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

December 7, 2021 _____
Dr. Mourad Debbabi , Dean
Gina Cody School of Engineering and Computer Science

Abstract

Refactoring Detection in JavaScript

Mosabbir Khan Shiblu

TODO Para1

TODO Para2

TODO Para3

TODO Para4

Acknowledgments

I would like to express my gratitude and thanks to my supervisor, Dr. Nikolaos Tsantalis. His invaluable guidance and continuous 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 best 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

TODO

1.1 Motivation

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

Leo Brodie [5] first mentioned the word "Refactoring" in his book "Thinking Forth", originally published in 1984. In addition to describing refactoring techniques, the author also discussed many software development principles and practices.

Over the past few decades, various techniques for detecting refactoring activities have been proposed, implemented, and validated.

2.1 Refactoring Detection Approach

2.1.1 Detection Using Meta Data

2.1.2 Detection by Static Source Code Analysis

Demeyer et al. [6] 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. [1] 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 [16] 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 [12] 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 [13], [14]. Lastly, the authors stated that their technique is susceptible

to multiple refactorings performed on the same entity.

Dig et al. [7] developed an Eclipse plug-in named REFACTORINGCRAWLER which initially uses a computationally inexpensive text-based similarity metric - Shingles encoding [4] 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 [16] 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. [3] replicated Weißgerber's approach using three different similarity metrics: CCFinder (text-based), JCCD [2] (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 [18] used both textual and structural similarity to detect refactorings between two versions of a system in their Eclipse plug-in named JDEVAN [19] [20]. JDEVAN initially constructs two UML logical design models from the source code corresponding to two versions of a Java system. Next, using UMLDiff [17], 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 [18] 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.

2.1.3 Real-time Detection

Murphy-Hill et al. [14] 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. [15] 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 [15], GHOSTFACTOR [10] and REVIEWFACTOR [11] infer fully completed refactorings by monitoring the fine-grained code changes in real-time inside the IDE. On the other hand BENEFACTOR [9] and WITCHDOCTOR [8] offer code completion by detecting ongoing manual refactorings.

Chapter 3

Then

Chapter 4

Conclusion and future work

TODO

Bibliography

[1] 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.

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

[3] 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.

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

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

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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)