MINING REFACTORINGS FROM VERSION HISTORIES: STUDIES, TOOLS, AND APPLICATIONS

DANILO FERREIRA E SILVA

MINING REFACTORINGS FROM VERSION HISTORIES: STUDIES, TOOLS, AND APPLICATIONS

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Orientador: Marco Túlio de Oliveira Valente

Belo Horizonte

Janeiro de 2020

DANILO FERREIRA E SILVA

MINING REFACTORINGS FROM VERSION HISTORIES: STUDIES, TOOLS, AND APPLICATIONS

Thesis presented to the Graduate Program in Computer Science of the Universidade Federal de Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Marco Túlio de Oliveira Valente

Belo Horizonte

January 2020

© 2020, Danilo Ferreira e Silva.
Todos os direitos reservados.

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa CRB 6ª Região nº 1510

Silva, Danilo Ferreira e.

S586m Mining refactorings from version histories: studies, tools, and applications/ Danilo Ferreira e Silva — Belo Horizonte, 2020.
xxi, 105 f. il.; 29 cm.

Tese (doutorado) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação;
Orientador: Marco Túlio de Oliveira Valente.

1. Computação – Teses. 2. Refatoração de software. 3. Engenharia de Software. 4. Evolução de software. 5. Mineração de repositórios de software. I. Orientador. II. Título.

CDU 519.6*32(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Mining Refactorings from Version Histories: Studies, Tools, and Applications

DANILO FERREIRA E SILVA

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE - Orientador
Departamento de Ciência da Computação - UFMG

PROF. [redacted]
Departamento de Sistemas e Computação - UFCG

PROF. ANDRÉ CAVALCANTE HORA
Departamento de Ciência da Computação - UFMG

PROF. EDUARDO MAGNO LAGES FIGUEIREDO
Departamento de Ciência da Computação - UFMG

PROF. PAULO HENRIQUE MONTEIRO BORBA
Centro de Informática - UFPE

Belo Horizonte, 6 de Fevereiro de 2020.

Agradecimentos

Gostaria de ressaltar minha gratidão a todos aqueles que me ajudaram na desafiadora jornada do doutorado.

Em primeiro lugar, sou imensamente grato pelo empenho e dedicação de meu orientador, professor Marco Túlio Valente, que ofereceu contribuição inestimável e todo o apoio necessário para que este trabalho fosse desenvolvido.

Também ressalto a importância do professor Nikolaos Tsantalis, professor André Hora, professor Ricardo Terra, professor Gustavo Santos, João Paulo Silva e professor Eduardo Figueiredo, com os quais tive o privilégio de trabalhar em conjunto, resultando em importantes publicações para meu doutorado.

Agradeço também aos meus colegas do grupo de pesquisa ASERG, sempre prontos para ajudar nas dificuldades e calorosos ao celebrar nossas conquistas.

Agradeço ao Programa de Pós-Graduação em Ciência da Computação (PPGCC) por oferecer um curso de tal nível de excelência.

Ressalto ainda meu agradecimento aos professores Rohit Gheyi, Paulo Borba, Eduardo Figueiredo e André Hora por aceitarem o convite de participar de minha defesa.

Por fim, agradeço a minha família, que sempre ofereceu apoio e motivação para que eu não me deixasse abater pelas dificuldades e continuar em busca de meus objetivos.

Resumo

Refatoração de código é uma prática importante no desenvolvimento de sistemas e um fator essencial para entender a evolução de um software. Sendo assim, pesquisadores frequentemente reportam e discutem a prática de refatoração em sistemas reais. Infelizmente, estudos empíricos sobre refatoração são frequentemente limitados pela dificuldade de se obter informações confiáveis sobre a atividade de refatoração e muitas questões permanecem em aberto. Nesta tese, primeiro investigamos uma importante questão: *por que desenvolvedores refatoram?* Para esse fim, desenvolvemos dois estudos empíricos em larga escala, baseados na mineração de refatorações em históricos de versões. Inicialmente, investigamos a relação entre a refatoração Extrair Método e reúso de código. Após analizar mais de 10 mil revisões de 10 sistemas, encontramos evidências de que em 56,9% dos casos tal refatoração é motivada pelo reúso de código. Em seguida, investigamos as motivações para refatorações encontradas em sistemas de código aberto com base em respostas dos próprios desenvolvedores que as aplicaram. Como resultado, compilamos um catálogo com 44 motivações distintas para 12 tipos de refatorações. Tal catálogo revela que o esforço de refatoração é mais direcionado pela necessidade de evolução do sistema do que pela resolução de problemas de projeto conhecidos como *code smells*. Notadamente, Extrair Método é a refatoração mais versátil, servindo a 11 propósitos diferentes. Em uma segunda linha de pesquisa, nós propomos RefDiff, uma nova ferramenta para mineração de refatorações em histórico de versões, com suporte a múltiplas linguagens de programação e alta precisão. Nossa ferramenta introduz um algoritmo de detecção de refatorações baseado na *Code Structure Tree* (CST)—uma representação do código fonte que abstrai as particularidades das linguagens de programação—e em uma métrica de similaridade de código baseada na técnica *TF-IDF*. Apesar do seu projeto multilinguagem, nossa avaliação revelou que nossa ferramenta tem precisão (96%) e revocação (80%) equivalentes ao estado da arte em ferramentas especializadas na linguagem Java.

Palavras-chave: Refatoração, Evolução de Software, Mineração de Repositórios.

Abstract

Refactoring is an important aspect of software development and a key factor to understand software evolution. As such, researchers often report and discuss refactoring practice on real software projects. Unfortunately, empirical studies on refactoring are often hindered by the difficulty of obtaining reliable information of refactoring activity, and many questions remain open. In this thesis, we first investigate an overarching question: *why do developers refactor?* To this end, we developed two large-scale empirical studies that rely on mining refactorings from version histories. Initially, we investigated the relationship between Extract Method refactoring and code reuse. After analyzing over 10,000 revisions of 10 open source systems, we found evidence that, in 56.9% of the cases, Extract Method is motivated by code reuse. Next, we investigated the motivations for refactorings applied to open source systems based on feedback from the developers who performed the refactorings. By applying thematic analysis on the collected responses, we compiled a catalogue of 44 distinct motivations for 12 well-known refactoring types. We found that refactoring activity is mainly driven by changes in the requirements and much less by code smells. Notably, Extract Method is the most versatile refactoring operation, serving 11 different purposes. Additionally, we found evidence that the IDE used by the developers affects the adoption of automated refactoring tools. As a second line of research, we propose RefDiff, a novel approach to mine refactorings from version histories that supports multiple programming languages and offers high precision and recall. Our tool leverages existing techniques and introduces a novel refactoring detection algorithm that relies on the Code Structure Tree (CST)—a simple yet powerful representation of the source code that abstracts away the specificities of particular programming languages—and on a code similarity metric based on TF-IDF technique. Despite its language-agnostic design, our evaluation shows that RefDiff's precision (96%) and recall (80%) are on par with state-of-the-art refactoring detection approaches specialized in the Java language.

Keywords: Refactoring, Software Evolution, Mining Software Repositories.

List of Figures

2.1 Overview of the methodology 11
2.2 Example of the *Duplication* scenario 17
2.3 Example of the *Immediate Reuse* scenario 18
2.4 Example of the *Later Reuse* scenario 19
2.5 Proportion of observed reuse patterns 20

3.1 Distribution of (a) age, (b) commits, (c) Java source files, and (d) contributors of repositories 27
3.2 Example of Extract Method refactoring that was required to reach a consensus. 32
3.3 An example of a commit with refactorings and their motivations. 47

4.1 Illustrative diff between two revisions of a system annotated with refactoring operations 59
4.2 CST of both revisions of the example system from Figure 4.1 60
4.3 Algorithm to find relationships 63
4.4 Relationships found in the example from Figure 4.1 66
4.5 Transformation of the body of methods into a multiset of tokens 68
4.6 Illustrative diff of an *Extract Method* refactoring considered as true positive by the validators, taken from commit ce4f629 from *infinispan* project. . . . 74
4.7 Violin plot of execution time per commit in log-10 scale 78

5.1 Typical diff visualization of a code change containing a moved method . . 88
5.2 Diff visualization focused on the moved method 89
5.3 *git-blame* visualization of the `ReactMount.js` file from React project, showing that function `getReactRootElementInContainer` was last modified by Developer A 90
5.4 Diff visualization of the commit in witch the conditional inside function `getReactRootElementInContainer` was introduced, showing that the actual author is Developer B 91

5.5 Hypotetical change made by first developer: Test if `length` is zero 92
5.6 Hypotetical change made by second developer: Move method `median` from class `Main` to class `Statistics` 93
5.7 When we merge the changes from figures 5.5 and 5.6 we get a conflict . . . 93

List of Tables

2.1 Studied repositories 12
2.2 Statistics from the analyzed repositories 14
2.3 Extract Method instances related to code reuse 15
2.4 Extracted methods that were reused along the version history 16

3.1 RefactoringMiner Recall and Precision 30
3.2 Refactoring activity 33
3.3 Extract Method motivations 35
3.4 Motivations for Move Class, Attribute, Method (MC, MA, MM), Rename Package (RP) Inline Method (IM), Extract Superclass, Interface (ES, EI), Pull Up Method, Attribute (PUM, PUA), Push Down Attribute, Method (PDA, PDM) 36
3.5 Manual vs. automated refactoring 40
3.6 Refactoring automation by type 41
3.7 Reasons for not using refactoring tools 41
3.8 IDE popularity 42

4.1 AST nodes that are represented in CSTs 59
4.2 Definitions used in the Algorithm from Figure 4.3 and in the conditions from Table 4.3 62
4.3 Relationship types and the conditions to find them 65
4.4 Java precision and recall results 75
4.5 JavaScript and C repositories used in the evaluation 80
4.6 Search queries for each refactoring type 82
4.7 JavaScript precision and recall results 83
4.8 C precision and recall results 84

Contents

Agradecimentos **ix**

Resumo **xi**

Abstract **xiii**

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**

1.1 Problem and Motivation 1

1.2 Proposed Thesis 4

1.3 Outline 6

2 Investigating Extract Method Refactoring Associated with Code Reuse **9**

2.1 Introduction 9

2.2 Methodology 10

2.2.1 Selection of the Java repositories 11

2.2.2 Detecting refactorings 11

2.2.3 Counting the method invocations 13

2.3 Results 13

2.3.1 RQ1: How often is Extract Method motivated by code reuse? . 14

2.3.2 RQ2: How often is Extract Method motivated by removing duplicate code? 15

2.3.3 RQ3: How often does Extract Method favor code reuse? 15

2.4 Reuse patterns related to Extract Method 16

2.5 Threats to Validity 19

2.6 Conclusion . . . 21

3 An Empirical Study on Refactoring Motivations 23
3.1 Introduction . . . 23
3.2 Related Work . . . 25
3.3 Research Methodology . . . 27
3.3.1 Selection of GitHub Repositories . . . 27
3.3.2 RefactoringMiner Tool . . . 28
3.3.3 Study Design . . . 29
3.3.4 Examined Refactorings . . . 33
3.4 Why do Developers Refactor? . . . 34
3.4.1 Motivations for Extract Method . . . 34
3.4.2 Motivations for Other Refactorings . . . 38
3.5 Refactoring Automation . . . 39
3.5.1 Are refactoring tools underused? . . . 40
3.5.2 Why do developers refactor manually? . . . 40
3.5.3 What IDEs developers use for refactoring? . . . 42
3.6 Discussion . . . 42
3.7 Threats to Validity . . . 45
3.8 Conclusions . . . 46
3.9 Artifact Description . . . 47
3.9.1 License . . . 48
3.9.2 How to contribute? . . . 49

4 Detecting Refactoring in Version Histories 51
4.1 Introduction . . . 52
4.2 Background . . . 54
4.2.1 RefDiff 1.0 . . . 55
4.2.2 Refactoring Miner/RMiner . . . 55
4.2.3 Refactoring Crawler . . . 56
4.2.4 Ref-Finder . . . 57
4.3 Proposed Approach . . . 58
4.3.1 Phase 1: Source Code Analysis . . . 58
4.3.2 Phase 2: Relationship Analysis . . . 61
4.3.3 Code Similarity . . . 67
4.3.4 Implementation details . . . 71
4.4 Evaluation with Java Projects . . . 73

4.4.1 Evaluation Design . . . 73
4.4.2 Results . . . 75
4.4.3 Execution time . . . 77
4.4.4 Threats to Validity . . . 78
4.5 Evaluation with JavaScript and C . . . 79
4.5.1 Evaluation Design: Precision . . . 79
4.5.2 Evaluation Design: Recall . . . 81
4.5.3 Results for JavaScript and C . . . 83
4.5.4 Threats to Validity . . . 84
4.6 Challenges and limitations . . . 84
4.7 Conclusion . . . 85

5 Practical Applications of Refactoring Detection 87
5.1 Refactoring-aware Diff . . . 87
5.2 Tracking changes of a code element . . . 90
5.3 Resolving merge conflicts . . . 92

6 Conclusion 95
6.1 Contributions . . . 95
6.2 Future Work . . . 96

Bibliography 99

Chapter 1

Introduction

1.1 Problem and Motivation

Refactoring is the process of improving the design of an existing code base, without changing its behavior [Opdyke, 1992]. Since the beginning, the adoption of refactoring practices was fostered by the availability of refactoring catalogues, as the one proposed by Fowler [1999]. These catalogues define a name and describe the mechanics of common refactoring operations, as well as demonstrate its application through some code examples. For example, *Extract Method* is a well-known refactoring which consists in turning a code fragment that can be grouped together into a new method, usually to decompose a large and complex method or to eliminate code duplication. As a second example, *Move Method* consists in moving a method from one class to another one, usually because it is more related to features of another class than the class on which it is defined.

There are strong evidences that refactoring is widely adopted by development teams. In fact, it is considered one of the pillars of agile development methodologies. For example, Extreme Programming (XP) [Beck, 2000] and Test Driven Development (TDD) [Beck, 2003] advocate the use of refactoring as an essential step in a software development cycle. Moreover, refactoring is not only discussed in theory, but also observed in practice, as there are several empirical studies that report and discuss refactoring activity found on real software projects [Murphy-Hill et al., 2012; Tsantalis et al., 2013; Kim et al., 2012, 2014; Negara et al., 2013]. In addition, some types of refactoring are applied so often that mainstream development environments, such as Eclipse, IntelliJ IDEA, NetBeans, and Visual Studio, provide some sort of automated support to apply them. All these facts corroborate to the argument that refactoring is an important and well-known technique.

Given that refactoring is an essential aspect of the software development workflow, it is also a key factor to understand software evolution. As such, researchers have investigated refactoring activity to study several questions: how and when developers refactor [Murphy-Hill et al., 2012], the impact of refactoring on code quality [Kim et al., 2012, 2014], the challenges of refactoring [Kim et al., 2012, 2014], the risks of refactoring [Kim et al., 2012, 2014, 2011; Weißgerber and Diehl, 2006a; Bavota et al., 2012; Ferreira et al., 2018], and the relationship between refactoring and merge conflicts [Mahmoudi et al., 2019].

Neverthless, some questions still need to be further investigated. In particular, the motivations behind refactorings are not thoroughly investigated by existing empirical studies. For example, Fowler [1999] describes typical reasons or code smells that motivates each refactoring, but there are no studies verifying that these are the actual motivations for the refactorings applied in practice. Moreover, there may be other motivations driving refactorings other than those documented in refactoring catalogues. Although existing studies interview professional software developers asking the reasons that motivate their refactoring activities [Kim et al., 2014; Wang, 2009], they discuss the issue in a broader manner.

In fact, empirical studies that investigate actual refactorings applied in source code are challenging, because refactoring information is not readily available. For example, suppose that we intend to study the correlation between refactorings and defects on a software component. Many software teams document fixed bugs in issue tracking systems, but we could only correlate them with refactorings if we have a history of refactoring activity in the defective code. Such information is usually not documented, and would have to be inferred from the code changes. If we incorrectly classify code changes as refactoring, or if we fail to find existing refactorings, the conclusions we drawn might be incorrect. In summary, the feasibility of such studies depends on some technique to identify refactoring activity, and the validity of their findings depends on how reliable such technique is. Thus, *refactoring information is valuable to study software evolution.*

Unfortunately, obtaining refactoring information from version histories is a non-trivial task, for three main reasons:

1. Refactorings are rarely documented. Thus, approaches that search for a predefined set of terms in commit messages, such as the one proposed by Ratzinger et al. [2008], suffer from low recall. For example, in a study using Eclipse and Mylyn version histories, Murphy-Hill et al. [2012] found refactorings in 19 out of 40 commits whose messages did not contain any of the refactoring indicator key-

words proposed by Ratzinger et al. [2008]. Moreover, in a study with JHotDraw and Apache Common Collections version histories, Soares et al. [2013] reported a recall of only 16% when using the approach proposed by Ratzinger et al. [2008] to find refactorings.

2. Refactorings are performed interleaved with feature additions and bug fixes [Murphy-Hill et al., 2012]. Thus, analyzing code changes to find refactorings is challenging, because code elements may have changed due to refactoring and to other types of code changes simultaneously. For example, a developer may add more code to an existing method to implement a new feature, and then decide to move it to another class that is more related to it. Identifying that the method has been moved is more challenging in this case, because it is not identical to its original version. In fact, existing approaches that detect refactorings by analyzing code changes have precision and recall issues when used in practice. For example, in a study conducted by Soares et al. [2013], Ref-Finder [Prete et al., 2010; Kim et al., 2010], one well-known refactoring detection approach, achieved only 35% of precision and 24% of recall.

3. Refactorings are not always performed using automated support. Therefore, even if we monitor refactoring tools usage, we may miss a significant portion of the refactoring activity. For example, Murphy-Hill et al. [2012] report that several refactorings found in their study using Eclipse and Mylyn data (89% and 91%, respectively) were not found in the refactoring tool usage logs they collected, suggesting that they were performed manually. As a second example, in a study using data collected from 23 developers, Negara et al. [2013] found that more than half of the refactorings were performed manually.

Despite the aforementioned difficulties, it is important that we develop reliable and efficient approaches that are able to mine refactorings from version histories. The large amount of open source repositories available on platforms such as GitHub offers huge potential for empirical studies on refactoring practice, enabling researchers to verifying previous findings and investigating new research questions. Moreover, such approaches have great potential for practical application in code reviewing, code merging, and tracking code evolution.

1.2 Proposed Thesis

In this thesis, we develop two research lines: empirical studies on refactoring practice and a tool to mine refactorings from version histories. In the first research line, we investigate an overarching question: *Why developers refactor*. To answer that question, we performed two empirical studies.

In **Study 1**, we investigated the relationship between Extract Method refactoring and code reuse. Although Extract Method is usually associated with code smells such as *Long method* [Fowler, 1999], empirical studies suggest that it serves multiple porposes [Tsantalis et al., 2013]. In particular, we suspected that code reuse could be one important motivation for Extract Method. Thus, we proposed three research questions: (i) *How often is Extract Method motivated by code reuse*; (ii) *How often is Extract Method motivated by removing duplicate code*; and (iii) *How often does Extract Method favors code reuse*. To answer these questions, we mined Extract Method refactorings from 10 open source Java repositories using an adapted version of Refactoring Miner, an existing refactoring detection tool [Tsantalis et al., 2013], and also gathered information of method invocations along the entire history of the studies projects. As result, we found that in 56.9% of the cases Extract Method is motivated by code reuse. Additionally, 7.9% of the cases are motivated by removing duplication, and 4.8% of the cases are not motivated by code reuse, but enables reuse in future modifications of the system.

In **Study 2**, we investigated the motivations for refactorings applied to open source systems based on feedback from the developers who did the refactorings. This time, we extended the analysis to 12 well-known refactoring types: Move Class/Method/Attribute, Extract Method, Inline Method, Pull Up Method/Attribute, Push Down Method/Attribute, Rename Package, and Extract Superclass/Interface. Specifically, we monitored 124 Java repositories hosted on GitHub to detect recently applied refactorings, and asked the developers to explain the reasons behind their decision to refactor the code. We developed an automated workflow based on Refactoring Miner that was capable of identifying applied refactorings in a daily basis, enabling us to contact developers in a timely manner, while the refactoring was still fresh in their minds. By applying thematic analysis on the collected responses, we compiled a catalogue of 44 distinct motivations for the 12 studies refactoring types. In summary, we found that refactoring effort is mainly driven by changes in the requirements and much less by code smells. In particular, we found that Extract Method is the most versatile refactoring operation, serving 11 different purposes. Additionally, we also gathered insightful information about the usage of refactoring tools. For example, we

found evidence that the IDE used by the developers affects the adoption of automated refactoring tools.

After our experience with the aforementioned studies, we felt the need for improving refactoring detection approaches. For example, in Study 2, we manually evaluated each refactoring identified by our tool set, and found out that 37% of them were false positives (i.e., we achieved 63% of precision). For that reason, we initiated a second line of research, focusing on improving the state-of-the-art on refactoring detection approaches.

First, we introduced **RefDiff**, a novel refactoring detection approach, whose main goal was to improve precision over existing tools. Our approach employs a combination of heuristics based on static analysis and code similarity to detect 13 well-known refactoring types. One of its distinguishing features is using an adaptation of the classical TF-IDF similarity measure from information retrieval to compute code similarity. We evaluated precision and recall of our tool using an oracle of known refactorings applied by students, and compared it with three existing approaches, namely Refactoring Miner [Tsantalis et al., 2013], Refactoring Crawler [Dig et al., 2006], and Ref-Finder [Kim et al., 2010]. As result, our approach achieved 100% of precision and 88% of recall, surpassing the three tools subjected to the comparison.

Later, after further studies, we proposed an improved version of our tool, named as **RefDiff 2.0**. Besides introducing improvements to our detection heuristics, we redesigned our tool to support multiple programming languages. Our revised refactoring detection algorithm relies on the Code Structure Tree (CST), a simple yet powerful representation of the source code that abstracts away the specificities of particular programming languages. Along with the core algorithm, we provide plugins for three programming languages: Java, JavaScript, and C. In this study, we also performed a large-scale evaluation of our tool using an oracle of 3,248 real refactorings, applied across 538 commits from 185 open source Java repositories. In this extended evaluation, we compared our tool with RMiner, which is an evolution of Refactoring Miner [Tsantalis et al., 2018] and the current state-of-the-art in Java refactoring detection. As result, RefDiff 2.0 achieves 96.4% of precision and 80.4% of recall. RefDiff's precision and recall is on par with RMiner (98.8% of precision and 81.3% of recall), which is a key achievement because our approach is not specialized in a single language. Moreover, our evaluation in JavasScript and C also showed promising results. RefDiff's precision and recall are respectively 91% and 88% for JavaScript, and 88% and 91% for C.

1.3 Outline

Three out of six chapters of this theses consists in studies published in software engineering conference and journals. Therefore, these chapters preserve the original structure of the manuscripts in order to facilitate independent read. We organized the remainder of this work as follows:

Chapter 2: Investigating Extract Method Refactoring Associated with Code Reuse. In this chapter we present **Study 1**, which investigates the relationship between Extract Method refactoring and code reuse. This chapter consists of a translated version of the following publication:

Silva, D., Valente, M. T., and Figueiredo, E. (2015) Um Estudo sobre Extração de Métodos para Reutilização de Código. In *18th Conferencia Iberoamericana de Software Engineering (CIbSE) – Experimental Software Engineering Track (ESELAW)*, pages 404–417.

Chapter 3: An Empirical Study on Refactoring Motivations. In this chapter we present **Study 2**, which investigates the motivations behind refactorings applied in 124 Java projects hosted on GitHub. This chapter consists of the following publication:

Silva, D., Tsantalis, N., and Valente, M. T. (2016) Why we refactor? confessions of GitHub contributors. In *24th International Symposium on the Foundations of Software Engineering (FSE)*, pages 858–870 (Distinguished paper award).

Chapter 4: Detecting Refactoring in Version Histories. In this chapter, we present **Study 4**, which describes RefDiff, an approach to detect refactoring in version histories. Besides, we also present an evaluation of our tool in Java, JavaScript, and C projects. This chapter consists of the following manuscript, currently under review:

Silva, D., Silva J. P., Santos, G., Terra, R., and Valente, M. T. (2019) RefDiff 2.0: A Multi-language RefactoringDetection Tool. Under review in a journal.

It is worth noting that this chapter is also based on the following publication, in which we initially proposed **RefDiff** (superseded by **RefDiff 2.0**):

Silva, D. and Valente, M. T. (2017) RefDiff: Detecting Refactorings in Version Histories. In *14th International Conference on Mining Software Repositories (MSR)*, pages 1–11.

Chapter 5: Practical Applications of Refactoring Detection. In this chapter, we describe practical problems that could benefit from refactoring detection approaches, such as RefDiff.

Chapter 6: Conclusion. This final chapter concludes the thesis, presents our contributions, and gives suggestions for future research.

Chapter 2

Investigating Extract Method Refactoring Associated with Code Reuse

Refactoring is a well-know technique that is frequently studied by the scientific community. However, little is known about the actual reasons developers refactor. In this study, we investigate the relationship between Extract Method refactoring and code reuse, in order to better understand the motivations behind it. After analyzing over 10,000 revisions of 10 open source systems, we found evidence that, in 56.9% of the cases, Extract Method is motivated by code reuse. Also, in a small portion of these cases (7.9%), reuse eliminates duplicate code that already exists in the system. Finally, we also find that there are cases in which Extract Method favored long-term code reuse, even though their initial motivation were not code reuse.

