefined set of terms (e.g. ”refactor") in commit messages to classify them as refactoring changes. Kim et al. [39] reported that in some cases, a branch may be created exclusively to refactor the code. Soares et al. [65] 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 [40] implemented CMMiner that is capable of detecting 12 refactoring types based on analysing commit logs provided by developers.

2.1.2 Detection By Tracking IDE Activities

Murphy-Hill et al. [49] 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, they often interleave refactorings with other behavior-modifying programming activities. Furthermore, developers rarely explicitly report their refactoring activities in commit messages.

Negara et al. [50] 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 on 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 [50], GHOSTFACTOR [31] and REVIEWFACTOR [32] infer fully completed refactorings by monitoring the fine-grained code changes in

real-time inside the IDE. On the other hand BENEFACTOR [30] and WITCHDOCTOR [28] 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. [20] 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, 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 on 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 accuracy 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. [5] 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 [72] developed the first technique for identifying class-level/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 [35] to compare their bodies and then rank them. CCFINDER is also configured to ignore whitespaces/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 [40], [49]. Lastly, the authors stated that their technique is susceptible to multiple refactorings performed on the same entity.

Dig et al. [21] developed an Eclipse plug-in named REFACTORINGCRAWLER which initially uses a computationally inexpensive text-based similarity metric - Shingles encoding [14] 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 on 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 [72] the authors manually discovered the applied refactorings by inspecting their commits/ release notes for computing recall while they inspected the source code to compute precision. Later, Biegel et al. [11] replicated Weißgerber's approach using three different similarity metrics: CCFinder (text-based), JCCD [10] (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 [75] used both textual and structural similarity to detect refactorings between two versions of a system in their Eclipse plug-in named JDEVAN [76] [77]. JDEVAN initially constructs two UML logical design models from the source code corresponding to two versions of a Java system. Next, using UMLDiff [74], 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 [75] 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 on 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 [37] by Pete et al. [57] is capable of detecting 63 of Fowlers' catalog [29] of 72 refactoring types which contains the most comprehensive list of refactoring types to date. 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. changefacts) 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 is created and a number of PullUp 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 subsequence algorithm. The tool was tested on three open-source Java programs and precision of 95% and recall of 79% were reported. However, later studies reported lower precision and recall for Ref-Finder [64] [63] [66]. Considering refactorings applied on isolation (root canal refactorings [49]) and ignoring refactorings with overlapping changes (i.e. floss refactorings [49]) 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 refactoring detection tools RefDiff [61], RefactoringMiner 2.0 [68] and RefDetect [47] which are closely related to our work.

RefactoringMiner

Tsantalis et al. [67] proposed a tool based on an extended, lightweight variation of the UMLDiff [74] 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. [12] 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. [62] extended and re-introduced the tool as Refactoring Miner 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, Refactoring Miner 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. [19].

RefDiff 1.0

In their tool REFDIFF, Silva and Valente [63] introduced the concept of analyzing only the changed, added, or deleted files between two versions of a program to detect refactorings. Its is capable of detecting 13 high-level refactoring types through static analysis and code similarity comparison. As a first step, Ref Diff 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 similarity between code entities, a variation of the TF-IDF weighting scheme [60] 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 [62]. Finally, similar to the evaluation performed by Perte et al. [57], they evaluated REFDIFF based on an oracle of root canal refactorings applied by graduate students in 20 open-source projects. The evaluation suggested that REFDIFF surpassed RMiner [62], RefactoringCrawler [21] and Ref-Finder [57] in terms of performance and accuracy.

RefactoringMiner 1.0 /RMiner

Later, Tsantalis et al. [69] proposed a major evolution of their existing RefactoringMiner [62] [67] 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 change, added, or deleted files of a commit however, unlike its competitors such as REFDIFF, REF-FINDER, UMLDIFF, and REFACTORINGCRAWLER; RMiner can tolerate unparseable programs. Consequently, this technique can achieve better accuracy and efficiency by decreasing the number of incorrect code

entity matches as the majority of the change history of software systems cannot be successfully compiled [70]. 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 so far, surpassing RefDiff [63], the previous state-of-the-art, which achieved a precision of 75.7% and a recall of 85.8% in this dataset. The superiority of RMiner is also confirmed by Tan and Bockisch [66] where it emerges as the winner among its competitors: RefactoringCrawler [21], Ref-Finder [57] and RefDiff [63].