2.1 Introduction

There is a common sense that refactoring effort is mainly driven by certain code patterns known as *Bad Smells* [Fowler, 1999], which signal system design problems. However, there are few empirical studies investigating the actual motivations behind refactoring, especially for those refactoring types that may serve multiple purposes. In particular, *Extract Method*, one of the most widely applied refactorings in practice [Murphy-Hill et al., 2012; Negara et al., 2013], serves multiple porposes. For example, Tsantalis et al. [2013] found nine distinct motivations for this refactoring, some of them related to the need for system extension rather than to code problems.

For this reason, this study investigates the relationship between Extract Method refactoring and code reuse. We suspect that in many cases in which a method is extracted the developer intends to reuse code. For example, when adding a new feature, a developer may identify a code snippet in the system that shares common behavior with the new feature, extract such code as a new method, and reuse it. Specifically, we investigate three research questions:

RQ1 How often is Extract Method motivated by code reuse?

RQ2 How often is Extract Method motivated by removing duplicate code?

RQ3 How often does Extract Method favors code reuse?

We believe that answers to these questions can contribute to a better understanding of why development teams refactor their code. Such knowledge may be relevant to propose or improve techniques and methodologies employed in software development. In particular, it may be possible to refine existing tools specialized in recommending the Extract Method [Silva et al., 2014; Tsantalis and Chatzigeorgiou, 2011] refactorings, as new heuristics can be developed to address the most frequently occurring scenarios in practice.

The remainder of this chapter is organized as follows. Section 2.2 describes the methodology used in this study. Section 2.3 presents the results obtained and answers the research questions raised. Section 2.4 formalizes and provides examples for the reuse patterns we found during the analysis. Section 2.5 discusses threats to the validity of the study. Finally, Section 2.6 presents the conclusions.

2.2 Methodology

The methodology employed in this study, illustrated in Figure 2.1, can be divided into three steps:

1. We selected a set of Java repositories from GitHub using predefined criteria.
2. We detected *Extract Method* refactorings applied to the systems with the aid of a tool. To this end, each code change (*commit*) in the version history was analyzed.
3. We counted the number of invocations of the extracted methods along the version history. To this end, all future revisions were analyzed to identify method invocations, regardless of when they were introduced.

In the following sections we describe the details of each step.

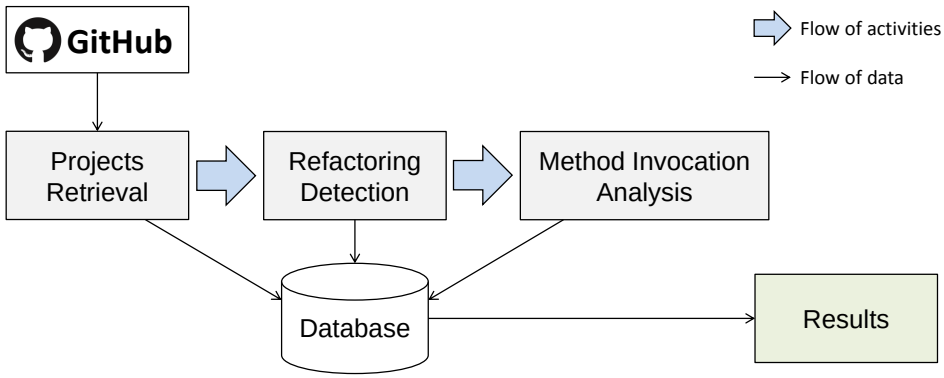


Figure 2.1: Overview of the methodology

2.2.1 Selection of the Java repositories

To select the GitHub repositories, we defined the criteria listed below, according to good practices on mining GitHub repositories described in the literature [Kalliamvakou et al., 2014]:

- Projects must have Java as the primary language, due to limitations of the tools used in the analysis.
- Projects must have at least six months of development to avoid projects that have not gone through a relevant maintenance time.
- Projects must have at least 200 revisions for the same reasons as the previous restriction.
- Projects should not be derived (*forks*) from another project to avoid duplicate data.
- The projects obtained must be the 100 most popular projects that meet the other criteria, using the `stargazers_count` field as a metric.

From the set of 100 repositories, we selected a random sample of 10 systems to perform this study due to limitations imposed by the long time required to perform the analysis on the complete data. Table 2.1 lists the selected repositories and a brief description of each. Note that there are well-known projects among them, such as the JUnit testing framework and the Clojure programming language.

2.2.2 Detecting refactorings

In this step, we analyzed each commit from the selected repositories to find instances of Extract Method refactoring. For this task, we used an automated approach to find

refactorings proposed by Tsantalis et al. [2013], which we will denote as RefactoringMiner 0.1. Such approach compares the source code before and after a code change and employs a combination of heuristics designed to identify eleven types of refactoring. In this study, our interest lies in Extract Method refactoring, whose detection heuristics are described below.

Let M^+ be the set of methods that were added between two consecutive revisions v and v'. Additionally, let $M^=$ be the set of methods that exist both in v and v' (they were neither added or deleted between revisions). A method m_j is extracted from another method m_i when four conditions are met:

- $m_i \in M^=$
- $m_j \in M^+$
- The body of m_i before the change contains the statements within the body of m_j
- The body of m_i after the change contains an invocation of m_j

If the above conditions are met for more than one pair (m_i, m_j) for the same m_j, this means that the method was extracted from more than one location. For example, if these conditions hold for two pairs (m_1, m_2) and (m_3, m_2), then the code encapsulated by m_2 was duplicated in m_1 and m_3.

We executed the aforementioned heuristic comparing each pair of consecutive revisions, covering the entire commit history of each repository. We only discarded

Table 2.1: Studied repositories

Repository	Description
android	GitHub Android App
android-async-http	An Asynchronous HTTP Library for Android
clojure	The Clojure programming language
facebook-android-sdk	SDK to integrate Android apps with Facebook Platform
jsoup	Java HTML Parser
junit	A testing framework for Java.
picasso	A powerful image downloading and caching API for Android
spark	A Sinatra inspired framework for Java
storm	Distributed and fault-tolerant realtime computation system
yuicompressor	A JavaScript compressor

commits that merge changes from different branches, because otherwise we would collect duplicate information. For example, suppose a developer applies a refactoring r_1 in a branch b_1. When the developer merges b_1 with b_2, the merge commit will apply the refactoring in b_2 too. We also modified the source code of the original tool to automate the navigation in the commit graph of a repository. Finally, we modified its source code to prevent it from compiling the entire project during its analysis. This optimization allowed us to reduce the execution time significantly and made it possible to use it in large scale.

2.2.3 Counting the method invocations

To detect the number of invocations of the methods of interest, we developed a tool based on the JDT Core API, which is the API used by the Eclipse IDE to parse and manipulate Java code. This API allows us to analyze the source code at the Abstract Syntactic Tree (AST) level, making it possible to find every method invocation in the code. In addition, the JDT Core API provides a resolution mechanism to bind a method invocation to its declaration.

Taking advantage of information provided by such API, we searched for invocations of each method of interest in each revision of the system. Specifically, the methods of interest were the ones that were extracted from another method at some point in the history, i.e,. they were created by applying an Extract Method refactoring. For that reason, invocations of methods defined externally to the project, i.e., methods defined in third party APIs, are not relevant to the analysis. This makes it possible to use JDT Core API even in the absence of all build dependencies of the project. By the end of this step, we recorded the number of invocations of each extracted method, along with the methods that invoked them. This information is recorded for every revision after the extracted method is introduced.

2.3 Results

In this section we present and discuss the results of the analysis. In total, we analyzed 10,931 commits from 10 repositories, as detailed in Table 2.2. Additionally, Table 2.2 presents the number of distinct methods declared along the history of each repository and the number of methods created by applying Extract Method. Interestingly, 649 methods out of 30,161 are extracted, which corresponds to 2.2%. Moreover, every repository contains extracted methods. It is important to note that the number of methods analyzed is always greater than or equal to the total number of methods in

the last version of a system. That is because throughout the evolution of the system, methods are created and removed. In fact, out of 30,161 methods analyzed, only 17,707 (58.7%) of them exist in the last version of their respective projects. In the next sections we discuss each research question in detail, taking into consideration the set of 649 extracted methods.

2.3.1 RQ1: How often is Extract Method motivated by code reuse?

To answer this first research question, we searched for the following scenario: in a single commit, a developer extracts a method and invokes it two or more times in the system. We assume that, in such cases, the developer extracted the method to reuse it. The second column of Table 2.3 presents the number of extracted methods that are reused (i.e., two or more invocations). When analyzing each repository individually, we note that the lowest percentage of reused methods is 41.1%, for *junit* repository, and the highest percentage is 68.6%, for *jsoup*. Moreover, for 7 out of 10 repositories, the percentage of reused methods is above 50%. Last, 369 out of 649 methods fall in this scenario (56.9%), which yields the following answer to RQ1: **in 56.9% of the cases developers apply Extract Method to reuse code**.

Table 2.2: Statistics from the analyzed repositories

Repository	Commits	Analyzed Methods	Extracted Methods
android	2,351	3,386	100 (3.0%)
androidasynchttp	557	726	34 (4.7%)
clojure	2,629	6,276	122 (1.9%)
facebook-android-sdk	451	4,840	144 (3.0%)
jsoup	711	1,760	35 (2.0%)
junit	1,611	5,822	107 (1.8%)
picasso	461	1,357	41 (3.0%)
spark	281	738	19 (2.6%)
storm	1,491	4,987	38 (0.8%)
yuicompressor	388	269	9 (3.3%)
Total	10,931	30,161	649 (2.2%)

2.3.2 RQ2: How often is Extract Method motivated by removing duplicate code?

In this second research question, we investigate a more specific scenario: in a single commit, a developer extracts a method from two or more places of the system. As a consequence, duplicate code in the original methods is removed and replaced by invocations of the extracted method. The last column of Table 2.3 shows how often such scenario occurs in total and in each repository. We found at least one case of removal of duplicated code in each repository, except for the *storm* project. The project with more occurrences is *junit*, with 11 instances, which represents 10.3% of all Extract Method instances. It is worth noting that such duplicate code removal scenario is a specialization of the scenario studied in RQ1. That is, out of 369 methods extracted with more than one invocation, 51 were characterized as duplicate code removal. In summary, we found the following answer to RQ2: **in 7.9% of the cases developers apply Extract Method to eliminate duplicate code in the system.** Although this case is less frequent than the prior, it could be observed in almost all repositories.

2.3.3 RQ3: How often does Extract Method favor code reuse?

While in the previous research questions we analyzed scenarios in which a method is extracted and reused in the same commit, this time we investigate whether extracted

Table 2.3: Extract Method instances related to code reuse

Repository	Extracted methods	Invocations ≥ 2	Duplicated code
android	100	60 (60.0%)	10 (10.0%)
androidasynchttp	34	15 (44.1%)	3 (8.8%)
clojure	122	72 (59.0%)	11 (9.0%)
facebookandroidsdk	144	95 (66.0%)	9 (6.3%)
jsoup	35	24 (68.6%)	1 (2.9%)
junit	107	44 (41.1%)	11 (10.3%)
picasso	41	26 (63.4%)	4 (9.8%)
spark	19	10 (52.6%)	1 (5.3%)
storm	38	18 (47.4%)	0 (0.0%)
yuicompressor	9	5 (55.6%)	1 (11.1%)
Total	649	369 (56.9%)	51 (7.9%)

Table 2.4: Extracted methods that were reused along the version history

Repository	Reused immediately	Reused later	Never reused
android	60 (60.0%)	3 (3.0%)	37 (37.0%)
android-async-http	15 (44.1%)	5 (14.7%)	14 (41.2%)
clojure	72 (59.0%)	5 (4.1%)	45 (36.9%)
facebook-android-sdk	95 (66.0%)	2 (1.4%)	47 (32.6%)
jsoup	24 (68.6%)	2 (5.7%)	9 (25.7%)
junit	44 (41.1%)	9 (8.4%)	54 (50.5%)
picasso	26 (63.4%)	2 (4.9%)	13 (31.7%)
spark	10 (52.6%)	2 (10.5%)	7 (36.8%)
storm	18 (47.4%)	1 (2.6%)	19 (50.0%)
yuicompressor	5 (55.6%)	0 (0.0%)	4 (44.4%)
Total	369 (56.9%)	31 (4.8%)	249 (38.4%)

methods with a single invocation are eventually invoked by two or more methods throughout the evolution of the system. Table 2.4 presents the number of methods that fall into this scenario (column 3), in contrast to the ones that already started with more than one invocation (column 2) and those that are never reused by other methods along the version history (column 4). The overall result shows that the extracted methods are reused later in 31 out of 649 cases (4.8%). This scenario is observed in almost all repositories, except for the *yuicompressor* (possibly because of the small number of extracted methods identified in it). Table 2.4 also shows that in 249 out of 649 cases (38.4%) there is only one invocation of the extracted method throughout the entire history of the project, i.e., the method is never reused. These results suggest the following answer to RQ3: **in 4.8% of the cases that developers apply Extract Method, there is no initial intention to reuse code, but a reuse opportunity appeared later.** Although not frequent, this case is also consistently observed.

2.4 Reuse patterns related to Extract Method

In the light of the results presented in previous sections, we propose three distinct patterns of code reuse associated with Extract Method refactoring:

Duplication: In this scenario, a developer identifies duplicated code in two or more places in the system and applies Extract Method to fix the issue, replacing the

duplicated code with a method invocation. Therefore, the refactoring is applied to fix the *Duplicated Code* bad smell, which is a classical motivation for Extract Method, as documented in the refactoring literature [Fowler, 1999]. Figure 2.2 presents an example of this scenario in *junit* project. A developer extracted the method `createLoader`, encapsulating a piece of duplicated code in methods `load` (lines 8–9) and `reload` (lines 12–13). These lines of code, which were responsible for instantiating the `TestCaseClassLoader` class, are then replaced by the invocation of `createLoader`. It is interesting to note that a new statement was also introduced in `createLoader` (line 18), which would result in one more duplicate line of code if the method was not extracted.

Figure 2.2: Example of the *Duplication* scenario

Immediate Reuse: In this scenario, a developer identifies the opportunity to reuse existing code when modifying the system, either to introduce new functionality or fix a problem. Extract Method is then applied and the new method is invoked in the new feature. Figure 2.3 presents an example of this scenario in the *storm* project. A developer introduced a new method `tryPublish`, whose behavior was very similar to the existing `publish` method. To do this, an overloaded method `publish(Object, boolean)` was extracted from `publish(Object)` and reused in `tryPublish`. Note that the boolean parameter introduced in the extracted method allows a slight variation in the logic,

enabling its reuse in both cases.

Figure 2.3: Example of the *Immediate Reuse* scenario

Later Reuse: In this scenario, a developer applies the Extract Method refactoring and the extracted method is invoked in only one location. However, in future code changes, a developer finds the opportunity to reuse that method. Therefore, there is no evidence that the initial motivation for the refactoring is code reuse, but it favors reuse later, possibly as a side effect. Figure 2.4 presents an example of this scenario in the *clojure* project. A developer extracted the `isMacro` method from `analyzeSeq`, encapsulating the code between lines 2799–2805. In this case, the method was probably refactored to improve readability, since much of the logic in `analyzeSeq` was intended to determine whether a certain object is a macro, which became clearly indicated by the name of the extracted method. However, in a later modification, the same logic was required and another invocation of the `isMacro` method was introduced.

Figure 2.5 shows the proportion of the three observed reuse pattern in the studied repositories. Overall, the sum of the three scenarios represents 61.6% of Extract Method instances, which indicates that code reuse is an important driver for refactor-

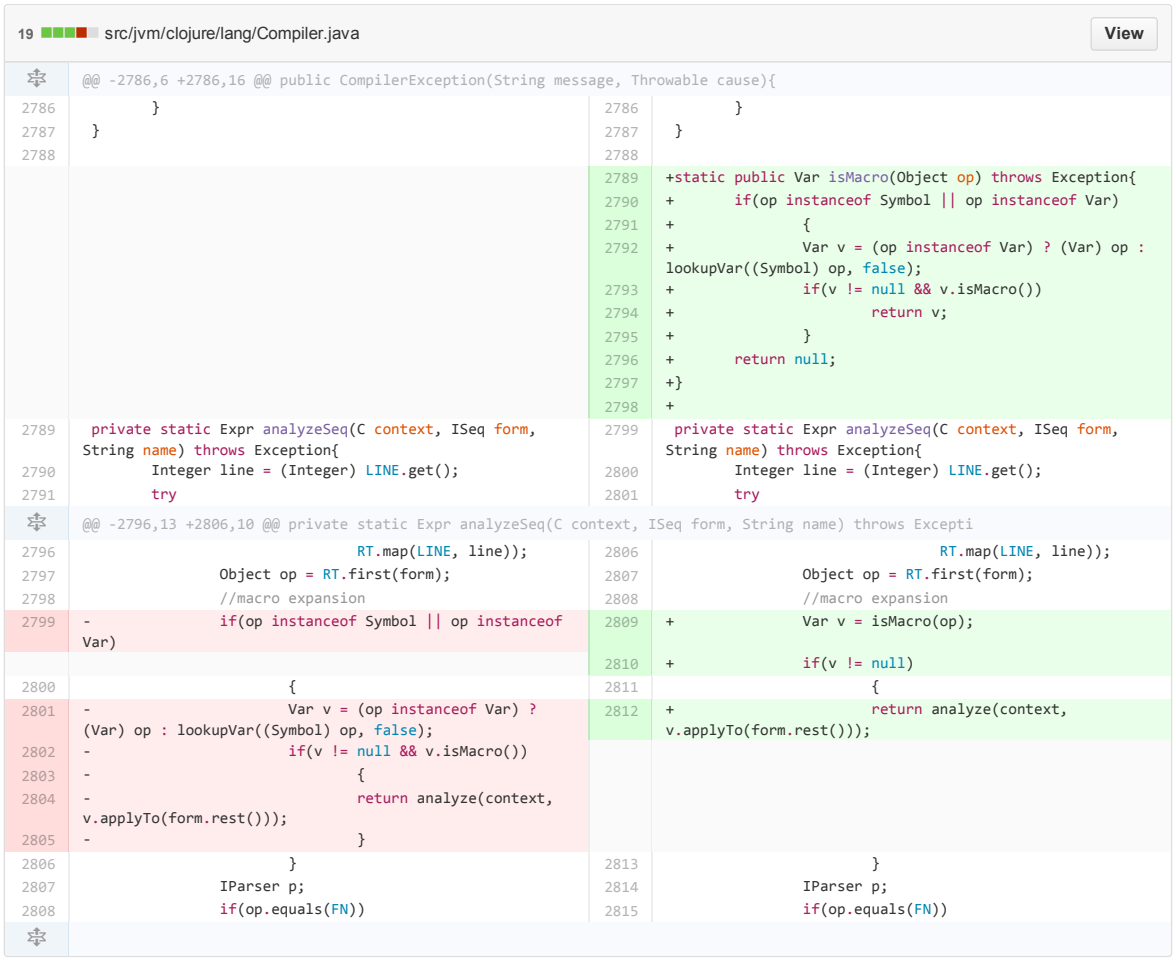


Figure 2.4: Example of the *Later Reuse* scenario

ing effort. The complementary 38.4% are those cases where the extracted methods are never reused.

2.5 Threats to Validity

There are at least two threats to the internal validity of the study:

- The accuracy of the results depends on how accurate the refactoring detection tool is. Although the authors reported a precision of 96.4% [Tsantalis et al., 2013], precision could be different under the circumstances of this study. To mitigate such a threat, we manually inspected a sample of the Extract Method instances we found. We choose 50 instances at random for manual inspection and assessed that 4 of them were false positives (92.0% of precision). Additionally, there is the possibility that the tool does not detect all Extract Method instances

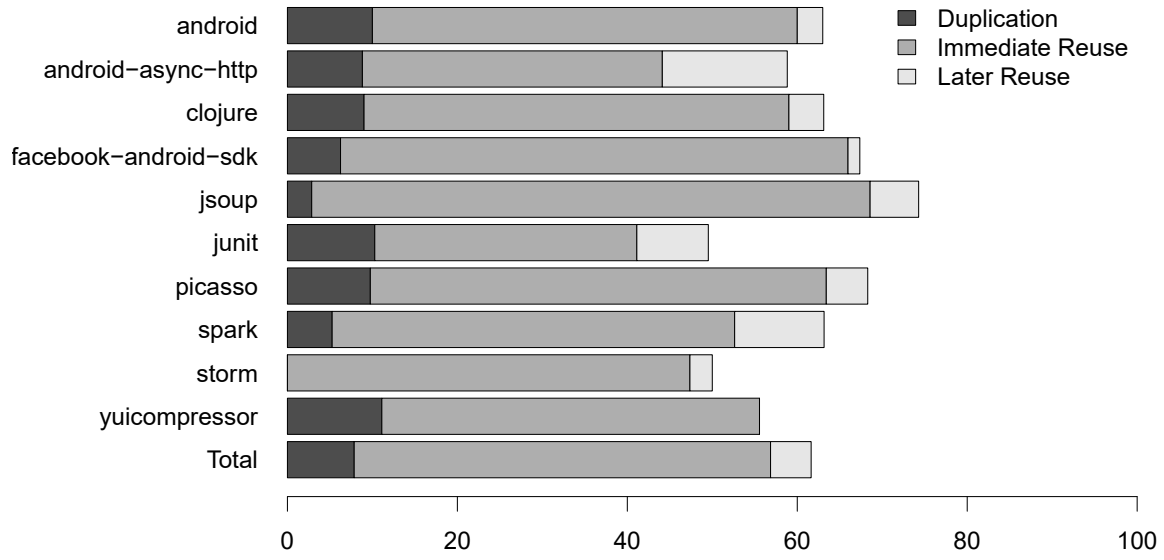


Figure 2.5: Proportion of observed reuse patterns

applied (false negatives). Unfortunately, it is not feasible to assess the tool recall by manually inspecting the code changes.

- The validity of the results depends on the correctness of the method invocation analysis tooling developed specifically for this study. In fact, we believe that not every method invocation is identified for three reasons: (i) the methodology uses only static analysis and it is not able to identify method invocations relying on meta-programming features of the Java language, (ii) the JDT Core API is not able to resolve method invocation bindings in 100% of the cases in the absence of the complete compilation dependencies, and (iii) extracted methods that are moved or renamed in a subsequent revision are not accounted for in the *later reuse* scenario. Nevertheless, such issues would only result in an underestimation of the number of invocations. Therefore, the existence of false negatives would not undermine the finding that code reuse is a major motivation for Extract Method.

We should also mention the threat of external validity, especially considering that the selected systems share the following characteristics:

- Only Java systems were considered, due to limitations of the analysis tools. The results may differ if we consider other programming languages.

- The systems analyzed are all open source systems from a single source (GitHub). Results may differ for proprietary systems developed in a different context.

2.6 Conclusion

In this study we analyzed 10,931 commits from 10 open-source Java projects to investigate the relationship between Extract Method refactoring and code reuse. We found evidence that 56.9% of the Extract Method instances are motivated by code reuse. Specifically, this occurs in two distinct scenarios: **Duplication** (7.9% of cases) and **Immediate Reuse** (49.0% of cases). In addition, we found a third scenario, **Later Reuse**, in which Extract Method refactoring favored long-term code reuse, even though there was no indication that this was the initial intention (4.8% of cases). All three scenarios were found in virtually all systems analyzed in similar proportion.

The results obtained indicate that it is incorrect to assume that Extract Method refactoring is always associated with the resolution of an existing bad smell, such as *Long Method* or *Duplicated Code*. Approximately half of the cases analyzed fall into the **Immediate Reuse** scenario, in which a developer refactors existing code to enable code reuse in new code he is working on when modifying the system. From another perspective, this also suggests that developers avoid introducing duplication when fixing a defect or implementing new functionality.

In particular, this observation has two implications for research on Extract Method recommendation approaches. First, regardless of the recommendation heuristic used, certain decisions made by a developer take into consideration the feature she will introduce to the system. Thus, it is quite challenging for a recommendation tool that relies only on the existing code to suggest an appropriate recommendation in these cases. Second, given that code reuse is a frequent scenario, new recommendation approaches could be explored, for example, suggesting the extraction of code snippets that are likely to be reused in the project.

Chapter 3

An Empirical Study on Refactoring Motivations

Refactoring is a widespread practice that helps developers to improve the maintainability and readability of their code. However, there is a limited number of studies empirically investigating the actual motivations behind specific refactoring operations applied by developers. To fill this gap, we monitored Java projects hosted on GitHub to detect recently applied refactorings, and asked the developers to explain the reasons behind their decision to refactor the code. By applying thematic analysis on the collected responses, we compiled a catalogue of 44 distinct motivations for 12 well-known refactoring types. We found that refactoring activity is mainly driven by changes in the requirements and much less by code smells. Extract Method is the most versatile refactoring operation serving 11 different purposes. Additionally, we found evidence that the IDE used by the developers affects the adoption of automated refactoring tools.

3.1 Introduction

Refactoring catalogues, such as the one proposed by Fowler [1999], define a name and describe the mechanics of each refactoring, and usually discuss a *motivation* for applying it. In many cases, such motivation is associated to the resolution of a code smell. For example, Extract Method is recommended to decompose a large and complex method or to eliminate code duplication. As a second example, Move Method is associated to smells like Feature Envy and Shotgun Surgery [Fowler, 1999].

However, there is a limited number of studies investigating the real motivations driving the refactoring practice based on interviews and feedback from actual developers. Kim et al. [2014] explicitly asked developers "in which situations do you perform

refactorings?" and recorded 10 code symptoms that motivate developers to initiate refactoring. Wang [2009] interviewed professional software developers about the major factors that motivate their refactoring activities and recorded human and social factors affecting the refactoring practice. However, both studies were based on general-purpose surveys or interviews that were not focusing the discussion on specific refactoring operations applied by the developers, but rather on general opinions about the practice of refactoring.

Contribution: To the best of our knowledge, this is the first study investigating *the motivations behind refactoring based on the actual explanations of developers on specific refactorings they have recently applied.* To fill this gap on the empirical research in this area, we report a large scale study centered on 463 refactorings identified in 222 commits from 124 popular, Java-based projects hosted on GitHub. In this study, we asked the developers who actually performed these refactorings to explain the reasons behind their decision to refactor the code. Next, by applying thematic analysis [Cruzes and Dyba, 2011], we categorized their responses into different themes of motivations. Another contribution of this study is that we make publicly available[1] the data collected and the tools used to enable the replication of our findings and facilitate future research on refactoring.

Relevance to existing research: The results of this empirical study are important for two main reasons. First, having a list of motivations driving the application of refactorings can help researchers and practitioners to infer rules for the automatic detection of these motivations when analyzing the commit history of a project. Recent research has devised techniques to help in understanding better the practice of code evolution by identifying frequent code change patterns from a fine-grained sequence of code changes [Negara et al., 2014], isolating non-essential changes in commits [Kawrykow and Robillard, 2011], and untangling commits with bundled changes (e.g., bug fix and refactoring) [Dias et al., 2015]. In addition, we have empirical evidence that developers tend to interleave refactoring with other types of programming activity [Murphy-Hill et al., 2012], i.e., developers tend to *floss refactor*. Therefore, *knowing the motivation behind a refactoring can help us to understand better other related changes in a commit.* In fact, in this study we found several cases where developers extract methods in order to make easier the implementation of a feature or a bug fix.

Second, having a list of motivations driving the application of refactorings can help researchers and practitioners to develop refactoring recommendation systems tailored to the actual needs and practices of the developers. Refactoring serves multiple

[1] http://aserg-ufmg.github.io/why-we-refactor

purposes [Fowler, 1999], such as improving the design, understanding the code [Bois et al., 2005], finding bugs, and improving productivity. However, research on refactoring recommendation systems [Bavota et al., 2014a] has mostly focused on the design improvement aspect of refactoring by proposing solutions oriented to code smell resolution. For example, most refactoring recommenders have been designed based on the concept that developers extract methods either to eliminate code duplication, or decompose long methods [Tsantalis and Chatzigeorgiou, 2011; Silva et al., 2014; Tairas and Gray, 2012; Hotta et al., 2012; Meng et al., 2015; Tsantalis et al., 2015]. In this study, we found 11 different reasons behind the application of Extract Method refactorings. Each motivation requires a different strategy in order to detect suitable refactoring opportunities. Building refactoring recommendation systems tailored to the real needs of developers will help to promote more effectively the practice of refactoring to the developers, by recommending refactorings helping to solve the actual problems they are facing in maintenance tasks.

3.2 Related Work

Refactoring is recognized as a fundamental practice to maintain a healthy code base [Fowler, 1999; Beck, 2000; Opdyke, 1992; Mens and Tourwe, 2004]. For this reason, vast empirical research was recently conducted to extend our knowledge on this practice.

Studies on refactoring practices: Murphy et al. [2006] record the first results on refactoring usage, collected using the Mylar Monitor, a standalone framework that collects and reports trace information about a user's activity in Eclipse. Murphy-Hill et al. [2012] rely on multiple data sources to reveal how developers practice refactoring activities. They investigate nine hypotheses about refactoring usage and conclude for instance that commit messages do not reliably indicate the presence of refactoring, that programmers usually perform several refactorings within a short time period, and that 90% of refactorings are performed manually. Negara et al. [2013] provide a detailed breakdown on the manual and automated usage of refactoring, using a large corpus of refactoring instances detected using an algorithm that infers refactorings from fine-grained code edits. As their central findings, they report that more than half of the refactorings are performed manually and that 30% of the applied refactorings do not reach the version control system.

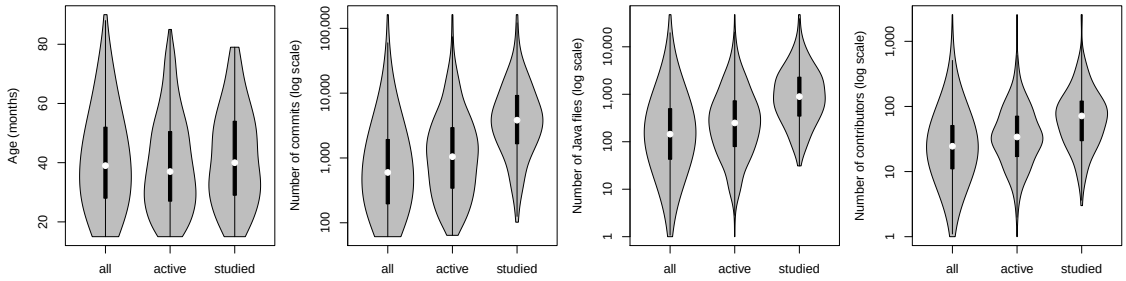
Studies based on surveys & interviews: Kim et al. [2012, 2014] present a field study of refactoring benefits and challenges in a major software organization. They

conduct a survey with developers at Microsoft regarding the cost and risks of refactoring in general, and the adequacy of refactoring tool support, and find that the developers put less emphasis on the behavior preserving requirement of refactoring definitions. They also interview a designated Windows refactoring team to get insights into how system-wide refactoring was carried out, and report that the binary modules refactored by the refactoring team had a significant reduction in the number of inter-module dependencies and post-release defects. Wang [2009] interviews 10 professional software developers and finds a list of intrinsic (i.e., self-motivated) and external (i.e., forced by peers or the management) factors motivating refactoring activity.

Studies on refactoring tools: Vakilian et al. [2012] reveal many factors that affect the appropriate and inappropriate use of refactoring tools. They show for example that novice developers may underuse some refactoring tools due to lack of awareness. Hill and Black [2008] investigate the barriers in using the tool support provided for the Extract Method refactoring [Fowler, 1999]. They report that users frequently made mistakes in selecting the code fragment they want to extract and that error messages from refactoring engines are hard to understand. Murphy-Hill et al. [2012] show that 90% of configuration defaults in refactoring tools are not changed by the developers. As a practical consequence of these studies, refactoring recommendation systems [Bavota et al., 2014a] have been proposed to foster the use of refactoring tools and leverage the benefits of refactoring, by alerting developers about potential refactoring opportunities [Tsantalis and Chatzigeorgiou, 2009, 2011; Bavota et al., 2011; Sales et al., 2013; Bavota et al., 2014b; Silva et al., 2014].

Studies on refactoring risks: Kim et al. [2011] show that there is an increase in the number of bug fixes after API-level refactorings. Rachatasumrit and Kim [2012] show that refactorings are involved in almost half of the failed test cases. Weißgerber and Diehl [2006a] show that refactorings are sometimes followed by an increasing ratio of bug reports.

However, existing studies on refactoring practices do not investigate in-depth the *motivation* behind specific refactoring types, i.e., why developers decide to perform a certain refactoring operation. For instance, Kim et al. [2014] do not differentiate the motivations between different refactoring types, and Wang [2009] does not focus on the technical motivations, but rather on the human and social factors affecting the refactoring practice in general. The only exception is a study conducted by Tsantalis et al. [2013], in which the authors themselves manually inspected the relevant source code before and after the application of a refactoring with a text diff tool, to reveal possible motivations for the applied refactorings. Because they conducted this study without asking the opinion of the developers who actually performed the refactorings,



(a) (b) (c) (d)

Figure 3.1: Distribution of (a) age, (b) commits, (c) Java source files, and (d) contributors of repositories

the interpretation of the motivation can be considered subjective and biased by the opinions and perspectives of the authors. In addition, the manual inspection of source code changes is a rather tedious and error-prone task that could affect the correctness of their findings. Finally, the examined refactorings were collected from the history of only three open source projects, which were libraries or frameworks. This is a threat to the external validity of the study limiting the ability to generalize its findings beyond the characteristics of the selected projects. In this study, we collected refactorings from 124 different projects, and asked the developers who actually performed these refactorings to explain the reasons behind their decision to refactor the code.

3.3 Research Methodology

In this section we describe our methodology. First, we detail how we selected the studied repositories (Section 3.3.1). Second, we describe RefactoringMiner, the tool we used to find refactorings in version histories (Section 3.3.2). Third, we discuss the study's design (Section 3.3.3). Last, we present statistics of the collected data (Section 3.3.4).

3.3.1 Selection of GitHub Repositories

First, we selected the top 1,000 Java repositories ordered by popularity in GitHub (stargazers count) that are not forks. From this initial list, we discarded the lower quartile ordered by number of commits, to focus the study on repositories with more maintenance activity. The final selection consists of 748 repositories, including well-known projects, like JetBrains/intellij-community, Apache/cassandra, Elastic/elasticsearch, Gwtproject/gwt, and Spring-projects/spring-framework.

Figure 3.1 shows violin plots [Hintze and Nelson, 1998] with the distribution of age (in months), number of commits, size (number of `*.java` files), and number of contributors of the selected repositories. We provide plots for all 748 systems (labeled as *all*), for the 471 systems (63%) with at least one commit during the study period (labeled as *active*), and for the 124 systems (17%) effectively analyzed in the study (labeled as *studied*), which correspond to the repositories with at least one refactoring detected in the commits during the study period (61 days), along with answers from the developers to our questions about the motivation behind the detected refactorings. We can observe in Figure 3.1 that the *active* systems tend to have a higher number of commits, source files, and contributors than the initially selected systems (*all*). The same holds when comparing the *studied* systems with the *active* systems. These observations are statistically confirmed by applying the one-tailed variant of the Mann-Whitney U test.

3.3.2 RefactoringMiner Tool

In the study, we search for refactorings performed in the version history of the selected GitHub repositories by analyzing the differences between the source code of two revisions. For this purpose, we use a refactoring detection tool proposed in a previous work [Tsantalis et al., 2013]. The tool, named RefactoringMiner 0.1 in this study, implements a lightweight version of the UMLDiff [Xing and Stroulia, 2005] algorithm for differencing object-oriented models. This algorithm is used to infer the set of classes, methods, and fields added, deleted or moved between successive code revisions. After executing this algorithm, a set of rules is used to identify different types of refactorings. Unlike other existing refactoring detection tools, such as Ref-Finder [Kim et al., 2010] and JDevAn [Xing and Stroulia, 2008], RefactoringMiner provides an API and can be used as an external library independently from an IDE, while Ref-Finder and JDevAn can be executed only within the Eclipse IDE. The strong dependency of Ref-Finder and JDevAn to the Eclipse IDE prevented us from using these tools in our study, since as it will be explained in Section 3.3.3, our study required a high degree of automation, and this could be achieved only by being able to use RefactoringMiner programmatically through its API.

In the study, we analyze 12 well-known refactoring types detected by RefactoringMiner, as listed in the first column of Table 3.2. The detection of Rename Class/Method/Field refactorings was not supported by RefactoringMiner at the time of this study. Typically, these refactorings are performed to give a more meaningful name to the renamed code element. Previous studies show that they are usually per-

formed automatically, using the refactoring tool support of popular IDEs [Murphy-Hill et al., 2012; Negara et al., 2013].

3.3.2.1 RefactoringMiner Precision and Recall

As we rely on RefactoringMiner to find refactorings performed in the version history of software repositories, it is important to estimate its recall and precision. For this reason, we evaluated RefactoringMiner using the dataset reported in a study by Chaparro et al. [2014]. This dataset includes a list of refactorings performed by two Ph.D. students on two software systems (ArgoUML and aTunes) along with the source code before and after the modifications. There are 173 refactoring instances in total, from which we selected all 120 instances corresponding to 8 of the refactoring types considered in this study (8×15 instances per type). The dataset does not contain instances of Extract Superclass/Interface, Move Class, and Rename Package refactorings. We compared the list of refactorings detected by RefactoringMiner with the known refactorings in those systems to obtain the results of Table 3.1, which presents the number of true positives (TP), the number of false positives (FP), the number of false negatives (FN), the recall and precision for each refactoring type. In total, there are 111 true positives (i.e., existing refactoring instances that were correctly detected) and 9 false negatives (i.e., existing refactoring instances that were not detected), which yield a fairly high recall of 0.93. Besides, there are 2 false positives (i.e., incorrectly detected refactoring instances), which yield a precision of 0.98. The lowest observed recall is for Pull Up Method (0.80), while the lowest observed precision is for Extract Method (0.88).

In conclusion, the accuracy of RefactoringMiner is at acceptable levels, since Ref-Finder (the current state-of-the-art refactoring reconstruction tool) has an overall precision of 79% according to the experiments conducted by its own authors [Prete et al., 2010], while an independent study by Soares et al. [2013] has shown an overall precision of 35% and an overall recall of 24% for Ref-Finder.

3.3.3 Study Design

During 61 days (between June 8^{th} and August 7^{th} 2015), we monitored all selected repositories to detect refactorings. We built an automated system that periodically fetches commits from each remote repository to a local copy (using the `git fetch` operation). Next, the system iterates through each commit and executes RefactoringMiner to find refactorings and store them in a relational database.

As in a previous study [Tsantalis et al., 2013], we compare each examined commit with its parent commit in the directed acyclic graph (DAG) that models the commit

Table 3.1: RefactoringMiner Recall and Precision

Refactoring	TP	FP	FN	Recall	Precision
Extract Method	15	2	0	1.00	0.88
Inline Method	13	0	2	0.87	1.00
Pull Up Attribute	15	0	0	1.00	1.00
Pull Up Method	12	0	3	0.80	1.00
Push Down Attribute	15	0	0	1.00	1.00
Push Down Method	13	0	2	0.87	1.00
Move Attribute	15	0	0	1.00	1.00
Move Method	13	0	2	0.87	1.00
Total	111	2	9	0.93	0.98

history in git-based version control repositories. Furthermore, we exclude merge commits from our analysis to avoid the duplicate report of refactorings. Suppose that commit C_M merges two branches containing commits C_A and C_B, respectively. Suppose also that a refactoring *ref* is performed in C_A, and therefore detected when we compare C_A with its parent commit. Because the effects of *ref* are present in the code that resulted from C_M, *ref* would be detected again if we compared C_M with C_B. Although a developer may introduce arbitrary changes in C_M (that are not in C_A or C_B), C_M is unlikely to contain additional refactorings. Thus, we assume that discarding merge commits from our analysis does not lead to significant refactoring loss.

On each working day, we retrieved the recent refactorings from the database to perform a manual inspection, using a web interface we built to aid this task. In this step, we filter out false positives by analyzing the source code diff of the commit. In this way, we avoid asking developers about false refactorings. Additionally, we also marked commits that already include an explanation for the detected refactoring in the commit description, to avoid asking an unnecessary question. For instance, in one of the analyzed commits we found several methods extracted from a method named `onCreate`, and the commit description was:

"Refactored `AIMSICDDbAdapter::DbHelper#onCreate` for easier reading"

Thus, it is clear that the intention of the refactoring was to improve readability by decomposing method `onCreate`. Therefore, it would be unnecessary and inconvenient to ask the developer.

This process was repeated daily, to detect the refactorings as soon as possible

after their application in the examined systems. In this way, we managed to ask the developers shortly after they perform a refactoring, to increase the chances of receiving an accurate response. We send at most one email to a given developer, i.e., if we detect a refactoring by a developer who has been already contacted before, we do not contact her again, to avoid the perception of our messages as spam email. The email addresses of the developers were retrieved from the commit metadata.

In each email, we describe the detected refactoring(s) and provide a GitHub URL for the commit where the refactoring(s) is(are) detected. In the email, we asked two questions:

1. Could you describe why did you perform the listed refactoring(s)?

2. Did you perform the refactoring(s) using the automated refactoring support of your IDE?

With the first question, our goal is to reveal the actual motivation behind real refactorings instances. With the second question, we intend to collect data about the adequacy and usage of refactoring tools, previously investigated in other empirical studies [Murphy-Hill et al., 2012; Negara et al., 2013]. In this way, we can check whether the findings of these studies are reproduced in our study. We should clarify that by "automated refactoring" we refer to user actions that trigger the refactoring engine of an IDE by any means (e.g., through the IDE menus, keyboard shortcuts, or drag-and-drop of source code elements).

During the study period, we sent 465 emails and received 195 responses, achieving a response rate of 41.9%. Each response corresponds to a distinct developer and commit. The achieved response rate is significantly larger than the typical 5% rate found in questionnaire-based software engineering surveys [Singer et al., 2008]. This can be attributed to the *firehouse interview* [Hill et al., 2015] nature of our approach, in which developers provide their feedback shortly after performing a refactoring and have fresh in their memory the motivation behind it. Additionally, we included in our analysis all 27 commits whose description already explained the reasons for the applied refactorings, totaling a set of 222 commits. This set of commits covers 124 different projects and contains 463 refactoring instances in total.

After collecting all responses, we analyzed the answers using thematic analysis [Cruzes and Dyba, 2011], a technique for identifying and recording patterns (or "themes") within a collection of documents. Thematic analysis involves the following steps: (1) initial reading of the developer responses, (2) generating initial codes for each response, (3) searching for themes among codes, (4) reviewing the themes to find

opportunities for merging, and (5) defining and naming the final themes. These five steps were performed independently by two validators, with the support of a simple web interface we built to allow the analysis and tagging of the detected refactorings. At the time of the study, the first validator (Validator#1) had 3 years of research experience on refactoring, while the second validator (Validator#2) had over 8 years of research experience on refactoring.

After the generation of themes from both validators, a meeting was held to assign the final themes. In 155 cases (58%), both validators suggested semantically equivalent themes that were rephrased and standardized to compose the final set of themes. The refactorings with divergent themes were then discussed by both validators to reach a consensus. In 94 cases (35%), one validator accepted the theme proposed by the other validator. In the remaining 18 cases (7%), the final theme emerged from the discussion and was different from what both validators previously suggested. Figure 3.2 shows a case of an Extract Method refactoring instance that was required to reach a consensus between the validators. The developer who performed the refactoring explained that the reason for the refactoring was to support a new feature that required pagination, as described in the following comment:

"Educational part of PyCharm uses stepic.org courses provider. This server recently decided to use pagination in replies."

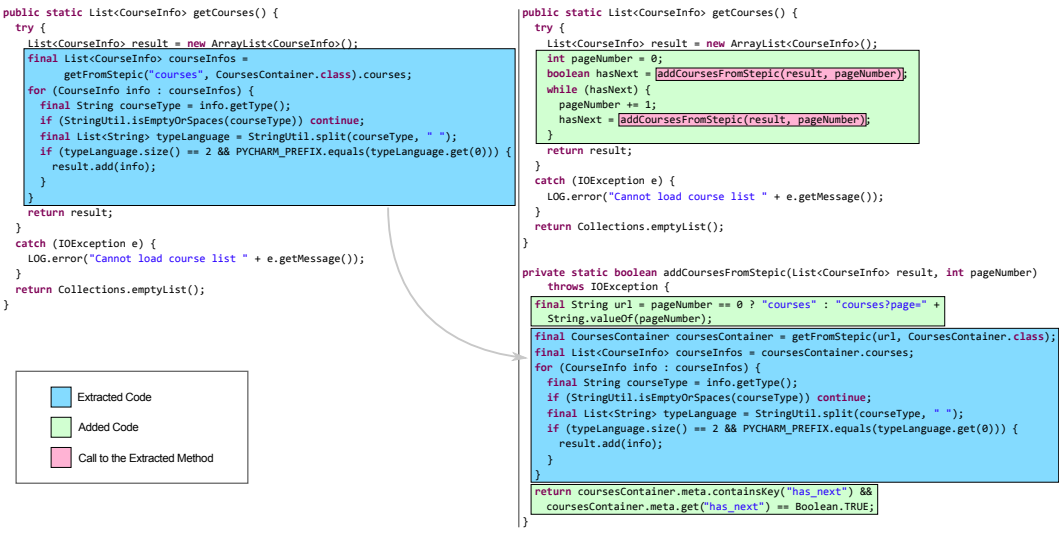


Figure 3.2: Example of Extract Method refactoring that was required to reach a consensus.

By inspecting the source code changes, we can see that a part of the original method `getCourses()` (left-hand side of Figure 3.2) was extracted into method

`addCoursesFromStepic()` (right-hand side of Figure 3.2). After the refactoring, the extracted method is called twice, once before the `while` loop added in the original method, and once inside the `while` loop. For this reason, Validator#1 labeled this case as "Avoid duplication", since the extracted method is reused two times after the refactoring. However, the extracted method contains additional new code to compute properly the URL based on the page number passed as a parameter (first line in the extracted method), and to return a `boolean` indicating if there exists a next page (last line in the extracted method). For this reason, Validator#2 labeled this case as "Facilitate extension", since the extracted method also helps to implement the new pagination requirement. After deliberation, the validators reached a consensus by keeping both theme labels, since the extracted method serves both purposes of reuse and extension.

3.3.4 Examined Refactorings

We monitored 748 Java projects during the study period, and found commits in 471 projects (63%), i.e., 277 projects remained inactive. We also found 285 projects with refactoring activity, as detected by RefactoringMiner (including false positives). In these projects, 2,241 refactoring instances were detected (in 729 commits), and were manually to confirm whether they are indeed true positives.

Table 3.2: Refactoring activity

Refactoring	TP	FP	Precision	Commits	Projects
Extract Method	468	135	0.78	312	136
Move Class	432	512	0.46	85	60
Move Attribute	129	44	0.75	45	38
Rename Package	105	0	1.00	25	24
Move Method	99	48	0.67	40	31
Inline Method	58	67	0.46	44	36
Pull Up Method	33	1	0.97	18	17
Pull Up Attribute	23	1	0.96	11	11
Extract Superclass	22	11	0.67	18	16
Push Down Method	16	1	0.94	6	6
Push Down Attribute	15	1	0.94	7	7
Extract Interface	11	8	0.58	10	9
Total	1,411	830	0.63	539	185

Table 3.2 shows the number of true positives (TP), false positives (FP), and precision (Prec.) of RefactoringMiner by refactoring type, as computed after the manual inspection of the detected refactorings. In general, our tool achieves very high precision for Rename Package (100%), Pull Up/Push Down Attribute/Method refactorings (over 94%), and relatively high precision for Extract Method and Move Attribute refactorings (over 75%), while the precision for Move Method and Extract Superclass is 67%. However, for some refactorings the precision is closer to 50%, namely Extract Interface (58%), Inline Method (46%), and Move Class (46%). We observed several cases of inner classes falsely detected as moved, because their enclosing class was simply renamed. By supporting the detection of Rename Class refactoring, we could improve Move Class precision. It should be emphasized that we asked the developers only about the true positives detected by RefactoringMiner. In comparison to the results presented in Section 3.3.2.1, the precision is lower, because the commits analyzed from GitHub projects may include tangled changes, while the commits analyzed in Section 3.3.2.1 include only refactoring operations. Tangled changes make the detection of refactorings more challenging, thus resulting in more false positives. Finally, Table 3.2 shows the number of distinct commits and projects containing at least one true positive refactoring (539 out of 729 commits and 185 out of 285 projects with detected refactorings contain at least one true positive refactoring).

3.4 Why do Developers Refactor?

In this section, we present the results for the first question answered by the developers, regarding the reasons behind the application of the refactorings we detected. Based on the results of the thematic analysis process (Section 3.3.3), we compile a catalogue of 44 distinct motivations. We dedicate Section 3.4.1 to discuss Extract Method, which is the most frequently occurring refactoring operation in our study, and also the one with the most observed motivations (11). Section 3.4.2 presents the motivations for the remaining refactorings.