RefDiff 2.0

In continuation to their previous work, Silva et al. [61] upgraded REFDIFF [63] and represented as 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 Code Structure Tree (CST) that abstracts away the detail of a particular language. Each node in CST is

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

represented by higher-level entities such as classes, functions, etc. 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 [69] [2] that was used to evaluate RMiner [69] and a 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 experiments. For JavaScript, the computed precision was 91% using 87 refactorings and recall was 88% using 65 refactoring instances. Later Brito and Valente [13] 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 [69] and introduced RefactoringMiner 2.0[3] [68]. The tool is capable of detecting over 80 different types of refactoring operations including low-level ones that occur in 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 evaluation, the authors compare their tool with existing tools including its predecessor RefactoringMiner 1.0/ RMiner [69], and RefDiff 2.0 [61] 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%) from the tools

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

evaluated. Zarina [4] from JetBrains-Research [5] leads and maintains kotlinRMiner which is essentially an extension for Refactoring Miner 2.0. Additionally, Python extensions are also created by Atwi et al. [6] and Dilhara et. al. [6].

RefDetect

In contrast to RefDiff 2.0 [61] which uses token similarity and RefactoringMiner 2.0 [68] 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 [18]) 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. 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 on the same aforementioned publicly available refactoring oracle. The authors

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

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 as matched. Since developers often perform other maintenance activities (e.g. bug fixes, performance) during refactoring [49], similarity thresholds often help endure such changes. Modern refactoring detection tools comes 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 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 [3], [24], [51], [26], [27]. Dig et al. [22] reported that precision and recall can vary significantly for the same software system based on different values of thresholds. Moreover, Aniche et al. [4] has 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 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 [68].

Dependence on built project versions Tools like RefactoringCrawler, RefFinder and UMLDiff requires building the project versions in order to retrieve structural information such as field access, method call, subtyping and overriding etc. To accurately obtain such information in Java projects, a compiler such as 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 [70]. As a result, these tools may not be suitable for performing large-scale refactoring detection in the entire commit hisoty of a project.

Incomplete oracle: Calculating the true recall for refactoring detection studies has always been very challenging and debatable since it is hard to identify all the refactorings have been performed in commit. Inspecting commit messages has been proved to be an unreliable indicator of refactorings in a commit as developers do not always mention them in the commit message [49]. Moreover, Moreno et al. [41] 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 and requires expertise and does not guarantee of finding all the refactorings.

Java Based Literature: Majority of the existing approaches support only Java systems. In fact, to the best of our knowledge no other tool can detect refactorings in JavaScript ecosystem except RefDiff 2.0. Even so, because of the nature of its detection methodology,

RefDiff 2.0 cannot detect method level refactorings such as rename variable, add parameter etc.

Chapter 3

Then

Chapter 4

Conclusion and future work

TODO

Bibliography

[1] C. Abid, V. Alizadeh, M. Kessentini, T. d. N. Ferreira, and D. Dig. 30 years of software refactoring research: a systematic literature review. *arXiv preprint arXiv:2007.02194*, 2020.

[2] E. A. AlOmar, M. W. Mkaouer, A. Ouni, and M. 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.

[3] T. L. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[4] M. Aniche, C. Treude, A. Zaidman, A. Van Deursen, and M. A. 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.

[5] 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.

[6] H. Atwi, B. Lin, N. Tsantalis, Y. Kashiwa, Y. Kamei, N. Ubayashi, G. Bavota, and M. Lanza. Pyref: Refactoring detection in python projects.

[7] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. 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.

[8] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015.

[9] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. Recommending refactoring operations in large software systems. In *Recommendation Systems in Software Engineering*, pages 387–419. Springer, 2014.

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

[11] B. Biegel, Q. D. Soetens, W. Hornig, S. Diehl, and S. 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.

[12] B. Biegel, Q. D. Soetens, W. Hornig, S. Diehl, and S. 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.

[13] R. Brito and M. T. Valente. Refdiff4go: Detecting refactorings in go. SBCARS '20, page 101–110, New York, NY, USA, 2020. Association for Computing Machinery.

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

[15] L. Brodie. *Thinking Forth*. Punchy Publishing, 2004.

[16] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. 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.

[17] D. Cedrim, L. Sousa, A. Garcia, and R. 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.

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

[19] O. Chaparro, G. Bavota, A. Marcus, and M. D. 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.

[20] S. Demeyer, S. Ducasse, and O. 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.

[21] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. 07 2006.

[22] D. Dig and R. 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.

[23] S. Fakhoury, D. Roy, A. Hassan, and V. Arnaoudova. Improving source code readability: theory and practice. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 2–12. IEEE, 2019.

[24] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, and H. C. Almeida. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2):244–257, 2012.

[25] F. A. Fontana, P. Braione, and M. Zanoni. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.*, 11(2):5–1, 2012.

[26] F. A. Fontana, V. Ferme, M. Zanoni, and A. 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.

[27] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191, 2016.

[28] S. R. Foster, W. G. Griswold, and S. 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.