3.4.1 Motivations for Extract Method

Table 3.3 describes 11 motivations for Extract Method refactoring and the number of occurrences for each of them. The most frequent motivation is to extract a reusable method (43 instances), which is consistent with the first finding from Study 1 (see Section 2.3.1). In this case, the refactoring is motivated by the immediate reuse of a piece of code in multiple other places, in addition to the place from which it was

Table 3.3: Extract Method motivations

Theme	Description	Occur.
Extract reusable method	Extract a piece of reusable code from a single place and call the extracted method in multiple places.	43
Introduce alternative method signature	Introduce an alternative signature for an existing method (e.g., with additional or different parameters) and make the original method delegate to the extracted one.	25
Decompose method to improve readability	Extract a piece of code having a distinct functionality into a separate method to make the original method easier to understand.	21
Facilitate extension	Extract a piece of code in a new method to facilitate the implementation of a feature or bug fix, by adding extra code either in the extracted method, or in the original method.	15
Remove duplication	Extract a piece of duplicated code from multiple places, and replace the duplicated code instances with calls to the extracted method.	14
Replace Method preserving backward compatibility	Introduce a new method that replaces an existing one to improve its name or remove unused parameters. The original method is preserved for backward compatibility, it is marked as deprecated, and delegates to the extracted one.	6
Improve testability	Extract a piece of code in a separate method to enable its unit testing in isolation from the rest of the original method.	6
Enable overriding	Extract a piece of code in a separate method to enable subclasses override the extracted behavior with more specialized behavior.	4
Enable recursion	Extract a piece of code to make it a recursive method.	2
Introduce factory method	Extract a constructor call (class instance creation) into a separate method.	1
Introduce async operation	Extract a piece of code in a separate method to make it execute in a thread.	1

originally extracted. We often observe a concern among developers to reuse code wherever possible, by extracting pieces of reusable code. This is illustrated by the following comments:

"These refactorings were made because of code reusability. I needed to use the same code in new method. I always try to reuse code, because when there's a lot of code redundancy it gets overwhelmingly more complicated to work with the code in future, because when something change in code that has it's duplicate somewhere, it usually needs to be changed also there."

"The reason for me to do the refactoring was: Don't repeat yourself (DRY)."

Table 3.4: Motivations for Move Class, Attribute, Method (MC, MA, MM), Rename Package (RP) Inline Method (IM), Extract Superclass, Interface (ES, EI), Pull Up Method, Attribute (PUM, PUA), Push Down Attribute, Method (PDA, PDM)

Type	Theme	Description	Occur.
MC	Move class to appropriate container	Move a class to a package that is more functionally or conceptually relevant.	13
MC	Introduce sub-package	Move a group of related classes to a new subpackage.	7
MC	Convert to top-level container	Convert an inner class to a top-level class to broaden its scope.	4
MC	Remove inner classes from deprecated container	Move an inner class out of a class that is marked deprecated or is being removed.	3
MC	Remove from public API	Move a class from a package that contains external API to an internal package, avoiding its unnecessary public exposure.	2
MC	Convert to inner class	Convert a top-level class to an inner class to narrow its scope.	2
MC	Eliminate dependencies	Move a class to another package to eliminate undesired dependencies between modules.	1
MC	Eliminate redundant subpackage	Eliminate a redundant nesting level in the package structure.	1
MC	Backward compatibility	Move a class back to its original package to maintain backward compatibility.	1
MA	Move attribute to appropriate class	Move an attribute to a class that is more functionally or conceptually relevant.	15
MA	Remove duplication	Move similar attributes to another class where a single copy of them can be shared, eliminating the duplication.	4
RP	Improve package name	Rename a package to better represent its purpose.	8
RP	Enforce naming consistency	Rename a package to conform to project's naming conventions.	3
RP	Move package to appropriate container	Move a package to a parent package that is more functionally or conceptually relevant.	2
MM	Move method to appropriate class	Move a method to a class that is more functionally or conceptually relevant.	8
MM	Move method to enable reuse	Move a method to a class that permits its reuse by other classes.	3
MM	Eliminate dependencies	Move a method to eliminate dependencies between classes.	3
MM	Remove duplication	Move similar methods to another class where a single copy of them can be shared, eliminating duplication.	1
MM	Enable overriding	Move a method to permit subclasses to override it.	1
IM	Eliminate unnecessary method	Inline and eliminate a method that is unnecessary or has become too trivial after code changes.	13
IM	Caller becomes trivial	Inline and eliminate a method because its caller method has become too trivial after code changes, so that it can absorb the logic of the inlined method without compromising readability.	2
IM	Improve readability	Inline a method because it is easier to understand the code without the method invocation.	1
ES	Extract common state/behavior	Introduce a new superclass that contains common state or behavior from its subclasses.	7
ES	Eliminate dependencies	Introduce a new superclass that is decoupled from specific dependencies of a subclass.	1
ES	Decompose class	Extract a superclass from a class that holds many responsibilities.	1
PUM	Move up common methods	Move common methods to superclass.	8
PUA	Move up common attributes	Move common attributes to superclass.	7
EI	Facilitate extension	Introduce an interface to enable different behavior.	1
EI	Enable dependency injection	Introduce an interface to facilitate the use of a dependency injection framework.	1
EI	Eliminate dependencies	Introduce an interface to avoid depending on an existing class/interface.	1
PDA	Specialized implementation	Push down an attribute to allow specialization by subclasses.	2
PDA	Eliminate dependencies	Push down attribute to subclass so that the superclass does not depend on a specific type.	1
PDM	Specialized implementation	Push down a method to allow specialization by subclasses.	1

The second most frequent motivation is to introduce an alternative method signature for an existing method (25 instances), e.g., with extra parameters. To achieve that, the body of the existing method is extracted to a new one with an updated signature and additional logic to handle the extended variability. The original method is changed to delegate to the new one, passing some default values for the new parameters. The following comment illustrates this case:

"The extracted method `values(names List<String>, values List<Object>)` *could be of help for some users using Lists instead of arrays, and because the implementation already transformed the provided arrays into Lists internally."*

Decomposing a method for improving readability (21 instances) is the third most frequent motivation. Typically, this corresponds to a *Long Method* code smell [Fowler, 1999], as illustrated in this comment:

"The method was so long that it didn't fit onto the screen anymore, so I moved out parts."

The next two motivations are to facilitate extension (15 instances) and to remove duplication (14 instances). In the first case, a method is decomposed to facilitate the implementation of a new feature or the fix of a bug by adding code either in the extracted or in the original method, as illustrated in this comment:

"I was fixing an exception, in order to do that I had to add the same code to 2 different places. So I extracted initial code, replace duplicate with the extracted method and add the 'fix' to the extracted method."

In the second case (i.e., remove duplication), a piece of duplicated code is extracted from multiple places into a single method, as illustrated in the following comments:

"I refactored shared functionality into a single method."

"I checked how other test methods create testing User objects and noticed that it takes two lines of code that were repeated all over the test class. So I abstracted these two lines of code into a method for better readability and then reused the method in all the places that had the same code."

Finally, two other important motivations are to improve testability (6 instances)

and to replace a method by preserving backward compatibility (6 instances). In the first case, the decomposition enables the developer to test parts of the code in isolation, as illustrated in this comment:

"I wanted to test the part of `authenticate()` *which verifies that a member is element of a set, and that would have been more complex using* `authenticate` *directly."*

In the second case, the goal is to introduce a method having the same functionality with an already existing one, but a different signature (e.g., improved name, or removed unused parameter), and at the same time preserve the public API by making the original method delegate to the new one. This motivation is best illustrated in the following comment:

"I did that refactoring because essentially I wanted to rename the functions involved - you'll see the old functions just forward straight to the new ones. But I didn't just rename because other code in other projects might be referring to the old functions, so they would need to still be present (I guess they should have been marked as @deprecated then, but I was a bit lazy here)."

3.4.2 Motivations for Other Refactorings

Table 3.4 presents the motivations for the remaining refactorings. We found nine different motivations for Move Class. The two most frequent motivations are to move a class to a package that is more functionally or conceptually related to the purpose of the class (13 instances), and to introduce a sub-package (7 instances). The first one is illustrated by the following comment:

"This refactoring was done because common interface for those classes lived in `org.neo4j.kernel.impl.store.`
`record`*, while most of it's implementors lived in* `org.neo4j.`
`kernel.impl.store` *which did not make sense because all of them are actually records."*

For Move Attribute, the most common motivation is also to move the attribute to an appropriate class that is more functionally or conceptually relevant (15 instances), as in the example below:

"In this case, each of these fields was moved as their relevance changed. As `UserService` *already handles the login process, it makes sense that changes to the login process should be encapsulated within* `UserService`*."*

Remove duplication is another motivation for moving an attribute, as illustrated by the following comment:

"The attributes were duplicated, so I moved them to the proper common place."

For Rename Package, the most common motivation is to update the name of a package to better represent its purpose (8 instances), as in the example below:

"This was a simple package rename. `test` seems to fit better than `tests` here as a single test can be executed too."

We found three main reasons for a Move Method refactoring: move a method to an appropriate class (8 instances), move a method to enable reuse (3 instances), and move a method to eliminate dependencies (3 instances). The most frequent motivation for Inline Method is to eliminate an unnecessary or trivial method, as illustrated in the comment:

"Since the method was a one-liner and was used only in one place, inlining it didn't make the code more complex. On the other hand, it allowed to lessen calls to `getVirtualFile()`."

Extract Superclass is usually applied to introduce a new class with state or behavior that can be shared by subclasses (7 instances). Pull Up Method/Attribute is performed to move common code to an existing superclass (8 and 7 instances, respectively). Extract Interface and Push Down Attribute/Method are less popular refactorings and thus their motivations have at most two instances.

3.5 Refactoring Automation

In this section, we discuss the results drawn from the second question answered by the developers, regarding the use (or not) of automatic refactoring tools provided by their IDEs to apply the refactorings we presented. First, in Section 3.5.1, we present how many of the interviewed developers applied the refactoring(s) automatically. We also present which refactoring types are more frequently applied with tool support. In Section 3.5.2, we discuss some insights drawn from developers' answers that explain why refactoring is still applied manually in most of the cases. Last, in Section 3.5.3, we present additional details regarding which IDE developers most often used for refactoring.

3.5.1 Are refactoring tools underused?

Table 3.5 shows the results for this question. 95 developers (55% of valid answers) answered that the refactoring was performed manually without tool support; 66 developers (38%) answered that the refactoring engine of an IDE was used; 13 developers (7%) answered that the refactoring was partially automated. In summary, refactoring is probably more often applied manually than with refactoring tools.

Table 3.5: Manual vs. automated refactoring

Modification	Occurrences
Manual	95
Automated	66
Not answered	48
Partially automated	13

We also counted the percentage of automated refactorings by refactoring type, as presented in Table 3.6. Rename Package is the refactoring most often performed with tool support (58%), followed by Move Class (50%). Three other refactorings are performed automatically in around a quarter of the cases: Extract Method (29%), Move Method (26%), and Move Attribute (24%). Inline Method follows with 18% of automatic applications. Finally, for the remaining refactorings, we do not have a large number of instances to draw safe conclusions (maximum 9 instances), but there is a consistent trend showing that inheritance-related refactorings are mostly manually applied.

3.5.2 Why do developers refactor manually?

29 developers explained in their answers why they did not use a refactoring tool. Table 3.7 shows five distinct themes we identified in these answers.

Lack of trust (10 instances) was the most frequent reason. Some developers do not trust refactoring tools for complex operations that involve code manipulation and only use them for renaming or moving:

"I don't trust the IDE for things like this, and usually lose other comments, notation, spacing from adjacent areas."

"I'd say developers are reluctant to let a tool perform anything but trivial refactorings, such as the ones you picked up on my commit."

Table 3.6: Refactoring automation by type

Refactoring Type	Occurrences	Automated %
Extract Method	118	29
Move Class	36	50
Move Attribute	21	24
Move Method	19	26
Inline Method	17	18
Rename Package	12	58
Extract Superclass	9	11
Pull Up Method	9	11
Pull Up Attribute	7	14
Extract Interface	3	0
Push Down Attribute	3	33
Push Down Method	2	0

Table 3.7: Reasons for not using refactoring tools

Description	Occurrences
The developer does not trust automated support for complex refactorings.	10
Automated refactoring is unnecessary, because the refactoring is trivial and can be manually applied.	8
The required modification is not supported by the IDE.	6
The developer is not familiar with the refactoring capabilities of his/her IDE.	3
The developer did not realize at the moment of the refactoring that he/she could have used refactoring tools.	2

On the other hand, some developers also think that tool support is unnecessary in simple cases (8 instances). Sometimes the operation may involve only local changes and is trivial to do by hand. Thus, calling a special operation to do it is considered unnecessary, as illustrated by this comment:

"Automated refactoring is overkill for moving some private fields."

Additionally, developers also mentioned: lack of tool support for the specific refactoring they were doing (6 instances), not being familiar with refactoring features of the IDE (3 instances), and not realizing they could use refactoring tools at the moment of the refactoring (2 instances).

3.5.3 What IDEs developers use for refactoring?

When answering to our emails, 83 developers spontaneously mentioned which IDE they use. Therefore, we decided to investigate these answers, specially because our study is not dependent on any IDE, and thus differs from previous studies which are usually based only on Eclipse data [Murphy-Hill et al., 2012; Negara et al., 2013]. Table 3.8 shows the most common IDEs mentioned in these answers and the percentage of refactorings performed automatically in these cases. 139 developers (63%) did not explicitly mention an IDE when answering this question. Considering the answers citing an IDE, IntelliJ IDEA is the most popular one. It also has the highest ratio of refactorings performed automatically (71%). Since 11 JetBrains/intellij-community (and related plug-ins) developers answered to our questions, we also investigated the answers separately in two groups, namely answers from IntelliJ IDEA developers and from IntelliJ IDEA users. We observed that the ratio of automated refactorings in both groups is very similar (73% vs. 70%). Therefore, the responses from these 11 IntelliJ IDEA developers do not bias the percentage of automated refactoring reported for IntelliJ IDEA.

Table 3.8: IDE popularity

IDE	Occurrences	Automated %
Editor not mentioned	139	12
IntelliJ IDEA	51	71
Eclipse	18	44
NetBeans	8	50
Android Studio	4	25
Text Editor	2	0

3.6 Discussion

In this section, we discuss the main findings of our study.

Refactoring Motivations: Our study confirms that Extract Method is the "Swiss army knife of refactorings" [Tsantalis et al., 2013]. It is the refactoring with the most motivations (11 in total). In comparison to the study of Tsantalis et al. [2013], there is an overlap in the reported motivation themes for Extract Method. We found some new themes, such as *improve testability* and *enable recursion*, but we did not find any

instances of the themes *encapsulate field* and *hide message chain*, reported by Tsantalis et al. [2013], which are related to code smell resolution. We assume these different themes are due to the nature of the examined projects, since Tsantalis et al. [2013] examined only three libraries and frameworks, while in this study we examined 124 projects from various domains including standalone applications. By comparing to the code symptoms that initiate refactoring reported in the study by Kim et al. [2014], we found the *readability*, *reuse*, *testability*, *duplication*, and *dependency* motivation themes in common.

Most of the refactoring motivations we found have the intention to facilitate or even enable the completion of the maintenance task that the developer is working on. For instance, *extract reusable method*, *introduce alternative method signature*, and *facilitate extension* are among the most frequent motivations, and all of them involve enhancing the functionality of the system. Therefore, Extract Method is a key operation to complete other maintenance tasks, such as adding a feature or fixing a bug. In particular, *extract reusable method* was the most frequent motivation, confirming the findings from Study 1, which shows that code reuse is a major motivation for Extract Method. In contrast, only two out of the 11 motivations we found (*decompose method to improve readability* and *remove duplication*) are targeting code smells. This finding could motivate researchers and tool builders to design refactoring recommendation systems [Tsantalis and Chatzigeorgiou, 2011; Silva et al., 2014; Tairas and Gray, 2012; Hotta et al., 2012; Meng et al., 2015; Tsantalis et al., 2015] that do not focus solely on detecting refactoring opportunities for the sake of code smell resolution, but can support other refactoring motivations as well.

We also observe that developers are seriously concerned about avoiding code duplication, when working on a given maintenance task. They often use refactorings—especially Extract Method—to achieve this goal, as illustrated by the following comments:

"I need to add a check to both the then- and the else-part of an if-statement. This resulted in more duplicated code than I was comfortable with."

"There was already code duplication, but the bug fix required another cut-and-paste, which made it code triplication. That was above my pain level so I decided to group the replicated code out into `bail()`*."*

The other refactorings we analyzed are typically performed to improve the system design. For example, the most common motivation for Move Class, Move Attribute, and Move Method is to reorganize code elements, so that they have a

stronger functional or conceptual relevance.

Automated vs. Manual Refactoring: In a field study with Eclipse users, Negara et al. [2013] report that most refactorings (52%) are manually performed. In our study, involving developers using a wider variety of IDEs, we found that 55% of refactorings are manually performed. However, we also found that IntelliJ IDEA users tend to use more the refactoring tool support than other IDE users. Moreover, the results for automated Extract Method refactorings are very similar in both studies: 28% in our study vs. 30% in their study. While the total percentages of manually performed refactorings are very similar, we should keep in mind that Negara et al. counted simple refactorings, like renamings, which are more often applied with tool support. Compared to the study by Murphy-Hill et al. [2012], where they report that 89% of refactorings are performed manually (considering also renamings), we detected significantly more automated refactorings. We suspect this difference may be due to two reasons. First, automated refactoring tools may have become more popular and reliable over the last years. Second, our study involves developers using a broader range of IDEs, which may also influence how developers use refactoring tool support.

Regarding the reasons for not using automated refactoring, our results are in line with the three main factors found in the study by Murphy-Hill et al. [2012]: *awareness*, *opportunity*, and *trust*. The exception is the argument that tool support is unnecessary in simple cases, which is not closely related to any of the three aforementioned factors. However, the same argument can be observed in the study by Kim et al. [2014], in which some developers mention that they do not feel a great need for automated refactoring tools.

Refactoring Popularity: In this study we detected refactorings in 285 of the monitored repositories in a time window of 61 days. Given that only 471 out of the 748 monitored repositories were active during that period, we found refactoring activity in 60.5% of the repositories with at least one commit. This shows that refactoring is a common practice, especially considering that frequent refactorings such as Rename Class/Method/Field were not considered.

The top-5 most popular refactorings detected in our study are Extract Method, Move Class, Move Attribute, Rename Package, and Move Method. Move Method is the third most popular refactoring in the study by Negara et al. [2013]. The top-2 refactorings in this study (Rename Local Variable and Extract Local Variable) are low-level refactorings, which have not been considered in our study. We focused on high-level refactorings, because they can be motivated by multiple factors.

Using a sample of 40 commits with manual and automated refactorings, Murphy-Hill et al. [2012] report that the two most popular refactorings are Rename Constant and Push Down. However, Push Down refactorings are among the least popular ones in our study. This difference may be related to the number of commits analyzed in the studies (40 vs. 539 commits in our study), and the specialized nature of the software (i.e., the Eclipse IDE) examined by Murphy-Hill et al. [2012].

3.7 Threats to Validity

External Validity: This study is restricted to open source, Java-based, GitHub-hosted projects. Thus, we cannot claim that our findings apply to industrial systems, or to systems implemented in other programming languages. However, we collected responses from 222 developers contributing in 124 different projects, which is one of the largest samples of systems used in refactoring studies.

Internal Validity: First, we use in the study a tool that detects refactorings by comparing two revisions of the code. We evaluated the recall of this tool using a sample of 120 documented refactoring operations. We achieved a recall of 0.93. However, we cannot guarantee a similar recall in the studied GitHub projects, because some commits might contain tangled changes making more difficult to isolate (or untangle [Dias et al., 2015]) the changes related to refactorings. In addition, it is known that this kind of detection approach may miss refactorings that do not reach the version control system (e.g., sequences of overlapping refactorings applied to the same piece of code). We claim this threat should be tolerated in large scale studies, where we cannot assume that the developers would be willing to install an external monitoring tool in their IDEs [Negara et al., 2013]. Furthermore, as we showed in this study, developers nowadays use IDEs from multiple vendors. In order to cover as many IDEs as possible and strengthen the external validity, a study based on monitoring would require to develop a separate version of this tool for each IDE. Second, we cannot claim that the catalogue of motivations we propose is exhaustive. Notably, we have a limited number of motivation themes for less frequent refactoring types, such as Push Down Method/Attribute and Extract Interface. Third, to mitigate inconsistencies in the proposed themes, we rely on an initial classification performed independently by two evaluators, followed by a consensus building process. We also make publicly available the responses collected from the developers and the proposed refactoring motivation themes to provide a means for replication and verification.

3.8 Conclusions

In summary, the main conclusions and lessons learned are:

1. Refactoring activity is mainly driven by changes in the requirements (i.e., new feature and bug fix requests) and much less by code smell resolution. Only 2 out of the 11 motivations for Extract Method were related to fixing existing code smells (*remove duplication*, *decompose method*) covering only 25% (35/138) of the motivation instances. Nevertheless, developers frequently avoid introducing code smells, such as duplicate code.

2. Extract Method is a key operation that serves multiple purposes, specially those related to code reuse and functionality extension. It is also used to improve the testability of code, and deprecate public API methods.

3. The elimination of dependencies is the most common motivation among the *move/abstract* related refactorings.

4. Manual refactoring is still prevalent (55% of the developers refactored manually the code). In particular, inheritance related refactoring tool support seems to be the most under-used (only 10% done automatically), while Move Class and Rename Package are the most trusted refactorings (over 50% done automatically).

5. The IDE plays an important role in the adoption of refactoring tool support. IntelliJ IDEA users perform more automated refactorings (71% done automatically) than Eclipse users (44%) and NetBeans users (50%).

6. Compared to the study by Murphy-Hill et al. [2012], it seems that developers apply more automated refactorings nowadays. Our findings confirm Negara et al. [2013] who collected data only from Eclipse IDE users, but our study covers developers using more IDEs.

Based on our findings, we propose that future research on refactoring recommendation systems should refocus from code-smell-oriented to maintenance-task-oriented solutions. This could be achieved by leveraging feature location [Dit et al., 2013] and requirements tracing to automatically locate the code associated with a feature or bug fix request, or a requirement change, and recommend suitable refactorings that will make easier the completion of the maintenance task. We strongly believe this will boost the adoption of recommendation systems by the developers.

3.9 Artifact Description

The data collected in this study is provided as an artifact, which is publicly available at:

`http://aserg-ufmg.github.io/why-we-refactor`

The dataset includes a list of 539 commits from 185 GitHub-hosted Java projects with 1,411 refactorings identified by the RefactoringMiner tool and confirmed with manual inspection. The detected refactorings cover 12 different types:

1. Extract Method (EM)
2. Move Class (MC)
3. Move Attribute (MA)
4. Rename Package (RP)
5. Move Method (MM)
6. Inline Method (IM)
7. Pull Up Method (UM)
8. Pull Up Attribute (UA)
9. Extract Superclass (ES)
10. Push Down Method (DM)

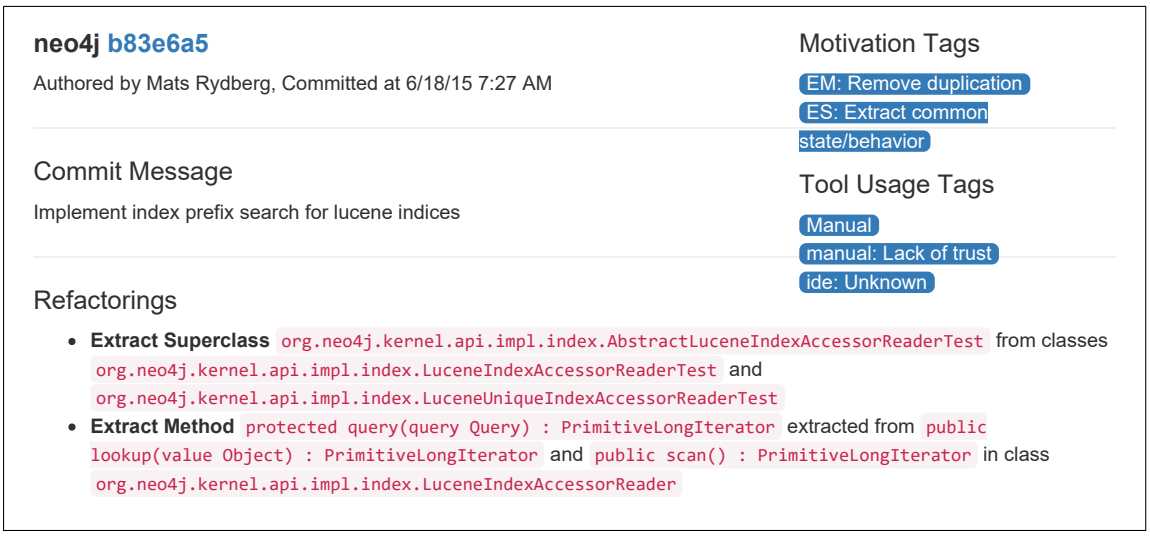


Figure 3.3: An example of a commit with refactorings and their motivations.

11. Push Down Attribute (DA)

12. Extract Interface (EI)

Additionally, the dataset includes a list of 222 commits with 463 refactorings and their motivations. This is the subset of the commits with refactorings where the developers answered to our mails, describing the reasons for performing the detected refactorings. The motivation theme for each refactoring was proposed after analyzing all developers' answers using thematic-analysis. The dataset contains details of the themes assigned by each evaluator that independently analyzed each commit, and also the final themes derived after a discussion phase to reach a consensus. Moreover, the dataset also includes information about the IDE used to perform the refactorings (if reported by the developer). In case the refactoring is performed manually, we also include a possible reason for not using the IDE refactoring tool support.

Figure 3.3 illustrates one of these commits. In this example, there are two refactorings applied in the Neo4j project. The first one is an Extract Superclass, in which the superclass `AbstractLuceneIndexAccessorReaderTest` was extracted from two other classes. This refactoring was performed to *Extract common state/behavior*. The second refactoring, in which the method `query` was extracted from two other methods of class `LuceneIndexAccessorReader`, was performed to *Remove duplication*. Additionally, the refactoring was performed manually (*Manual* tag) due to a *Lack of trust* in refactoring tools. Finally, the IDE is *Unknown*, because it was not reported by the developer. The dataset also includes meta-data about the commits (SHA1 hash, author, date, and time).

The dataset is available as a navigable web site, where researchers and practitioners can view and explore each commit including a refactoring with the motivation inferred in our study. However, we also provide the dataset in JSON format, to facilitate its import and use by other tools.

3.9.1 License

The dataset is distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, reproduction and adaptation in any medium provided that it is properly attributed.

Danilo Silva, Nikolaos Tsantalis, Marco Tulio Valente. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 24th International Symposium on the Foundations of Software Engineering* (FSE), pages 858-870, 2016.

3.9.2 How to contribute?

Researchers are welcome to contribute to this dataset either reporting issues or submitting new data from different projects or commits. Such contributions should be made via GitHub *issues* or *pull requests* in the following repository:

`https://github.com/aserg-ufmg/why-we-refactor`

Chapter 4

Detecting Refactoring in Version Histories

Identifying refactoring operations in source code changes is valuable to understand software evolution. Therefore, several tools have been proposed to automatically detect refactorings applied in a system by comparing source code between revisions. The availability of such infrastructure has enabled researchers to study refactoring practice in large scale, leading to important advances on refactoring knowledge. However, although a plethora of programming languages are used in practice, the vast majority of existing studies are restricted to the Java language due to limitations of the underlying tools. This fact poses an important threat to external validity. Thus, to overcome such limitation, we propose RefDiff 2.0, a multi-language refactoring detection tool. Our approach leverages techniques proposed in our previous work and introduces a novel refactoring detection algorithm that relies on the Code Structure Tree (CST), a simple yet powerful representation of the source code that abstracts away the specificities of particular programming languages. Despite its language-agnostic design, our evaluation shows that RefDiff's precision (96%) and recall (80%) are on par with state-of-the-art refactoring detection approaches specialized in the Java language. Our modular architecture also enables one to seamlessly extend RefDiff to support other languages via a plugin system. As a proof of this, we implemented plugins to support two other popular programming languages: JavaScript and C. Our evaluation in these languages reveals that precision and recall ranges from 88% to 91%. With these results, we envision RefDiff as a viable alternative for breaking the single-language barrier in refactoring research and in practical applications of refactoring detection.

4.1 Introduction

Knowing about the refactoring activity in software projects is a valuable information to help researchers understand software evolution. For example, past studies have used such information to shed light on important aspects of refactoring practice, such as: how developers refactor [Murphy-Hill et al., 2012], the usage of refactoring tools [Negara et al., 2013; Murphy-Hill et al., 2012], the motivations driving refactoring [Kim et al., 2012, 2014; Silva et al., 2016], the risks of refactoring [Kim et al., 2012, 2014, 2011; Weißgerber and Diehl, 2006a; Bavota et al., 2012], and the impact of refactoring on code quality metrics [Kim et al., 2012, 2014]. Moreover, it is often important to keep track of refactorings when performing source code evolution analysis because files, classes, or functions may have their histories split by the refactorings such as *Move* or *Rename* [Hora et al., 2018].

Additionally, knowing which refactoring operations were applied in the version history of a system may help in several practical tasks. For example, in a study by Kim et al. [2012], many developers mentioned the difficulties they face when reviewing or integrating code changes after large refactoring operations, which moves or renames several code elements. Thus, developers feel discouraged to refactor their code. If a tool is able to identify such refactoring operations, it can possibly resolve merge conflicts automatically. Moreover, diff visualization tools can also benefit from such information, presenting refactored code elements side-by-side with their corresponding version before the change. Another application for such information is adapting client code to a refactored version of an API it uses [Henkel and Diwan, 2005; Xing and Stroulia, 2008]. If we are able to detect the refactorings that were applied to an API, we might replay them on the client code automatically.

Given the importance of studying refactoring activity, we proposed RefDiff in previous work [Silva and Valente, 2017]. RefDiff is an automated approach that identifies refactorings operations performed in the version history of Java systems. By that time, our main goal was to provide a reliable tool to mine refactoring activity in a fully automated fashion, with better precision and recall than existing approaches. Since then, other approaches have emerged, such as RMiner [Tsantalis et al., 2018], which enhanced precision to even higher standards. Today, the availability of such tools enables large-scale and in-depth empirical studies on refactoring practice [Silva et al., 2016; Hora et al., 2018].

Nevertheless, despite the advancements in the field of refactoring detection, existing tools are all centered in the Java language. Thus, we are still not able to mine refactoring activity in a vast amount of software repositories written in other program-

ming languages. By restricting refactoring research to a single programming language, we may get a biased understanding of the reality. Moreover, the practical applications of such tools are hindered by the lack of support of other popular programming languages.

For all these reasons, in this chapter we propose a multi-language refactoring detection approach, named as RefDiff 2.0, which is a complete redesign of its first version that introduces an extensible architecture. In RefDiff 2.0, the refactoring detection heuristics are fully implemented in a common core, and support for programming languages is provided by plug-in modules. As a way to validate this architecture, we implemented and evaluated extension modules for three mainstream programming languages with distinct characteristics: Java, JavaScript (a widely popular dynamic programming language, used mostly to build web applications) and C (a procedural programming language, used mostly to implement system software).

Additionally, we reworked the refactoring detection heuristics of RefDiff to significantly improve its precision when compared to our previous work. Now, RefDiff achieves 96.4% of precision and 80.4% of recall when evaluated in the Java dataset proposed by Tsantalis et al. [2018], against 79.3% of precision and 80.2% of recall in its prior version. Moreover, RefDiff's precision is on par with RMiner, the current state-of-the-art in Java refactoring detection (96.4% vs. 98.8%). This is a relevant achievement because our approach is not specialized in a single language.

In summary, we deliver the following contributions in this work:

- A major extension of our refactoring detection approach proposed in previous work [Silva and Valente, 2017], which includes a redesign of its core to work with a language-independent model and improved detection heuristics.
- A publicly available implementation[1] of our approach, with out-of-the-box support for Java, C, and JavaScript.
- An evaluation of the precision and recall of RefDiff using a large scale dataset of refactorings performed in real-world Java open source projects, comparing it with RMiner, a state-of-the-art tool for detecting refactorings in Java. As a byproduct of this evaluation, we also extend the dataset with new refactoring instances discovered by our tool.
- An evaluation of the precision and recall of RefDiff in real-world C and JavaScript open source projects.

[1] RefDiff and all evaluation data are public available in GitHub: `https://github.com/aserg-ufmg/RefDiff`

The remainder of this chapter is structured as follows. Section 4.2 describes related work, discussing existing refactoring detection approaches. Section 4.3 presents the proposed approach in details. Section 4.4 describes the design and results of a large scale evaluation of RefDiff in Java projects. Section 4.5 describes the design and results of an evaluation of RefDiff in C and JavaScript projects. Section 4.6 discusses challenges and limitations. Last, Section 4.7 presents final remarks and concludes the chapter.

4.2 Background

Empirical studies on refactoring rely on means to identify refactoring activity. Thus, different techniques have been proposed and employed for this task. For example, Murphy-Hill et al. [2012] collected refactoring usage data using a plug-in that monitors user actions in the Eclipse IDE, including calls to refactoring commands. Negara et al. [2013] describe a tool, called CodingTracker, to infer refactorings from fine-grained code edits. They use this tool to study refactorings performed by 23 developers working in their IDEs during a total of 1,520 hours. The tool achieved a precision of 99.3% when evaluated with the automated Eclipse refactorings performed by the study participants. On a sample of both manual and automated refactorings, CodingTracker achieved a precision of 93% and a recall of 100%. However, CodingTracker requires the installation of a refactoring inference plugin in IDEs.

Other studies use metadata from version control systems to identify refactoring changes. For example, Ratzinger et al. [2008] search for a predefined set of terms in commit messages to classify them as refactoring changes. In specific scenarios, a branch may be created exclusively to refactor the code, as reported by Kim et al. [2014]. Another strategy is employed by Soares et al. [2010]. They propose an approach that identifies behavior-preserving changes by automatically generating and running test-cases. While their approach is intended to guarantee the correct behavior of a system after refactoring, it may also be employed to classify commits as behavior-preserving. Moreover, many existing approaches are based on static analysis. This is the case of the approach proposed by Demeyer et al. [2000], which finds refactored elements by observing changes in code metrics.

Static analysis is also frequently used to find differences in the source code by comparing two revisions [Dig et al., 2006; Weißgerber and Diehl, 2006b; Tsantalis et al., 2013; Prete et al., 2010; Kim et al., 2010; Silva and Valente, 2017; Tsantalis et al., 2018]. Approaches based on comparing source code differences have the advantage of beeing

able to identify refactoring operations applied in version histories. As RefDiff is one of these approaches, it can be directly compared with others within this category. In the next sections, we discuss RefDiff 1.0 and three other approaches.

4.2.1 RefDiff 1.0

The original version of RefDiff [Silva and Valente, 2017], which we will denote as RefDiff 1.0 throughout this chapter, employs a combination of heuristics based on static analysis and code similarity to detect 13 well-known refactoring types. One of its distinguishing characteristic is the use of the classical TF-IDF similarity measure from information retrieval to compute code similarity. In our previous work, we evaluated RefDiff 1.0 using an oracle of 448 refactoring operations, distributed across seven Java projects. We built this oracle by deliberately applying refactorings in software repositories in a controlled manner. Although this strategy poses the risk of creating an artificial dataset, this way we assured this oracle was complete and could be used to compute both precision and recall. We compared our tool with three existing approaches, namely Refactoring Miner 0.1 [Tsantalis et al., 2013], Refactoring Crawler [Dig et al., 2006], and Ref-Finder [Kim et al., 2010]. Our approach achieved precision of 100% and recall of 88%, surpassing the three tools subjected to the comparison.

4.2.2 Refactoring Miner/RMiner

Refactoring Miner 0.1 is an approach originally introduced by Tsantalis et al. [2013], 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*. In its original version, Refactoring Miner employs a lightweight algorithm, similar to the UMLDiff proposed by Xing and Stroulia [2005], for differencing object-oriented models, inferring the set of classes, methods, and fields added, deleted or moved between two code revisions. Refactoring Miner was employed and evaluated in empirical studies on refactoring along its evolution. In the first study, using the version histories of JUnit, HTTPCore, and HTTPClient, Tsantalis et al. [2013] reported 8 false positives for the *Extract Method* refactoring (96.4% precision) and 4 false positives for the *Rename Class* refactoring (97.6% precision). No false positives were reported for the remaining refactorings. In a second study that mined refactorings in 285 GitHub hosted Java repositories [Silva et al., 2016], we found found 1,030 false positives out of 2,441 refactorings (63% precision). However, we also evaluated Refactoring Miner using as a benchmark the dataset

reported by Chaparro et al. [2014], in which it achieved 93% precision and 98% recall.

In a recent study, Tsantalis et al. [2018] proposed a major evolution of its tool, now named as RMiner 1.0. In its current version, RMiner relies on an AST-based statement matching algorithm and a set of detection rules that cover 15 representative refactoring types. Its statement matching algorithm employes two techniques to be resilient to code restructuring during refactoring: abstraction, which deals with changes in statements' AST type due to refactorings, and argumentization, which deals with changes in sub-expressions within statements due to parameterization. To evaluate RMiner, the authors created a dataset with 3,188 real refactorings instances from 185 open-source projects. Using this oracle, the authors found that RMiner has a precision of 98% and recall of 87%, which was the best result so far, surpassing RefDiff 1.0, the previous state-of-the-art, which achieved precision of 75.7% and recall of 85.8% in this dataset.

4.2.3 Refactoring Crawler

Refactoring Crawler, proposed by Dig et al. [2006], is an approach capable of finding seven high-level refactoring types: *Rename Package/Class/Method*, *Pull Up Method*, *Push Down Method*, *Move Method*, and *Change Method Signature*. It uses a combination of syntactic analysis to detect refactoring candidates and a reference graph analysis to refine the results.

First, Refactoring Crawler analyzes the abstract syntax tree of a program and produces a tree, in which each node represents a source code entity (package, class, method, or field). Then, it employs a technique known as *shingles encoding* to find similar pairs of entities, which are candidates for refactorings. Shingles are representations for strings with the following property: if a string changes slightly, then its shingles also change slightly. In a second phase, Refactoring Crawler applies specific strategies for detecting each refactoring type, and computes a more costly metric that determines the similarity of references between code entities in two versions of the system. For example, two methods are similar if the sets of methods that call them are similar, and the sets of methods they call are also similar. The strategies to detect refactorings are repeated in a loop until no new refactorings are found. Therefore, the detection of a refactoring, such as a rename, may change the reference graph and enable the detection of new refactorings.

Dig et al. [2006] evaluated Refactoring Crawler comparing pairs of releases of three open-source software components: Eclipse UI, Struts, and JHotDraw. Such components were chosen because they provided detailed release notes describing API

changes. The authors relied on such information and on manual inspection to build an oracle containing 131 refactorings. The reported results are: Eclipse UI (90% precision and 86% recall), Struts (100% precision and 86% recall), and JHotDraw (100% precision and 100% recall). However, in our previous study [Silva and Valente, 2017], Refactoring Crawler achieved only 41.9% of precision and 35.6% of recall.

4.2.4 Ref-Finder

Ref-Finder, proposed by Prete et al. [2010]; Kim et al. [2010], is an approach based on logic programming capable of identifying 63 refactoring types from the Fowler's catalog [Fowler, 1999]. The authors express each refactoring type by defining structural constraints, before and after applying a refactoring to a program, in terms of template logic rules.

First, Ref-Finder traverses the abstract syntax tree of a program and extracts facts about code elements, structural dependencies, and the content of code elements, to represent the program in terms of a database of logic facts. Then, it uses a logic programming engine to infer concrete refactoring instances, by creating a logic query based on the constraints defined for each refactoring type. The definition of refactoring types also consider ordering dependencies among them. This way, lower-level refactorings may be queried to identify higher-level, composite refactorings. The detection of some types of refactoring requires a special logic predicate that indicates that the similarity between two methods is above a threshold. For this purpose, the authors implemented a block-level clone detection technique, which removes any beginning and trailing parenthesis, escape characters, white spaces, and return keywords and computes word-level similarity between the two texts using the longest common sub-sequence algorithm.

The authors evaluated Ref-Finder in two case studies. In the first one, they used code examples from the Fowler's catalog to create instances of the 63 refactoring types. The authors reported 93.7% recall and 97.0% precision for this first study. In the second study, the authors used three open-source projects: Carol, jEdit, and Columba. In this case, Ref-Finder was executed in randomly selected pairs of versions. From the 774 refactoring instances found, the authors manually inspected a sample of 344 instances and found that 254 were correct (73.8% precision). However, in a study by Soares et al. [2013] using a set of randomly select revisions of JHotDraw and Apache Common Collections containing 81 refactoring instances in total, Ref-Finder achieved 35% of precision and 24% of recall. Moreover, in our previous study [Silva and Valente, 2017], it also achieved 26.4% of precision and 64.2% of recall.

It is worth noting that Ref-Finder and Refactoring Crawler require a full build of the program under analysis. Therefore, their usage is not recommended when mining refactorings from version histories in the large. In this case, it might be a challenge to build each release, due to missing external dependencies, for example. For that reason, our evaluation (Section 4.4) focus on comparing RefDiff with RMiner.

4.3 Proposed Approach

Our approach consists of two phases: Source Code Analysis and Relationship Analysis. In the first phase, Source Code Analysis, we take as input two revisions of a system, v_1 and v_2, and build two models that represent their source code. Both models have the form of a tree, in which each node corresponds to a code element (classes, functions, etc.). In the second phase, Relationship Analysis, we compute a set R, which contains triples of the form (n_1, n_2, t), where n_1 is a code element from revision v_1, n_2 is a code element from revision v_2 and t is a relationship type. Such relationships may denote a high-level refactoring operation (move, rename, extract, etc.) or an exact correspondence between the code elements. For example, consider the diff between two revisions of a system depicted in Figure 4.1. Among other changes, the class `Calculator`, declared in revision 1, is renamed to `FpCalculator` in revision 2. This corresponds to a relationship of the type *Rename* between them. In the next sections, we describe in details each phase of our approach.

4.3.1 Phase 1: Source Code Analysis

The goal of this phase is to compute a language-independent model that represents the source code of the system, which we denote from now on as *Code Structure Tree* (CST). The CST is a tree-like structure that resembles an *Abstract Syntax Tree* (AST). However, in this representation we are only interested in coarse-grained code elements (e.g., classes and functions) that encompass a code region and may be referred by an identifier in other parts of the system.

To construct the CST, we need to parse the source code, generate the AST for the target programming language, and extract the necessary information. Thus, the decision of which types of AST nodes become CST nodes depends on the programming language. For example, in Java we represent classes, enums, interfaces, and methods as CST nodes. In contrast, local variables are not represented. Nevertheless, it is important to note that the granularity of the CST nodes determines the granularity of the relationships we are able to find, e.g., we can only find relationships between

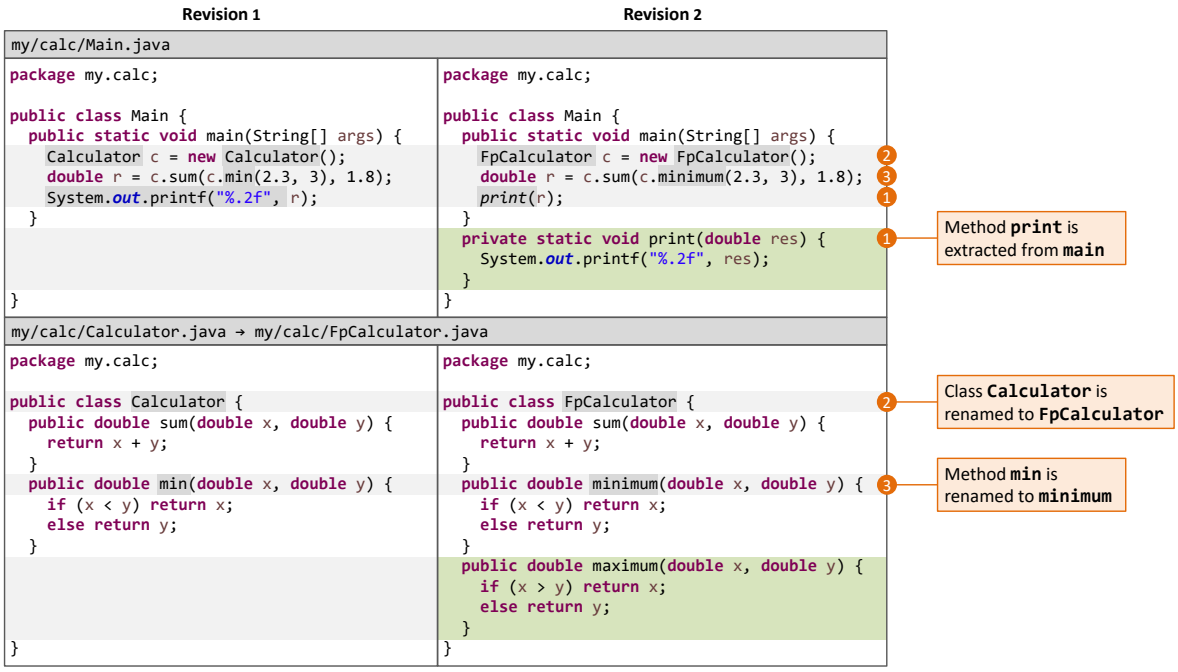


Figure 4.1: Illustrative diff between two revisions of a system annotated with refactoring operations

methods if we represent methods in the CST. Table 4.1 lists the types of AST nodes that are represented in the CST for each programming language supported by the current implementation of our approach.

Table 4.1: AST nodes that are represented in CSTs

Language	Node types
Java	class, enum, interface, method, and constructor
C	file and function
JavaScript	file, class, and function

Figure 4.2 exemplifies the transformation of the example system from Figure 4.1 into a corresponding CST. In revision 1, the class `Main` is declared with a single method `main` and the class `Calculator` contains two methods: `sum` and `min`. Note that these classes and methods become nodes in the CST for revision 1, preserving the same nesting structure of the source code. Analogously, the figure also depicts the CST for revision 2, which contains seven nodes in total (two classes and five methods).

Besides the representation of the code elements, the CST also embeds a simplified call graph and a type hierarchy graph of the nodes within the CST, that is, there are edges to represent whether a certain node n_1 calls n_2, or whether n_1 is a subtype of

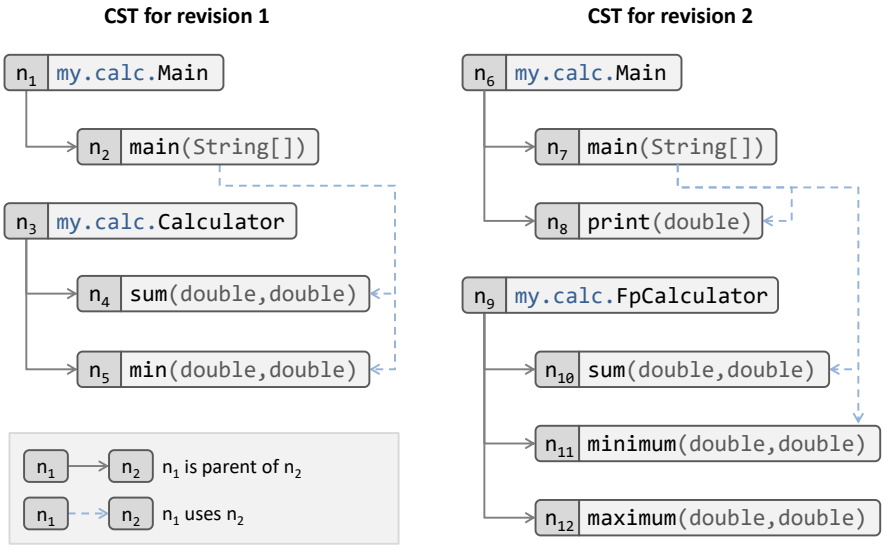


Figure 4.2: CST of both revisions of the example system from Figure 4.1

n_2. The first information is necessary to find *Extract* and *Inline* relationships between code elements, while the second is used to find inheritance-related relationships, such as *Pull Up* and *Push Down*.

Moreover, along with each node of the CST, we store the following information:

Identifier

An identifier of the code element in its declared scope. The identifier is usually the name of the code element, but it may also contain additional information to avoid ambiguities. For example, the identifier of the class `Calculator` from Figure 4.2 is simply its name, but the identifier of the method `sum` is `sum(double,double)`, because there could be an overloaded method with a different signature.

Namespace

An optional prefix that, along with the identifier, globally identifies the code element. This information only applies to top-level nodes and corresponds to the package or folder that the element is contained. For example, the namespace of the class `Calculator` from Figure 4.2 is `my.calc.`.

Node type

A string that identifies the node type in the target language (class, function, method, etc.).

Parameters list

An optional list of the name of the parameters, in the case the node corresponds

to a method or function.

Tokenized source code

The source code of the element in the form of a list of tokens. Here, we include all tokens in the code region that encompasses the complete declaration of the code element, including its name/signature. This information is necessary to compute the similarity between code elements, as explained in Section 4.3.3.

Tokenized source code of the body

The source code of the body of the code element in the form of a list of tokens. Here we include only the tokens within the body of the code element, but not its name/signature. This information is also necessary to compute the similarity between code elements in the special cases of *Extract* and *Inline* relationships, as explained in Section 4.3.3.2. It is worth noting that this information is optional, as not every node has a body (e.g., abstract methods).

It is worth noting that we generate the CST only for source files that have been added, removed, or modified between revisions. Such information can be efficiently obtained from version control systems, without the need to analyze the content of all files within the repository. This way, we avoid a costly operation that might compromise the scalability of our approach, as large repositories contain thousands of source files, but only a small fraction of them change between revisions.

Although the construction of the CST is a language-specific process, from this point on, the approach is language-independent and relies only on information encoded in CSTs. This way, one is able to extend our approach to work with different programming languages only by implementing the *Source Code Analysis* module. To demonstrate this capability, we provide implementations for three programming languages: Java, C, and JavaScript.

4.3.2 Phase 2: Relationship Analysis

This phase takes as input the CST's of revisions v_1 and v_2 and outputs the set of relationships R. Let N_1 and N_2 be the sets of code elements from the CST's of v_1 and v_2 respectively. Each relationship $r \in R$ is a triple (n_1, n_2, t), where $n_1 \in N_1$, $n_2 \in N_2$, and t is a relationship type. The types of relationships are listed in the first column of Table 4.3, and can be subdivided into two groups:

- **Matching relationships**, which indicate that the node n_1 corresponds to n_2 in the subsequent revision. The possible matching relationships are *Same*, *Convert*

Type, *Pull Up*, *Push Down*, *Change Signature*, *Move*, *Rename*, and *Move and Rename*. We say that a node n_1 matches with n_2 if there exists a relationship $(n_1, n_2, t) \in R$ such that t is a matching relationship.

- **Non-matching relationships**, which indicate that either node n_1 is decomposed to create n_2, or node n_1 is incorporated into n_2. There are four non-matching relationships: *Extract Supertype*, *Extract*, *Extract and Move*, and *Inline*.