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

[30] X. Ge, Q. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. *Proceedings - International Conference on Software Engineering*, pages 211–221, 06 2012.

[31] X. Ge and E. 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.

[32] X. Ge, S. Sarkar, J. Witschey, and E. Murphy-Hill. Refactoring-aware code review. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 71–79, 2017.

[33] A. Hora, D. Silva, M. T. Valente, and R. Robbes. Assessing the threat of untracked changes in software evolution. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1102–1113, 2018.

[34] M. Iammarino, F. Zampetti, L. Aversano, and M. 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.

[35] T. Kamiya, S. Kusumoto, and K. 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.

[36] M. Kim, D. Cai, and S. 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.

[37] M. Kim, M. Gee, A. Loh, and N. 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.

[38] M. Kim, T. Zimmermann, and N. 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.

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

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

[41] M. Laura, B. Gabriele, M. Di Penta, O. Rocco, M. Andrian, and G. Canfora. Arena: An approach for the automated generation of release notes. 2017.

[42] B. Lin, C. Nagy, G. Bavota, and M. 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.

[43] W. Lucas, R. Bonifácio, E. D. Canedo, D. Marcílio, and F. 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.

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

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

[46] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni. Many-objective software remodularization using nsga-iii. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):1–45, 2015.

[47] I. H. Moghadam, M. Ó. Cinnéide, F. Zarepour, and M. A. Jahanmir. Refdetect: A multi-language refactoring detection tool based on string alignment. *IEEE Access*, 9:86698–86727, 2021.

[48] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi. Does refactoring improve reusability? In *International conference on software reuse*, pages 287–297. Springer, 2006.

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

[50] S. Negara, N. Chen, M. Vakilian, R. Johnson, and D. Dig. A comparative study of manual and automated refactorings. volume 7920, pages 552–576, 07 2013.

[51] P. Oliveira, M. T. Valente, and F. P. 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.

[52] 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.

[53] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. 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.

[54] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. 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.

[55] J. Pantiuchina, M. Lanza, and G. 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.

[56] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. 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.

[57] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[58] J. Ratzinger, T. Sigmund, and H. 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.

[59] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object systems*, 3(4):253–263, 1997.

[60] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., USA, 1986.

[61] D. Silva, J. Silva, G. J. De Souza Santos, R. Terra, and M. T. O. Valente. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

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

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

[64] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson. Comparing approaches to analyze refactoring activity on software repositories. *J. Syst. Softw.*, 86(4):1006–1022, Apr. 2013.

[65] G. Soares, R. Gheyi, D. Serey, and T. Massoni. Making program refactoring safer. *IEEE software*, 27(4):52–57, 2010.

[66] L. Tan and C. Bockisch. A survey of refactoring detection tools. In *Software Engineering*, 2019.

[67] N. Tsantalis, V. Guana, E. Stroulia, and A. 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.

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

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

[70] M. Tufano, F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29, 2017.

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

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

[73] D. Wilking, U. F. Kahn, and S. Kowalewski. An empirical evaluation of refactoring. *e Informatica Softw. Eng. J.*, 1(1):27–42, 2007.

[74] Z. Xing and E. 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.

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

[76] Z. Xing and E. Stroulia. Differencing logical uml models. *Autom. Softw. Eng.*, 14:215–259, 07 2007.

[77] Z. Xing and E. 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.

Appendix A

First Appendix

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{i=n} x_i = \frac{x_1 + x_2 + \ldots + x_n}{n}$$

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need

for special content, but the length of words should match the language.

$$\int_0^\infty e^{-\alpha x^2}\mathrm{d}x = \frac{1}{2}\sqrt{\int_{-\infty}^\infty e^{-\alpha x^2}\mathrm{d}x \int_{-\infty}^\infty e^{-\alpha y^2}\mathrm{d}y} = \frac{1}{2}\sqrt{\frac{\pi}{\alpha}}$$

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

$$\sum_{k=0}^{\infty} a_0 q^k = \lim_{n\to\infty} \sum_{k=0}^{n} a_0 q^k = \lim_{n\to\infty} a_0 \frac{1-q^{n+1}}{1-q} = \frac{a_0}{1-q}$$

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{-p \pm \sqrt{p^2 - 4q}}{2}$$

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest

gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

$$\frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} + \frac{\partial^2 \Phi}{\partial z^2} = \frac{1}{c^2} \frac{\partial^2 \Phi}{\partial t^2}$$

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Appendix B

Concordia Logos

Figure B.1: Concordia University

Figure B.2: Gina Cody School of Engineering and Computer Science (vertical)

Figure B.3: Gina Cody School of Engineering and Computer Science (horizontal)