4.3.2.1 General algorithm to find relationships

Our approach employs the algorithm described in Figure 4.3 to find the relationships (i.e., to compute the set R). The procedure FINDRELATIONSHIPS has two parameters, t_1 and t_2, which are the root nodes of the CST's of both revisions. Initially, we define $R \leftarrow \emptyset$ as the set of relationships found so far (line 2). Additionally, we also define $M \leftarrow \emptyset$ as the set of pairs of matching nodes found so far (line 3). Then, we execute four subroutines:

1. In FINDMATCHINGSBYID, we recursively look for matching nodes that have the same identifier and parent, i.e., we assume that code elements with the same identifier and parent are the same. In the case of top-level nodes, which do not have parents, their namespace should be the same. Such assumption allows us

Table 4.2: Definitions used in the Algorithm from Figure 4.3 and in the conditions from Table 4.3

Definitions			
M_1	the set of nodes from N_1 that matches with a node from N_2	name(n)	simple name of the code element n
M_2	the set of nodes from N_2 that matches with a node from N_1	id(n)	identifier of the code element n
N^-	the set of unmatched nodes from N_1 ($N_1 \setminus M_1$)	nType(n)	node type of the code element n
N^+	the set of unmatched nodes from N_2 ($N_2 \setminus M_2$)	subtype(n_1, n_2)	n_1 is subtype of n_2
n'	the code element that matches with n in the other revision	uses(n_1, n_2)	n_1 uses n_2
$\pi(n)$	parent of a node n (it may be a namespace or a CST node)	sim(n_1, n_2)	code similarity between n_1 and n_2
ns(n)	namespace of the code element n	nameSim(n_1, n_2)	name similarity between n_1 and n_2
childr(n)	set of children of n in the CST	$\text{sim}_\text{x}(n_1, n_2)$	extract similarity between n_1 and n_2
		sortBySim(S)	elements of S sorted by sim descending

```
1: procedure FindRelationships(t1, t2)
2:     R ← ∅
3:     M ← ∅
4:     findMatchingsById(t1, t2)
5:     findMatchingsBySim
6:     findMatchingsByChildr
7:     resolveMatchings
8:     findNonMatchingRel
9:     return R
10:
11:    procedure findMatchingsById(p1, p2)
12:        for each (n1, n2) ∈ childr(p1) × childr(p2) do
13:            if id(n1) = id(n2) ∧ ns(n1) = ns(n2) then
14:                addMatch(n1, n2)
15:            end if
16:        end for
17:    end procedure
18:
19:    procedure findMatchingsBySim
20:        for each (n1, n2) ∈ sortBySim(N⁻ × N⁺) do
21:            if findMatchRel(n1, n2) ≠ ∅ then
22:                addMatch(n1, n2)
23:            end if
24:        end for
25:    end procedure
26:
27:    procedure findMatchingsByChildr
28:        for each (n1, n2) ∈ sortBySim(N⁻ × N⁺) do
29:            if matchingChildr(n1, n2) > 1 ∧ nameSim(n1, n2) > 0.5 then
30:                addMatch(n1, n2)
31:            end if
32:        end for
33:    end procedure
34:
35:    procedure resolveMatchings
36:        for each (n1, n2) ∈ M do
37:            R ← R ∪ findMatchRel(n1, n2)
38:        end for
39:    end procedure
40:
41:    procedure findNonMatchingRel
42:        for each (n1, n2) ∈ M1 × N⁺ do
43:            R ← R ∪ findExtractSupertype(n1, n2)
44:            R ← R ∪ findExtract(n1, n2)
45:            R ← R ∪ findExtractMove(n1, n2)
46:        end for
47:        for each (n1, n2) ∈ N⁻ × M2 do
48:            R ← R ∪ findInline(n1, n2)
49:        end for
50:    end procedure
51:
52:    procedure addMatch(n1, n2)
53:        if n1 ∈ N⁻ ∧ n2 ∈ N⁺ then
54:            M ← M ∪ {(n1, n2)}
55:            findMatchingsById(n1, n2)
56:        end if
57:    end procedure
58: end procedure
```

Figure 4.3: Algorithm to find relationships

to match many code elements at this step, reducing the number of possibilities that need to be checked in the next steps. The procedure consists of a loop that pairs the children of the nodes received as arguments and calls the procedure ADDMATCH whenever a matching is found (line 13). On its turn, ADDMATCH (lines 46-51) adds a pair of matching nodes to M and calls FINDMATCHINGSBYID again to look for matchings on their children, completing the recursion. The matching pairs found in this step will be resolved to *Same* and *Convert type* relationships later (see step 4).

2. In FINDMATCHINGSBYSIM, we look for matching nodes based on code similarity. The goal is to find *Change Signature*, *Pull Up*, *Push Down*, *Move*, *Rename*, and *Move and Rename* relationships. The procedure iterates over the unmatched pairs of nodes sorted by similarity in descending order. We use the notation N^- to denote the set of unmatched nodes from t_1 (presumably deleted) and N^+ to denote the set of unmatched nodes from t_2 (presumably added). For each pair (n_1, n_2), the procedure checks if it meets the conditions (specified in the second column of Table 4.3) for any matching relationship by calling *findMatchRel*(n_1, n_2). This function returns a singleton containing a matching relationship or an empty set if none of the conditions are met. Last, the ADDMATCH subroutine is called in the case of a matching (line 24). The conditions to find those relationships and the sortBySim function rely on a code similarity metric, which is described in details in Section 4.3.3.

3. In FINDMATCHINGSBYCHILDR, we look for matching nodes based on matchings of their children and name similarity. Once again, the procedure iterates over the unmatched pairs of nodes sorted by similarity in descending order. For each pair (n_1, n_2), if n_1 has more than one children that match with n_2's children and their names are similar, then we consider it a match. The nameSim function, used to compute the similarity between names, is described in details in Section 4.3.3.1. This heuristic is intended to cover the cases when a code element (e.g., a class) is moved (and/or renamed) and it is also subjected to many additions or removals of its members, so that its similarity with its matching pair is not enough to yield a match in the previous step. Failing to detect that a class has been moved (or renamed) may yield several incorrect *Move* relationships between its members before and after the change.

4. In RESOLVEMATCHINGS, we add the relationships corresponding to the matching pairs found at steps 1, 2, and 3 to R. The procedure iterates over the elements of

M and calls *findMatchRel* to find which relationship type holds between n_1 and n_2 (according to the conditions defined in Table 4.3). By the end of this step, R contains all matching relationships found. The rationale for postponing the resolution of the relationship type is discussed in Section 4.3.2.2.

5. In findNonMatchingRel, we look for non-matching relationships. First, we iterate over the pairs of matched/unmatched nodes, i.e., $M_1 \times N^+$, to look for *Extract Supertype*, *Extract* and *Extract and Move* relationships. Similarly, we also iterate over the pairs of unmatched/matched nodes ($N^- \times M_2$) to search for *Inline* relationships. The functions $findExtractSupertype$, $findExtract$, $findExtractMove$, and $findInline$ check the preconditions for the corresponding relationship types, according to Table 4.3. After this last step, R contains all matching and non-matching relationships between CST nodes of both revisions.

Table 4.3: Relationship types and the conditions to find them

Relationship type	Conditions
	$(n_1, n_2) \in N^- \times N^+$, such that:
Same	$\text{nType}(n_1) = \text{nType}(n_2) \wedge \text{id}(n_1) = \text{id}(n_2) \wedge \pi(n_1)' = \pi(n_2)$
Convert Type	$\text{nType}(n_1) \neq \text{nType}(n_2) \wedge \text{id}(n_1) = \text{id}(n_2) \wedge \pi(n_1)' = \pi(n_2)$
Pull Up	$\text{nType}(n_1) = \text{nType}(n_2) \wedge \text{id}(n_1) = \text{id}(n_2) \wedge \text{subtype}(\pi(n_1)', \pi(n_2))$
Push Down	$\text{nType}(n_1) = \text{nType}(n_2) \wedge \text{id}(n_1) = \text{id}(n_2) \wedge \text{subtype}(\pi(n_2), \pi(n_1)')$
Change Signature	$\text{nType}(n_1) = \text{nType}(n_2) \wedge \text{id}(n_1) \neq \text{id}(n_2) \wedge \text{name}(n_1) = \text{name}(n_2) \wedge \pi(n_1)' = \pi(n_2) \wedge \text{sim}(n_1, n_2) > 0.5$
Move	$\text{nType}(n_1) = \text{nType}(n_2) \wedge \text{name}(n_1) = \text{name}(n_2) \wedge \pi(n_1)' \neq \pi(n_2) \wedge \text{sim}(n_1, n_2) > 0.5$
Rename	$\text{nType}(n_1) = \text{nType}(n_2) \wedge \text{name}(n_1) \neq \text{name}(n_2) \wedge \pi(n_1)' = \pi(n_2) \wedge \text{sim}(n_1, n_2) > 0.5$
Move and Rename	$\text{nType}(n_1) = \text{nType}(n_2) \wedge \text{name}(n_1) \neq \text{name}(n_2) \wedge \pi(n_1)' \neq \pi(n_2) \wedge \text{sim}(n_1, n_2) > 0.5$
	$(n_1, n_2) \in M_1 \times N^+$, such that:
Extract Supertype	$\exists (n_3, n_4, PullUp) \in R\,(n_1 = \pi(n_3) \wedge n_2 = \pi(n_4))$
Extract	$\text{uses}(n_1', n_2) \wedge \pi(n_1)' = \pi(n_2) \wedge \text{sim}_\text{x}(n_2, n_1) > 0.5$
Extract and Move	$\text{uses}(n_1', n_2) \wedge \pi(n_1)' \neq \pi(n_2) \wedge \text{sim}_\text{x}(n_2, n_1) > 0.5$
	$(n_1, n_2) \in N^- \times M_2$, such that:
Inline	$\text{uses}(n_1, n_2') \wedge \text{sim}_\text{x}(n_1, n_2) > 0.5$

Figure 4.4 shows the relationships we find after running RefDiff in the example from Figure 4.1. Each relationship is represented by an edge connecting nodes from the left and right CSTs. There are three relationships of the type *Same*, involving the code elements whose identifiers do not change: the class `Main` and the methods

`main` and `sum`. Two of the relationships are of type *Rename*, indicating that the class `Calculator` is renamed to `FpCalculator`, and the method `min` is renamed to `minimum`. Moreover, there is an *Extract* relationship indicating that the method `print` is extracted from `main`. Finally, we can also note that two nodes, n_8 and n_{12}, are not involved in matching relationships. Thus, we classify them as added code elements. In this example, as every node on the left side is matched, there are no deleted code elements.

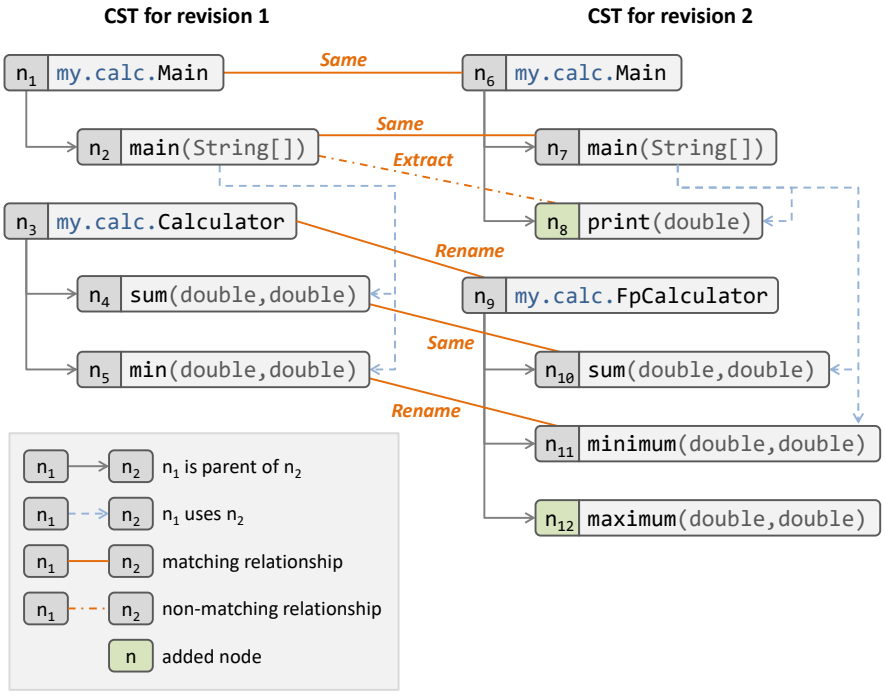


Figure 4.4: Relationships found in the example from Figure 4.1

4.3.2.2 Dependent and conflicting relationships

In some cases, correctly finding a relationship depends on finding a prior relationship. For example, consider the relationship $(n_5, n_{11}, \textit{Rename})$ in Figure 4.4 (method `min` renamed to `minimum`). The conditions for this relationship includes the clause $\pi(n_5)' = \pi(n_{11})$, which means that the matching node of the parent of n_5 should be equal to the parent of n_{11} (see Table 4.3, *Rename* row). This clause only yields true after the matching pair (n_3, n_9) is added to M, i.e., after we find out that `Calculator` is renamed to `FpCalculator`. In fact, if we call $\textit{findMatchRel}(n_5, n_{11})$ before M contains (n_3, n_9), we would incorrectly classify it as a *Move and Rename* relationship. To address this issue, we only resolve the actual relationship types in steps 4 and 5,

after all matching pairs are found (note that in steps 1, 2, and 3 we record the matching pairs in M, purposely ignoring the type of relationship).

Another issue which we may face when looking for relationships are conflicts, i.e., two or more matching relationships hold for the same code element (according to conditions from Table 4.3). For instance, in the example from Figure 4.4, the conditions for *Rename* yield true for the pair of methods `min` and `minimum` because their source code are similar and their parents match. However, this is also the case for the pair of methods `min` and `maximum`, whose bodies are also similar. We cannot match the same node twice, thus, we must decide upon which relationship we will accept and discard the other one. This issue is addressed in procedures findMatchingsBySim and findMatchingsByChildr by using the sortBySim function to sort the potential matching pairs, enforcing that we take first the most likely matches. The sortBySim function relies on a similarity metric, which we discuss in details in Section 4.3.3. After a matching pair (n_1, n_2) is added to M, no more matchings involving n_1 or n_2 are accepted, because addMatch procedure checks that $n_1 \in N^- \wedge n_2 \in N^+$ (line 47).

4.3.3 Code Similarity

A key element of our approach to find relationships, as previously mentioned, is computing the similarity between code elements (i.e., CST nodes). The first step to compute this similarity is to represent their source code as a multiset (or bag) of tokens. A multiset is a generalization of the concept of a set, but it allows multiple instances of the same element. The multiplicity of an element is the number of occurrences of that element within the multiset. This representation provides two advantages for our approach. First, it makes the CST simpler and less coupled to the syntax of programming languages, because we do not need to represent each statement (or AST node) from the source code. Second, we can apply set operations to the bag of tokens, such as subtraction, which is important to detect *Extract* relationships, as we will discuss in Section 4.3.3.2.

Formally, a multiset can be defined in terms of a multiplicity function $m : U \to \mathbb{N}$, where U is the set of all possible elements. In other words, $m(t)$ is the multiplicity of the element t in the multiset. Note that the multiplicity of an element that is not in the multiset is zero. For example, Figure 4.5 depicts the transformation of the source code of three methods (`sum`, `min`, and `power`), of the class `Calculator`, into multisets of tokens. In this figure, the multiplicity function m for each method is represented in a tabular form. For example, the multiplicity of the token `y` in method `min` is two (i.e., $m_{\texttt{min}}(\texttt{y}) = 2$), whilst the multiplicity of the token `if` in method `power` is zero

Source code of a class

```
public class Calculator {

  public int sum(int x, int y) {
    return x + y;
  }

  public int min(int x, int y) {
    if (x < y) return x;
    else return y;
  }

  public double power(int b, int e) {
    return Math.pow(b, e);
  }
}
```

⇒

Multiset of tokens for each method

Token t	$m_{\texttt{sum}}(t)$	$m_{\texttt{min}}(t)$	$m_{\texttt{power}}(t)$	n_t
`return`	1	2	1	3
`x`	1	2	0	2
`+`	1	0	0	1
`y`	1	2	0	2
`;`	1	2	1	3
`if`	0	1	0	1
`(`	0	1	1	2
`<`	0	1	0	1
`)`	0	1	1	2
`else`	0	1	0	1
`Math`	0	0	1	1
`.`	0	0	1	1
`pow`	0	0	1	1
`b`	0	0	1	1
`,`	0	0	1	1
`e`	0	0	1	1

Figure 4.5: Transformation of the body of methods into a multiset of tokens

(i.e., $m_{\texttt{power}}(\texttt{if}) = 0$).

After extracting a multiset of tokens, we also compute a weight for each token of the source code. In fact, some tokens are more important than others to discriminate a code element. For example, in Figure 4.5, all three methods contain the token `return`. In contrast, only one method (`power`) contains the token `Math`. Therefore, the later is a better indicator of similarity between methods than the former.

In order to take this into account, we employ a variation of the TF-IDF weighting scheme [Salton and McGill, 1984], which is a well-known technique from information retrieval. TF-IDF, which is the short form of *Term Frequency-Inverse Document Frequency*, reflects how important a term is to a document within a collection of documents. In the context of code elements, we consider a token as a term, and a code element as a document. Let E be the set of all code elements and n_t be the number of elements in E that contains the token t. The Inverse Document Frequency (idf), is defined as:

$$idf(t) = \log(1 + \frac{|E|}{n_t}) \tag{4.1}$$

Note that the value of $idf(t)$ decreases as n_t increases, because the more frequent a token is among the collection of code elements, the less important it is to distinguish them. For example, in Figure 4.5, the token `y` occurs in two methods (`sum` and `min`). Thus, its idf is:

$$idf(\texttt{y}) = \log(1 + \frac{|E|}{n_t}) = \log(1 + \frac{3}{2}) = 0.398$$

On the other hand, the token `else` occurs in one method (`min`), and therefore its *idf* is:

$$idf(\texttt{else}) = \log(1 + \frac{|E|}{n_t}) = \log(1 + \frac{3}{1}) = 0.602$$

Last, to compute the similarity between two code elements e_1 and e_2, we use a generalization of the Jaccard coefficient, known as weighted Jaccard coefficient [Chierichetti et al., 2010]. Let U be the set of all possible tokens and $m_i : U \to \mathbb{N}$ be the multiplicity function representing the multiset of tokens of a code element e_i. We define the similarity between e_1 and e_2 by the following formula:

$$\text{sim}(e_1, e_2) = \frac{\sum_{t \in U} \min(m_1(t), m_2(t)) \times idf(t)}{\sum_{t \in U} \max(m_1(t), m_2(t)) \times idf(t)} \tag{4.2}$$

The rationale behind this formula is that the similarity is at maximum (1.0) when the multiset of tokens representing e_1 and e_2 contain the same tokens with the same cardinality. In contrast, if the multisets contain no tokens in common, the similarity is zero. Additionaly, tokens with higher *idf* will have a higher weight.

4.3.3.1 Name similarity

Besides relying on the code similarity, our algorithm also depends on the function nameSim(n_1, n_2) in Step 3. This function denotes the similarity between the names of the code elements n_1 and n_2. For computing nameSim, we first decompose the identifiers of n_1 and n_2 into their composing words. Specifically, we split camel case (e.g., `myIdentifier`) or snake case patterns (e.g., `my_identifier`). For example, `SomeLong_Name` yields three terms: `Some`, `Long`, and `Name`. Then, we compute nameSim using the same formula from sim (see Equation 4.2), but in this case, each multiset of tokens contains the terms composing the identifiers of n_1 and n_2.

4.3.3.2 Extract similarity

While the similarity function sim presented previously is suitable to compute whether two code elements are similar, it is not appropriate to assess whether a code element is extracted from another one, because their source code may be significantly different on their entirety. However, if a method e_2 is extracted from e_1, we expect that part of

the code of e_1 is moved to e_2. Therefore, the source code of the body of e_2 should be similar to the source code removed from e_1. Additionally, not all code removed from e_1 may have been moved to e_2, i.e., some parts of the code may have been extracted to other locations or simply deleted. To be less susceptible to this issue, our similarity index for *Extract* relationships rely on the following assumption: the code from the body of e_2 should be mostly contained in the code removed from e_1.

Thus, to compute the extract similarity, first we need to compute the code removed from e_1. As we represent the source code as multisets of tokens, we are able to use the subtract operation to achieve this goal. Let m_1 be the multiset of tokens of e_1 before the change and m'_1 be the multiset of tokens of e_1 after the change. The subtract operation between both multisets, which we denote by $m'_1 \setminus m_1$, yields a new multiset m_1^- defined by the following multiplicity function:

$$m_1^-(t) = \max(0, m'_1(t) - m_1(t)) \tag{4.3}$$

Besides computing the code removed from e_1, we need to measure if it is contained within e_2. Thus, we employ a variation of the weighted Jaccard coefficient introduced previously (see Equation 4.2), which is defined as:

$$\text{sim}_{\subseteq}(m_1, m_2) = \frac{\sum_{t \in U} \min(m_1(t), m_2(t)) \times idf(t)}{\sum_{t \in U} m_1(t) \times idf(t)} \tag{4.4}$$

where m_1 and m_2 are multisets (defined by their multiplicity functions). In this variation, we change the denominator of Equation 4.2 to include only the multiplicity of the tokens from the first multiset (not their union). This way, the similarity is at maximum (1.0) when m_1 is a subset of m_2, even if both multisets are not identical. In contrast, the similarity is zero when the intersection between m_1 and m_2 is empty.

Given these definitions, we are able to define the extract similarity index, sim_x, as:

$$\text{sim}_\text{x}(e_1, e_2) = \text{sim}_{\subseteq}(m_2, m_1^-) \tag{4.5}$$

where m_1^- is the multiset representing the code removed from e_1 ($m_1 \setminus m'_1$) and m_2 is the multiset representing the source code of the body of e_2. The rationale behind this formula is that the similarity is at maximum when m_2 is a subset of m_1^-, addressing the previously described heuristic.

4.3.3.3 Inline similarity

The similarity index for computing *Inline* relationships is analogous to the *Extract* similarity index. If a code element e_1 is inlined into a code element e_2, we expect that the code from the body of e_1 should be mostly contained in the code added to e_2. Specifically, we define the inline similarity index, sim_i, as:

$$\text{sim}_\text{i}(e_1, e_2) = \text{sim}_\subseteq(m_1, m_2^+) \tag{4.6}$$

where m_1 is the multiset representing the source code of the body of e_1 and m_2^+ is the multiset representing the code added to e_2 ($m_2' \setminus m_2$). Such similarity index is at maximum (1.0) when m_1 is a subset of the added code (m_2^+).

4.3.3.4 Ignoring parameters and return keywords

When retrieving the tokenized source code of the body of a code element, some tokens are ignored to avoid that syntactical constructs necessary to its declaration introduce noise when computing the *Extract* or *Inline* similarity index. For example, suppose we take the refactoring operation #1 depicted in Figure 4.1: `print` is extracted from `main`. The body of the new method `print` contains a single statement:

```
System.out.printf("%.2f", res);
```

All the tokens within this method are present in `main` before the extraction, except the identifier `res`, which is a declared parameter of `print`. In fact, in the original statement, a variable `r` is used in place of `res`. To be less susceptible to such differences, we omit all occurences of parameters in the tokenized source code of the body. Similarly, occurences of `return` keywords are also ignored because they may be introduced when turning the extracted code into a method. It is worth noting that discarding such tokens is of responsability of the source analysis module. Thus, specific rules may be implemented according to the peculiarities of the programming language.

4.3.4 Implementation details

RefDiff implementation consists of a core module and language plugins:

- **`refdiff-core`**: implements our core algorithm and contains common data types to represent CSTs and interfaces to implement source code analysis (i.e., generation of CSTs) for each programming language. Currently, this module contains 3,103 lines of code, implemented in Java.

- **`refdiff-java`**: language plugin for Java, which relies on the Eclipse JDT library for parsing and analyzing Java code.[2] This module contains 1,137 lines of code.
- **`refdiff-c`**: language plugin for C, using the Eclipse CDT library.[3] This module contains 615 lines of code.
- **`refdiff-js`**: language plugin for JavaScript, using the Babel parser[4] and with 689 lines of code.

To add support to a new programming language, one must implement the `LanguagePlugin` interface, which defines two methods: `parse`, which builds the CST given a set of source files, and `getAllowedFilesFilter`, which returns an object with a list of allowed file extensions and an optional list of ignored file name suffixes. For example, **`refdiff-js`** ignores file names that end with `.min.js`, which are usually generated code.

When compared to existing refactoring detection approaches, RefDiff's design has the advantage of being loosed coupled to the syntax of Java (and of any other programming language). For example, RMiner, which is a Java-based approach, relies on a statement matching algorithm and applies two techniques to enable matching of statements that are not textually identical: Abstraction and Syntax-aware replacements of AST nodes [Tsantalis et al., 2018]. Both techniques manipulate syntactic constructs of the Java language, such as return statements, variable declarations, assignments, method invocations, conditional statements, class instantiations, types, literals, operators, and others. Thus, when adapting these techniques to other programming language, tool builders should carefully consider its particular syntactic constructs. On the other hand, RefDiff's similarity comparison is based on tokenized code. Therefore, it does not depend on the AST nodes of any given language. As another example, Java code is structured with classes, which contains methods and attributes, and RMiner detection rules are tightly based on this structure. In contrast, JavaScript code contains functions inside functions with arbitrary levels of nesting. RefDiff is able to deal with both languages because CSTs do not assume any particular hierarchical structure between different types of nodes.

In summary, we do not claim that existing approaches cannot not be extended to other languages, but that would require a non-trivial effort. By making fewer assumptions about the syntax of the target language we facilitate multi-language support.

[2] `https://www.eclipse.org/jdt/`
[3] `https://www.eclipse.org/cdt/`
[4] `https://babeljs.io/docs/en/babel-parser`

Note that the implemented language plugins have small code bases (between 615–1137 lines of code).

4.4 Evaluation with Java Projects

In this section, we evaluate the precision and recall of our approach using a recently proposed dataset of refactorings performed in real-world Java open-source projects. We also compare RefDiff's accuracy with RMiner—the current state-of-the-art tool for detecting refactorings in Java—and RefDiff 1.0, the previous version of our tool. First, we present our evaluation design (Section 4.4.1) and then we present the results (Section 4.4.2).

4.4.1 Evaluation Design

To evaluate the precision and recall of RefDiff in Java we initially use an oracle proposed by Tsantalis et al. [2018]. This oracle includes 3,188 manually-validated refactoring instances, detected in 538 commits from 185 open-source projects, and covering 15 refactoring types. It is important to emphasize that most commits contain non-refactoring changes interleaved with refactorings, which is the most challenging scenario for refactoring detection tools. In our evaluation, we also compare RefDiff's precision and recall against RMiner 1.0. For the purpose of the comparison, we restricted the oracle to 11 refactoring types supported by both tools. Specifically, we excluded *Change Package*, *Move Field*, *Push Down Field* and *Pull Up Field* from the analysis as they are not supported by RefDiff. Moreover, *Convert Type* and *Change Signature*, although supported by RefDiff, are not evaluated because they are not covered by the oracle. In total, our modified oracle contains 3,031 confirmed refactoring instances. Additionally, it also contains 704 refactoring instances classified as false positives in the process of manual validation performed by Tsantalis et al. [2018]. These instances are used to detect false positives reported by RefDiff, as described in the next paragraph.

First, we run RefDiff on each commit of the oracle. For each detected refactoring r we checked whether r is in the oracle, which may yield three outcomes: (i) if r is a confirmed refactoring from the oracle, then it is a true positive; (ii) if r is a false refactoring from the oracle, then it is a false positive; (iii) otherwise, r was inspected by two researchers to assess whether it is a false positive or a true positive not covered by the oracle. This extra manual validation is needed because the initial oracle must not be granted as complete, i.e., including all refactorings performed in the set of analysed commits. Specifically, it was constructed using a triangulation approach, based on an

initial list of refactorings produced by RMiner and RefDiff 1.0. For this reason, it might miss true refactorings only detected by RefDiff 2.0.

After following this procedure, RefDiff 2.0 detected 263 new refactoring instances (i.e., not listed in the initial oracle), which were validated by two researchers, called here validators. In the case of 175 refactorings (66%), the validators agreed on their classification, including 138 refactorings labelled as true positives by both validators and 37 labelled as false positives. After this initial and independent validation, the validators discussed together the remaining 88 cases (34%), to reach an agreement. As a result, 79 refactorings were considered true positives and 9 refactorings were classified as false positives. Figure 4.6 shows an example of a refactoring identified by RefDiff that both validators classified as true positive. In this case, a developer extracted method `createPrepareRpcOptions` from method `prepareOnAffectedNodes`.

In total, after completing the manual validation, 217 new refactorings instances were classified as true positives and therefore included in the oracle. The expanded oracle includes 3,248 refactoring instances (7.19% more than the initial one) and it is publicly available at RefDiff's GitHub repository.[5]

```
213     -   protected void prepareOnAffectedNodes(TxInvocationContext<?> ctx, PrepareCommand command, Collection<Address> recipients, bool
    211 +   protected void prepareOnAffectedNodes(TxInvocationContext<?> ctx, PrepareCommand command, Collection<Address> recipients) {
214 212         try {
215 213            // this method will return immediately if we're the only member (because exclude_self=true)
216     -          RpcOptions rpcOptions;
217     -          if (sync) {
218     -             rpcOptions = rpcManager.getRpcOptionsBuilder(ResponseMode.SYNCHRONOUS_IGNORE_LEAVERS, DeliverOrder.NONE).build();
219     -          } else {
220     -             rpcOptions = rpcManager.getDefaultRpcOptions(false);
221     -          }
222     -          Map<Address, Response> responseMap = rpcManager.invokeRemotely(recipients, command, rpcOptions);
223     -          checkTxCommandResponses(responseMap, command);
    214 +          Map<Address, Response> responseMap = rpcManager.invokeRemotely(recipients, command, createPrepareRpcOptions());
    215 +          checkTxCommandResponses(responseMap, command, (LocalTxInvocationContext) ctx, recipients);
224 216         } finally {
225 217            transactionRemotelyPrepared(ctx);
226 218         }
227 219      }

    417 +   protected RpcOptions createPrepareRpcOptions() {
    418 +      return cacheConfiguration.clustering().cacheMode().isSynchronous() ?
    419 +            rpcManager.getRpcOptionsBuilder(ResponseMode.SYNCHRONOUS_IGNORE_LEAVERS, DeliverOrder.NONE).build() :
    420 +            rpcManager.getDefaultRpcOptions(false);
    421 +   }
```

Figure 4.6: Illustrative diff of an *Extract Method* refactoring considered as true positive by the validators, taken from commit ce4f629 from *infinispan* project.

Table 4.4: Java precision and recall results

		RefDiff 1.0		RefDiff 2.0		RMiner 1.0	
Refactoring Type	#	Precision	Recall	Precision	Recall	Precision	Recall
Move Class	1,100	0.999	0.881	0.999	0.970	1.000	0.925
Move Method	319	0.322	0.746	0.871	0.803	0.955	0.658
Move and Rename/Rename Class	95	0.897	0.642	0.922	0.874	0.983	0.621
Rename Method	350	0.855	0.811	0.946	0.694	0.978	0.771
Extract Interface	24	0.769	0.417	0.875	0.875	1.000	0.833
Extract Superclass	70	1.000	0.157	1.000	0.743	0.958	0.971
Pull Up Method	91	0.806	0.593	0.974	0.824	1.000	0.791
Push Down Method	40	0.950	0.475	0.950	0.950	1.000	0.825
Extract/Extract and Move Method	1,037	0.904	0.833	0.962	0.663	0.985	0.768
Inline Method	122	0.842	0.787	0.957	0.721	0.990	0.795
Total	3,248	0.792	0.802	0.964	0.804	0.988	0.813

4.4.2 Results

Table 4.4 shows the precision and recall results for RefDiff 2.0 and RMiner using the oracle described in the previous section. The overall precision and recall of RefDiff 2.0 is 96.4% and 80.4%, respectively. Precision ranges from 87.1% (*Move Method*) to 100.0% (*Extract Superclass*), and it is above 90% for 8 out of 10 refactoring types. Recall ranges from 66.3% (*Extract Method*) to 97.0% (*Move Class*), and it is above 80% for 6 out of 10 refactoring types.

4.4.2.1 Comparison with RefDiff 1.0

We also show in Table 4.4 the results obtained with RefDiff 1.0 in this oracle. Note that overall precision is significantly improved (from 79.2% to 96.4%). Moreover, RefDiff 2.0 has less variation on recall across refactoring types. We can list five improvements over RefDiff 1.0 that justify such results.

- In RefDiff 2.0, we find moved/renamed types (e.g., classes) based on matched members (step 3 of our algorithm). This heuristic was introduced aiming to reduce the number of false negatives for class moves/renames, which also reduces the number of false positives for *Move Method*.
- We compute the removed and added code using set operations to improve *Extract* and *Inline* similarity functions. For example, our *Extract* similarity function compares the body of an extracted method with the code removed from the original method, strengthening the preconditions to detect *Extract* relationships.

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

Similarly, our *Inline* similarity function compares the body of an inlined method with the code added to its destination.

- Ignoring parameters/arguments and return keyword (Section 4.3.3.4) is also an improvement over RefDiff 1.0, making *Extract* and *Inline* similarity less sensitive to code changes related to the mechanics of the refactoring.

- *Pull Up* and *Push Down* rules no longer include body similarity comparison. Additionally, *Extract Supertype* also drops similarity comparison and it was rewritten based on *Pull Up* rule. These changes improved both precision and recall of these refactoring types.

- While RefDiff 1.0 relied on a set of thresholds, which were calibrated for each refactoring type, in RefDiff 2.0 we use a single similarity threshold, defined as 0.5 by default. We acknowledged that relying on user-defined thresholds or thresholds calibration is not ideal, as advocated by Tsantalis et al. [2018]. Thus, in RefDiff 2.0 we emphasized the aforementioned improvements to our algorithm, making it is less sensitive to similarity thresholds. In fact, we achieved better precision for all refactoring types, even without calibration. We only lost recall for *Rename Method*, *Extract Method* and *Inline Method*. We attribute this fact to the very low thresholds set for these refactoring types (between 0.1 and 0.3).

4.4.2.2 Comparison with RMiner

Table 4.4 also shows the results of RMiner, which achieves 98.8% of overall precision (ranging from 95.5% to 100.0%) and 81.3% of overall recall (ranging from 64.1% to 97.1%). When we analyze individual refactoring types, RefDiff's precision is lower in all but one refactoring type (*Extract Superclass*). However, recall is higher in 6 refactoring types. In summary, both tools have very similar total recall, but RMiner's precision is slightly higher. We can list at least three differences between RefDiff and RMiner that might explain the differences in the results.

- Unlike RefDiff, we believe RMiner does not account for methods moved to added classes, nor methods moved from deleted classes, as RMiner's detection rule for *Move Method* includes the clauses $(td_a, td'_a) \in TD^{=}$ and $(td_b, td'_b) \in TD^{=}$ [Tsantalis et al., 2018]. Many of the false negatives for *Move Method* involve such scenarios, which explains the lower recall for RMiner.

- Both approaches find moved/renamed types (e.g., classes) based on matched members (step 3 of our algorithm). However, ReffDiff's detection rule requires

that a pair of candidate types (t_1, t_2) have more than one children in common, while RMiner's rule is more strict, requiring that either all members of t_1 are in t_2, or all members of t_2 are in t_1. Additionally, RefDiff also finds moved/renamed types by similarity. These might be the reasons for RMiner's lower recall for *Move and Rename/Rename Class*.

- They use very different approaches for computing code similarity. While RefDiff relies on tokenized code and a TF-IDF based similarity function, RMiner relies on a statement matching algorithm and syntax-aware replacement of AST nodes. Such difference potentially impacts precision and recall for several refactoring types, and might be an advantage factor for RMiner.

Despite the aforementioned differences, we emphasize that RefDiff and RMiner have much in common:

- Both approaches match elements by full name/signature in their first step.
- Many of the refactoring detection rules are similar.
- Both approaches enforce an order of detection between refactoring types and use a best match strategy to choose between conflicting refactoring candidates.
- RefDiff 2.0 included an heuristic to find moved/renamed types (e.g., classes) based on matched members, which is similar to RMiner's detection rule.
- Ignoring parameters/arguments and return keyword (Section 4.3.3.4) serves a similar purpose to the argumentization and abstraction techniques proposed by RMiner.

4.4.3 Execution time

Besides comparing precision and recall, we also compared the execution time of both RefDiff and RMiner. For this purpose, we ran both tools using the same computer (an Intel Core i5-750 with 8GB of RAM and a 7200 RPM HDD) and measured the time spent in the analysis of each of the 538 commits from our oracle.[6] Figure 4.7 shows a violin plot of the execution time per commit for both tools, using a log-10 scale. We can note that the median is lower for RMiner (109 ms vs. 157 ms), but RefDiff has less variation in the execution times. For example, the maximum execution time

[6]We repeated the experiment three times and discarded the times collected in the first run, which was considered as a warm up.

for RMiner was 85s, at commit ab98bca from *java-algorithms-implementation*, whilst RefDiff spent 10s at maximum, in commit 4baf0a4 from *aws-sdk-java*. Nevertheless, both tools analyze the majority of the commits in less than one second and are viable for practical use. It is worth noting that we executed both RefDiff and RMiner using their file-based API, which reads a list of files directly from disk. This means that the time to clone or checkout revisions from git repositories is not included in our measurements. However, we do not expect significant differences between both tools when using their git-based API, which includes services from mining refactorings directly from git repositories. The reason is that both tools retrieve only the necessary files using the *jgit* library, therefore avoiding checking out the entire project on disk.

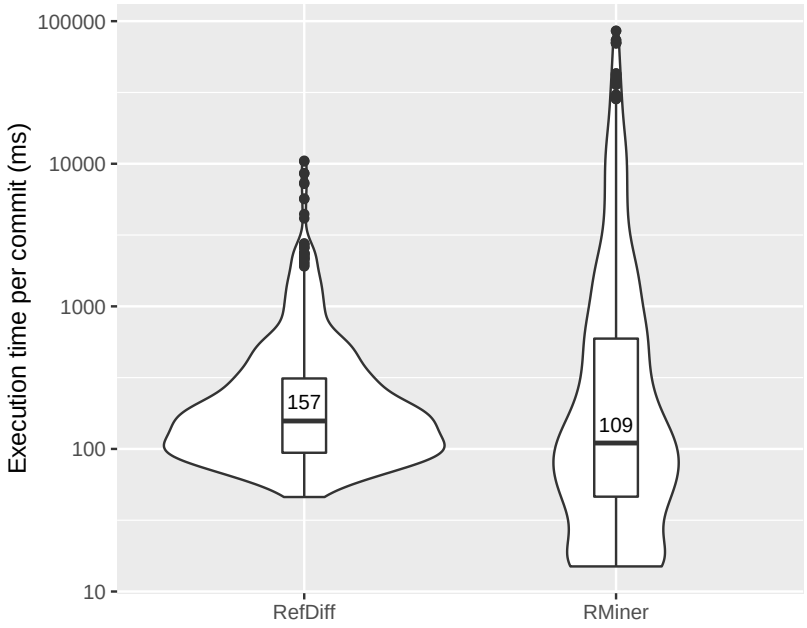


Figure 4.7: Violin plot of execution time per commit in log-10 scale

4.4.4 Threats to Validity

There are at least two threats to the validity of the evaluation with Java projects. The first one is the subjectivity inherent to the manual classification of the reported refactorings as true/false positives, which directly impact the computed precision. Different validators may have a different interpretation of the code change under analysis, which is demonstrated by the fact that in 34% of the cases the validators initially disagreed. We mitigated this threat by having each refactoring assessed by two validators independently. Moreover, there are not precise and consensual definitions of the mechanics of each refactoring type. In fact, Oliveira et al. [2019] shows that developers and IDE

refactoring tools use different mechanics for most refactoring types. Second, as discussed in Section 4.4.1, our oracle can not be granted as complete, i.e., there might exist refactorings in the analyzed revisions that are not detected by any of the tools. Thus, the actual recall might be lower than the computed recall. Unfortunately, it is not feasible to assure the completeness of an oracle at this scale with manual inspection. A single commit usually contains hundreds of changed lines of code, making such task extremely time consuming and error prone. Nevertheless, the computed recall serves the purpose of comparison between tools, and it should also be a fair approximation of actual recall, as our oracle is based on refactorings found by three tools: RMiner 1.0, RefDiff 1.0, and RefDiff 2.0.

4.5 Evaluation with JavaScript and C

Besides the Java evaluation, we also evaluated precision and recall of RefDiff in JavaScript and C. Unfortunately, we did not find a dataset with detailed information about real refactorings performed in these languages that we could use as an oracle. Therefore, we had to adopt a different strategy. To evaluate precision, we manually validated the refactorings detected by RefDiff in a set of GitHub repositories, in both languages (Section 4.5.1). Then, to evaluate recall, we searched for documented refactoring operations in commit messages of the same set of repositories (Section 4.5.2). After that, in Section 4.5.3, we report the precision and recall achieved by RefDiff. We are not aware of any other tool for detecting refactorings in these languages. Therefore, in this second evaluation, it was not possible to compare RefDiff's results with competitor tools.

4.5.1 Evaluation Design: Precision

To compute RefDiff's precision when detecting refactorings in JavaScript and C, we followed these steps:

1. We selected the 20 most popular GitHub repositories of each language. For this, we queried the GitHub API for repositories, sorting by stars count—which is a reliable indicator of popularity in GitHub [Borges et al., 2016; Silva and Valente, 2018] —and filtering by programming language. The resulting list of repositories was manually inspected to discard the ones that are not actual software projects, e.g., tutorials or code samples. Then, we forked each selected repository, to preserve their version histories from future changes pushed to the original project.

Table 4.5: JavaScript and C repositories used in the evaluation

Repository	Description	Commits
react	A declarative, efficient, and flexible JavaScript library for building user interfaces.	10,964
vue	Vue.js is a progressive, incrementally-adoptable JavaScript framework for building UI on the web.	3,014
d3	Bring data to life with SVG, Canvas and HTML.	4,148
react-native	A framework for building native apps with React.	16,875
angular.js	AngularJS - HTML enhanced for web apps.	8,963
create-react-app	Set up a modern web app by running one command.	2,233
jquery	A fast, small, and feature-rich JavaScript library.	6,403
atom	The hackable text editor.	36,752
axios	Promise based HTTP client for the browser and node.js.	847
three.js	JavaScript 3D library.	27,762
socket.io	Realtime application framework (Node.JS server).	1,715
redux	Predictable state container for JavaScript apps.	2,819
webpack	A bundler for javascript and friends.	7,852
Semantic-UI	Semantic is a UI component framework based around useful principles from natural language.	6,684
reveal.js	The HTML Presentation Framework.	2,341
meteor	Meteor, the JavaScript App Platform.	21,966
express	Fast, unopinionated, minimalist web framework for node.	5,555
material-ui	React components for faster and easier web development.	9,449
Chart.js	Simple HTML5 Charts using the canvas tag.	2,739
linux	Linux kernel source tree.	839,761
netdata	Real-time performance monitoring, done right!	8,338
redis	Redis is an in-memory database that persists on disk.	8,158
git	Git is a free and open-source distributed version control system.	55,723
ijkplayer	Android/iOS video player based on FFmpeg n3.4, with MediaCodec, VideoToolbox support.	2,584
php-src	The PHP Interpreter.	112,847
wrk	Modern HTTP benchmarking tool.	76
the_silver_searcher	A code-searching tool similar to ack, but faster.	2,016
emscripten	Emscripten: An LLVM-to-Web Compiler.	19,468
vim	The ubiquitous text editor.	9,875
jq	Command-line JSON processor.	1,287
FFmpeg	A complete, cross-platform solution to record, convert and stream audio and video.	93,898
tmux	A terminal multiplexer: it enables a number of terminals to controlled from a single screen.	7,618
nuklear	A single-header ANSI C gui library.	1,646
obs-studio	Free and open-source software for live streaming and screen recording.	6,727
libuv	Cross-platform asynchronous I/O.	4,319
swoole-src	Coroutine-based concurrency library for PHP (like Golang).	9,938
curl	A command line tool and library for transferring data with URL syntax.	24,339
toxcore	The future of online communications.	3,771
darknet	Convolutional Neural Networks.	436

Table 4.5 shows the name, short description, and number of commits of each selected repository, both for JavaScript and C.

2. We ran RefDiff in the version history of each repository. To select the commits, we navigate the commit graph backwards, starting from the most recent commit in the master branch. We also discarded merge commits, i.e., commits which have two predecessors. The rationale is that comparing a merge commit with their predecessors results in duplicated reports of refactorings applied in the commits prior to the merge operation. Moreover, to avoid over-representing projects with longer histories, we established a limit of 500 commits per repository. For each selected commit, we compared its source code with its predecessor using RefDiff, to detect refactoring operations.

3. Given the list of refactorings detected by RefDiff, we randomly selected 10 instances of each refactoring type to manually assess whether they correspond to actual refactorings (true positives), or incorrect reports (false positives). When applying the random selection, we enforced the constraint that we should not select two refactoring instances performed in the same commit. In this way, we avoid selecting similar refactorings which were performed in batch, e.g., multiple classes or functions moved together. To confirm each refactoring operation, we manually inspected the diff of the code changes in the corresponding commit.

After following the three steps, we compute precision as $P = TP/(TP + FP)$, where TP is the number of true positives and FP is the number of false positives.

4.5.2 Evaluation Design: Recall

To compute RefDiff's recall when detecting refactorings in JavaScript and C, we followed three steps:

1. We used GitHub API to find refactorings documented in commits from the repositories selected for evaluating precision (Section 4.5.1). Such queries consist in searching for keywords denoting a particular type of refactoring in the commit message, as described in Table 4.6. For example, when looking for *Rename Function* refactoring instances, we built a query that looks for commits which contain the keywords *rename* and *function* in their messages, among other combinations.

2. Given the list of results of a query, we manually inspected each item to assess whether it really contains a refactoring. We started by analyzing the commit

message. In many cases, the keywords are found in the text, but they are not referring to a refactoring operation. For example, one of the messages was: *"The routeToRegExp() function, introduced by 840b5f0, could not extract path params if the path contained question mark or hash."* This message contains the keywords *extract* and *function*, but clearly does not describe an *Extract Function* refactoring. In these situations, we discarded the commit with no further analysis. When the commit message described a refactoring, we checked the code diff to confirm it. Each confirmed refactoring was recorded in a normalized textual format compatible with the output of RefDiff. Inspecting the code diff is also necessary to locate the code elements involved in the operation. For example, the message *"Extract commit phase passes into separate functions"* documents a *Extract Function* refactoring, but does not specify the name of the extracted functions. We repeated this procedure until he found 10 instances of each refactoring type or when there were no more results to inspect. We found less than 10 instances when looking for *Move and Rename File*, *Move and Rename Function*, and *Inline Function* for JavaScript. Additionally, although modern JavaScript contains classes, we did not find documented refactorings instances of *Move and/or Rename Class*.

3. We ran RefDiff in the commits that contain documented and manually-validated refactorings to assess whether they are reported (true positives) or missed (false negatives).

After following these steps, we compute recall as $R = TP/(TP + FN)$, where TP is the number of true positives and FN is the number of false negatives.

Table 4.6: Search queries for each refactoring type

Change Signature	add parameter, remove parameter, add argument
Move/Rename File	move file, rename file, move folder, move rename file, move rename
Move/Rename Function	move function, rename function, move and rename
Extract Function	extract, duplicate, extract function, factor out
Inline Function	inline, inline into, remove indirection, indirect functions, remove wrapper

4.5.3 Results for JavaScript and C

Table 4.7 shows the precision and recall results for JavaScript. The overall precision is 91%. There are three refactorings with precision of 80%: *Rename Function*, *Move and Rename Function*, and *Inline Function*. For the remaining refactoring types, RefDiff has a precision of 90% (two refactoring types) or a precision of 100% (five refactoring types). Table 4.7 also shows the recall results, which reach 88% when all refactoring types are considered together. *Inline function* has the lowest recall (40%); however, our dataset has only five instances of this operation. There are three refactoring types with recall of 100%: *Move File*, *Move Function*, *Move and Rename File*. For the other ones, recall ranges between 80% and 90%.

Table 4.7: JavaScript precision and recall results

Refactoring Type	#	Precision	#	Recall
Move File	10	1.00	10	1.00
Move Class	2	1.00	0	
Move Function	10	0.90	10	1.00
Rename File	10	1.00	10	0.80
Rename Class	5	1.00	0	
Rename Function	10	0.80	10	0.90
Move and Rename File	10	1.00	3	1.00
Move and Rename Function	10	0.80	7	0.86
Extract Function	10	0.90	10	0.90
Inline Function	10	0.80	5	0.40
Total	87	0.91	65	0.88

Table 4.8 shows the precision and recall results for C. The overall precision is 88%. *Inline Function* is the refatoring for which precision is lower (50%). Besides, there are two refactorings with precision of 80%: *Move Function* and *Move and Rename Function*. For the remaining refactoring types, RefDiff has a precision of 90% (one refactoring type) or a precision of 100% (four refactoring types). We did not find any instance of *Move and Rename File*, thus we could not compute precision for this refactoring type. Table 4.8 also shows the recall results, which is 91% overall. *Extract Function* (70%) and *Move Function* (80%) are the ones with lowest recall. For the remaining refactoring types, RefDiff has a recall of 90% (three refactoring types) or a recall of 100% (four refactoring types).

Table 4.8: C precision and recall results

Refactoring Type	#	Precision	#	Recall
Change Signature	10	1.00	10	0.90
Move File	10	1.00	10	1.00
Move Function	10	0.80	10	0.80
Rename File	10	1.00	10	1.00
Rename Function	10	0.90	10	1.00
Move and Rename File	0		10	1.00
Move and Rename Function	10	0.80	10	0.90
Extract Function	10	1.00	10	0.70
Inline Function	10	0.50	10	0.90
Total	80	0.88	90	0.91

4.5.4 Threats to Validity

One threat to validity of the evaluation with JavaScript and C projects is its smaller scale. For example, we computed precision for JavaScript using 87 refactorings, and recall using 65 refactorings, while the evaluation in Java used an oracle with 3,249 refactorings. Particularly, we restricted the analysis to 10 instances per refactoring type. First, we acknowledge this limit does not express the frequency of each refactoring in practice. Second, as a result of this decision, our evaluation dataset for JavaScript and C is not complete with respect to true positives. In fact, the evaluation with JavaScript and C is a complement to the evaluation with Java aiming to show that RefDiff 2.0 provides similar results when used to detect refactorings in other programming languages. Last, we should also note that, similarly to the evaluation with Java projects, the subjectivity inherent to the manual classification of the reported refactorings is also a threat to validity.

4.6 Challenges and limitations

Low-level refactorings: RefDiff does not detect local refactorings, such as rename, extract or inline a local variable, because the syntactical structure of the source code within a CST node is not represented. While it is theoretically possible to extend the CST to include finer-grained code elements such as statements, local variables, and others, this would also make it harder to port RefDiff to other programming languages.
Generating call graphs: In our modular architecture, the generation of the CST, which includes information from a call graph and a type hierarchy graph, is delegated to a language-specific plugin. For languages such as Java, there are reliable parsers and

static analyzers that aid in this task (e.g., Eclipse JDT). However, we acknowledge that generating precise call graphs for untyped languages, such as JavaScript, might be a challenging problem. Nevertheless, we provided evidences that our approach works well even when the information encoded in the CST is not completely precise. For example, in our JavaScript implementation—which contains only 689 lines of code in total—we used a simple strategy in which we assume a node n_1 uses n_2 if n_1 contains a function call with the same identifier as n_2 and both are defined in the same file. However, to detect a *Extract* relationship between n_1 and n_2, we need two other conditions: (i) n_2 should be a new method and (ii) the body of n_2 should be similar to the code removed from n_1 between revisions. In other words, an incorrect edge in the call graph only leads to an incorrect *Extract* relationship in the unlikely scenario in which a function n_2 is introduced, the content of such function is similar to code removed from n_1 and n_1 calls a function with the same identifier of n_2 after the change, but that function is not actually n_2. A similar reasoning applies to *Inline* relationships, which also depends on information from call graphs. In summary, although generating precise call graphs is non trivial for untyped languages, we argue that it is not needed in practice to achieve acceptable precision, specially in the light of the results of our evaluation using JavaScript systems (91% of precision).

JavaScript class syntax: Our JavaScript implementation only considers classes defined with the new ES6 syntax, i.e., classes emulated by functions definitions and prototype-based inheritance are just treated like regular functions when generating the CST.

Field-related refactorings: As our refactoring detection algorithm is centered around code similarity and fields do not have a body, we did not implement the detection of *Move/Pull Up/Push Down Field* in RefDiff 2.0. Unrestricted detection of *Move Field* based solely on fields' types and names is prone to find many false positives. However, we plan to add support for field-related refactorings in future work by using stricter detection rules, similarly to RMiner (e.g., requiring a dependency between their source/destination classes).

4.7 Conclusion

To the best of our knowledge, RefDiff 2.0 is the first refactoring detection approach that supports multiple programming languages. We made this possible with two main design decisions. First, our refactoring detection algorithm relies only on information encoded in CSTs, a data structure that represents the source code but abstracts the

specificities of each programming language. Second, we compute code similarity at the level of the tokenized source code, using techniques from information retrieval. In summary, RefDiff is loosely coupled to the syntax of the target programming language, which makes it easier to extend it to other languages. Our evaluation using a dataset of real refactorings in Java showed that RefDiff's precision is 96.4% and recall is 80.4%. Although we were not able to surpass RMiner's precision of 98.8%, we argue that we achieved satisfactory results for a language-neutral approach. In one hand, specialized tools can use more advanced techniques to improve refactoring detection. On the other hand, the higher the coupling with the syntax of a particular language, the harder it becomes to port the approach to other programming languages. Last, our evaluation in JavaScript and C also showed promising results. RefDiff's precision and recall are respectively 91% and 88% for JavaScript, and 88% and 91% for C. These results show the viability our approach for languages other than Java. Thus, we claim that RefDiff 2.0 can pave the way for important advances in refactoring studies in JavaScript, C, and other languages in the future. Moreover, it can be employed in practical tasks, such as improving diff visualization, automatically documenting refactorings in commits, keeping track of the history of refactored code elements, and others.

Chapter 5

Practical Applications of Refactoring Detection

Throughout this thesis we discussed tools to mine refactorings from version histories and empirical studies that benefit from such techniques. In this chapter, we discuss three practical applications of such tools. We do not intend to thoroughly discuss all possible applications, but rather to exemplify that some relevant problems faced by development teams can be tackled with the help of such tools. Particularly, RefDiff's multi-language design is an important advantage for the practical applications described in this chapter, as software projects are developed using different programming languages.

5.1 Refactoring-aware Diff

One of the challenges developers face when refactoring, as reported in an in-depth study by Kim et al. [2014], is the difficulty of reviewing code after refactoring. This is illustrated by the following comment from one of the developers interviewed by Kim et al.:

"It (refactoring) typically increases the number of lines/files involved in a check-in. That burdens code reviewers and increases the odds that your change will collide with someone else's change."

In fact, version control systems are usually sensitive to rename and move refactoring, which makes it hard for developers to understand code changes, specially when refactorings are applied interleaved with other modifications in the system.

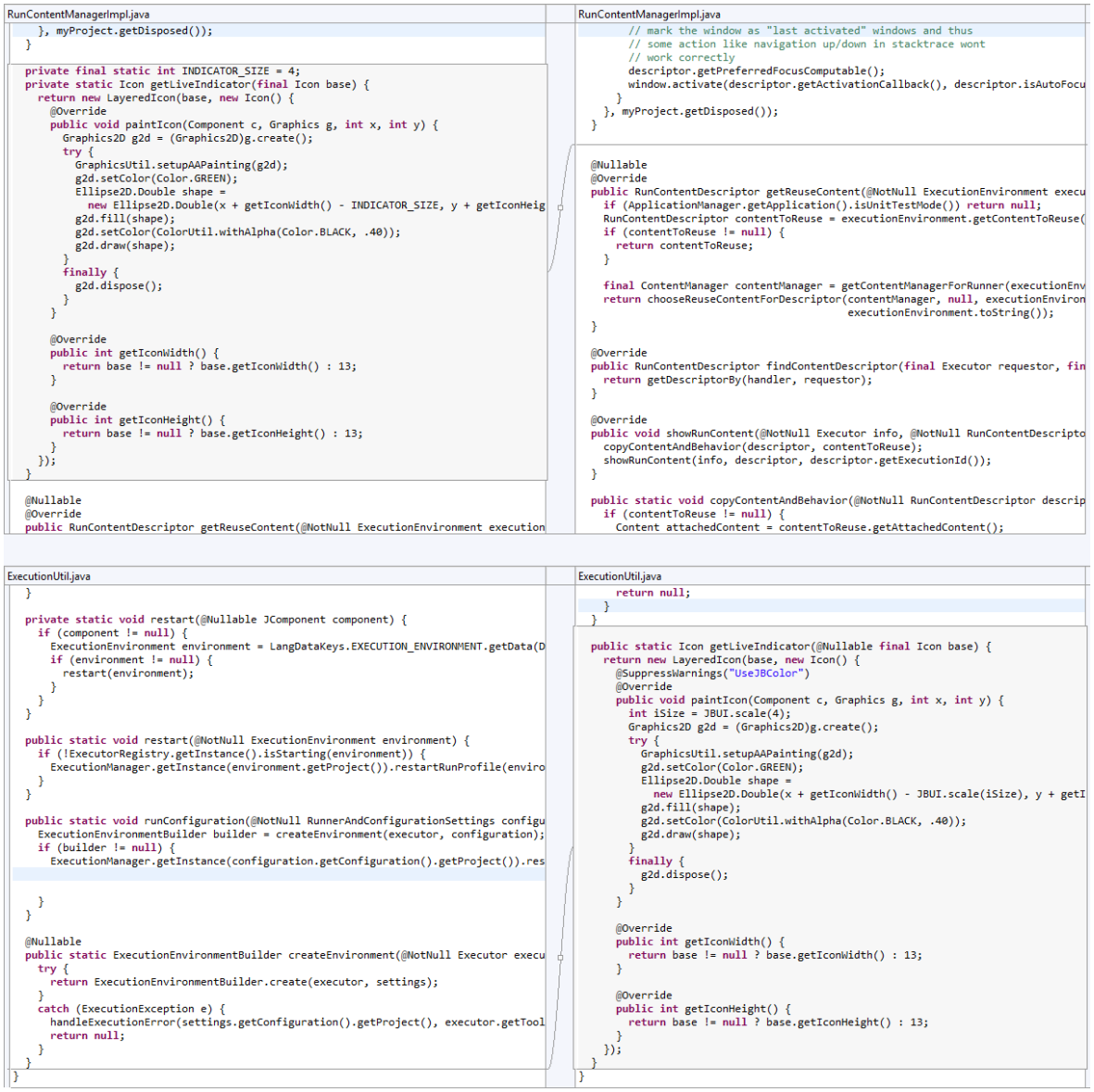


Figure 5.1: Typical diff visualization of a code change containing a moved method

As a concrete example, consider the code changes presented in Figure 5.1, which are taken from the `intellij-community` repository. In this figure, we present the code diff between two changed files. In the first one, `RunContentManagerImpl.java`, a large code block was removed, whilst in the second file (`ExecutionUtil.java`) a large code block was added. With a thoroughly analysis, we can note that the code removed from `RunContentManagerImpl.java` was actually moved to `ExecutionUtil.java`, i.e., the method `getLiveIndicator` was moved. However, this is not immediately obvious to a developer reviewing these changes, specially considering that there might be many other changed files in the commit. Moreover, although one can identify with some effort that the method `getLiveIndicator` has been moved, it will be much more difficult to identify small changes made to its body in this visualization.

In contrast, consider the diff visualization presented in Figure 5.2. In this case, the diff is focused in the method that was moved, `getLiveIndicator`, before and after the change. We can clearly see that some changes have been made to the body of `getLiveIndicator`. Specifically, the annotations `@Nullable` and `@SuppressWarnings` were introduced, a new local variable `iSize` was declared, and the expression passed as the first argument to the `Ellipse2D.Double` constructor was changed. All these changes are very hard to note in a traditional diff visualization, such as the one from Figure 5.1, but are easily discernible when we present the moved method compared with its counterpart. Other refactoring types might benefit from such idea as well. As a second example, it would also be convenient to highlight the differences between an extracted method and its originating code. This way, it would become easier to identify new statements added. A similar reasoning can be applied

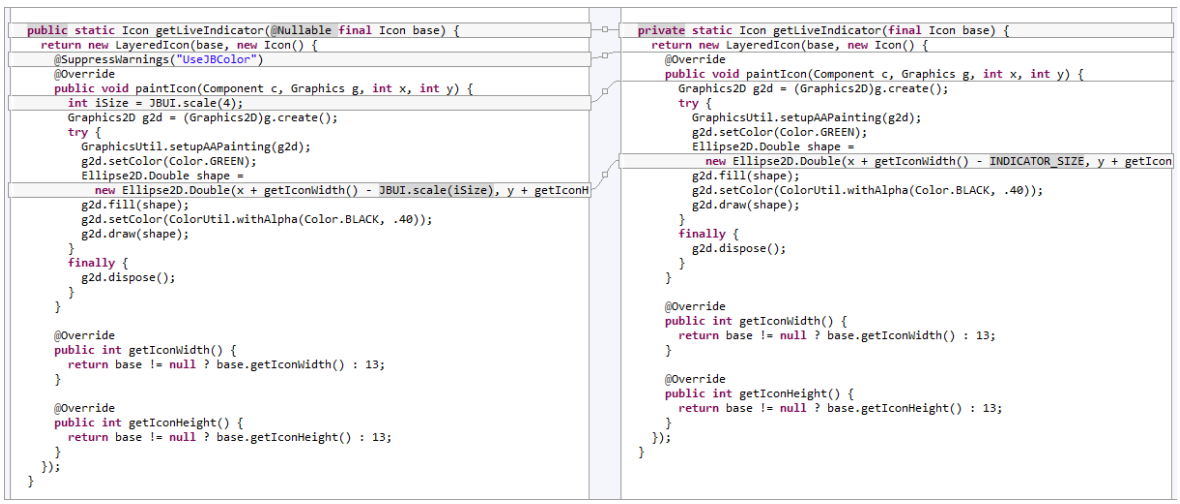


Figure 5.2: Diff visualization focused on the moved method

to Move Class, Inline Method, and other refactoring types.

Thus, we can improve diff visualization using the information about the refactorings detected in commits, using a tool such as RefDiff. We expect that such refactoring-aware diff visualization solution would improve developers' ability to discern and understand code changes in the presence of refactorings, enabling them to concentrate on behavioral changes rather than on code modifications resulting from refactorings.

5.2 Tracking changes of a code element

Although refactoring is very important to maintain a software system, it can make it harder to track code changes in the history, specially at the method/function level. For example, consider the `getReactRootElementInContainer` function from React project, depicted in Figure 5.3. Suppose one is investigating the history of that function to find out who introduced the conditional statement in lines 85–89. Developers typically use the *git-blame* tool for such tasks, as it shows what revision and author last modified each line of code of a file. However, although the *git-blame* output shows that Developer A was the last developer who modified the lines of code within `getReactRootElementInContainer`, this information may mislead one to think that he is the author of the function. In fact, a deeper anal-

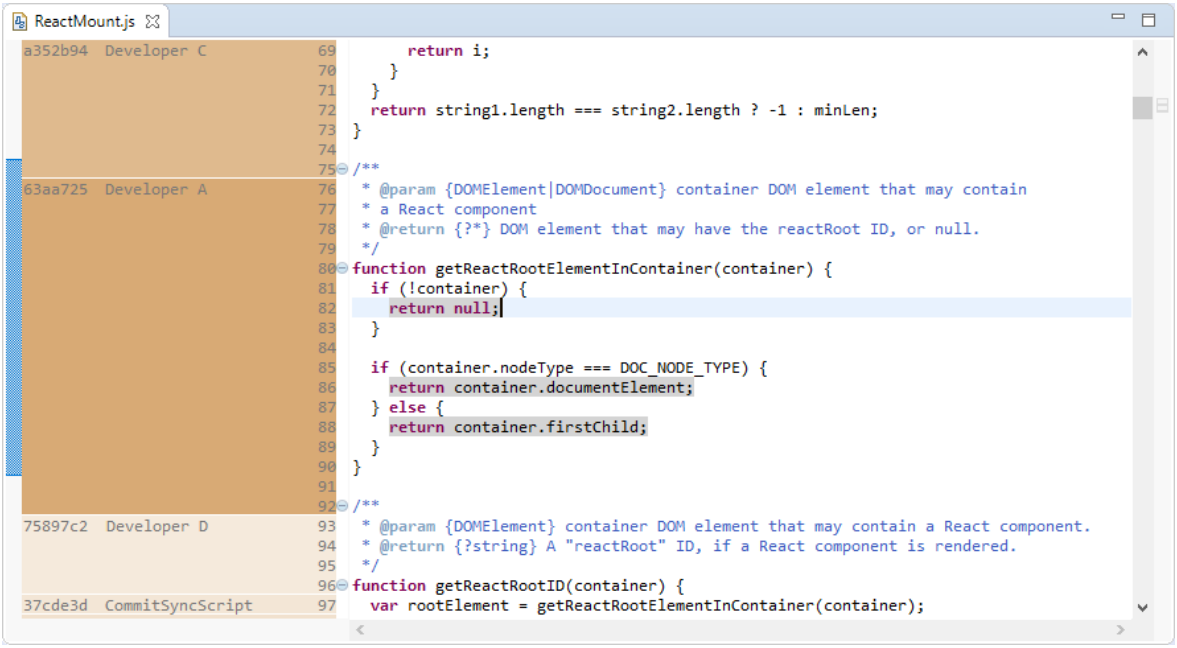


Figure 5.3: *git-blame* visualization of the `ReactMount.js` file from React project, showing that function `getReactRootElementInContainer` was last modified by Developer A

ysis of the history reveals that Developer A actually moved the function from file `getReactRootElementInContainer.js` to `ReactMount.js`. Thus, to find the actual author of those lines of code, one should inspect the history of changes of the original file of the function. Figure 5.4 shows that the conditional in lines 85–89 was actually introduced by Developer B.

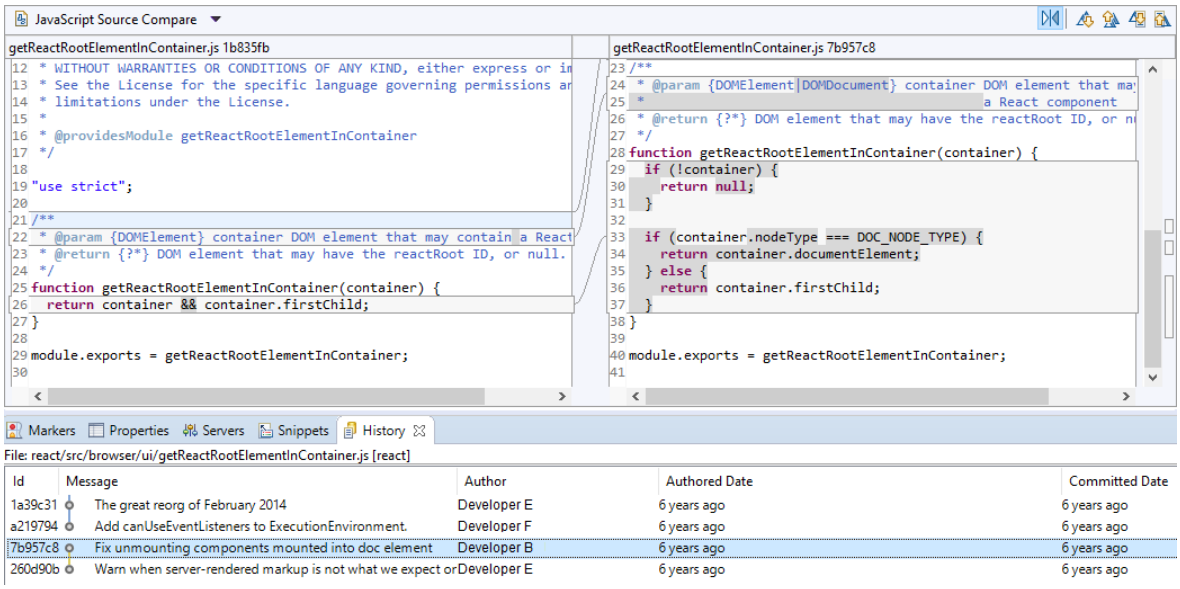


Figure 5.4: Diff visualization of the commit in witch the conditional inside function `getReactRootElementInContainer` was introduced, showing that the actual author is Developer B

In summary, in the example above, the *Move Function* refactoring split the history of `getReactRootElementInContainer`, making it harder to trace back the author of each line of code. The frequency in which such situations occur makes this problem even more important. Hora et al. [2018] studied the extent in which MSR approaches that relies on tracking the changes along all versions of each individual methods (or classes) are affected by refactorings, i.e., a method rename or move can be misinterpreted as the disappearance of a method and the appearance of a brand new one. The authors found that between 10% and 21% of method-level changes and 2% and 15% of class-level changes may introduce a discontinuity in their histories. Moreover, 25% of the code elements have at least one discontinuity in their histories.

For these reasons, this problem is a potential application of refactoring detection techniques. If we know the refactorings applied in the system, we can keep track of all moves/renames applied to each code element and reconstruct its full history. Moreover, this would enable us to provide an improved *git-blame* that shows the last modification of each line of code but is not susceptible to code movement.

5.3 Resolving merge conflicts

Another challenge developers face when refactoring, also reported by Kim et al. [2014], is the difficulty of merging code. In software projects, usually several developers work in parallel fixing bugs and adding features to the system. Thus, the different changes introduced by each developer should be integrated together in the same code base, in a process called as merging code. In many cases, merging can be done automatically by version control systems, which provide algorithms for such task. However, when different changes are made to the same lines of code, these algorithms report a conflict, and merge must be done manually. High-level refactoring operations amplify the odds that merge conflict occurs, because they usually involve movement of large chunks of code. In fact, a recent study on the relationship between refactoring and merge conflicts shows that refactoring operations are involved in at least 22% of merge conflicts [Mahmoudi et al., 2019].

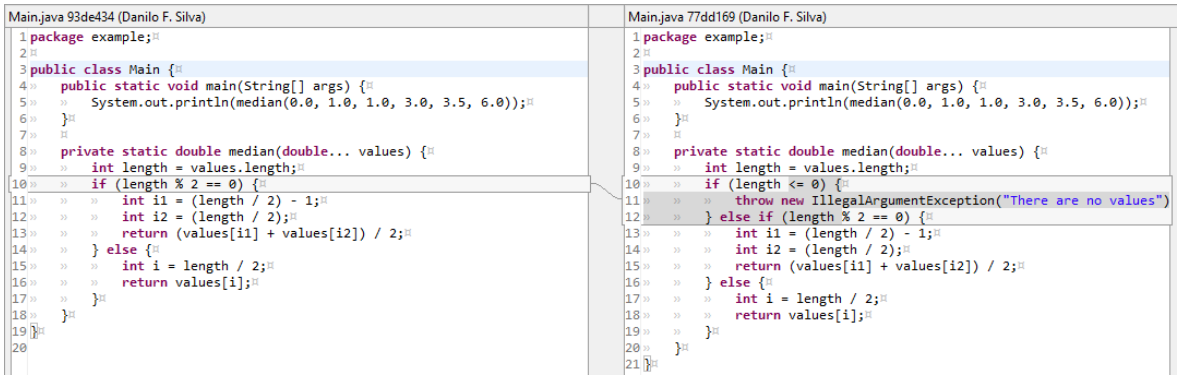


Figure 5.5: Hypotetical change made by first developer: Test if `length` is zero

We can illustrate such issues with the following scenario. Consider the example code diff in Figure 5.5, in which a first developer modified method `median` from class `Main`, adding a clause that throws an exception when `length` is zero. Now, suppose that a second developer, working in parallel, decides to move method `median` from class `Main` to a new class `Statistics`, such as depicted in Figure 5.6. Note that the first developer modified line 10 from file `Main.java`, while the second developer modified/deleted lines 5–17 from the same file. Thus, when we merge both changes, we will get a conflict, as depicted in Figure 5.7. In this case, to resolve the merge conflict, one must apply the same logic introduced by the first developer to the `median` method that is now in class `Statistics`, and accept the left side of the code from Figure 5.7 in `Main.java`. Note that, even in that small hypothetical example, resolving merge conflicts is tricky and error-prone. Thus, large-scale refactorings have the potential to make merging changes extremely complicated.

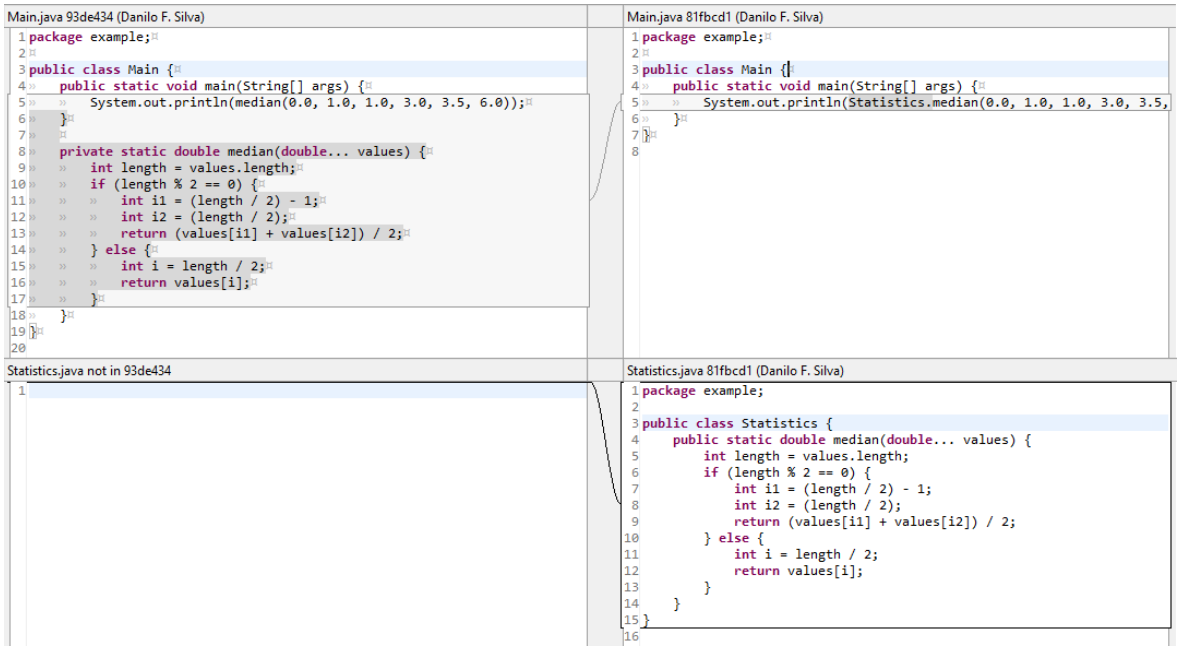


Figure 5.6: Hypotetical change made by second developer: Move method `median` from class `Main` to class `Statistics`

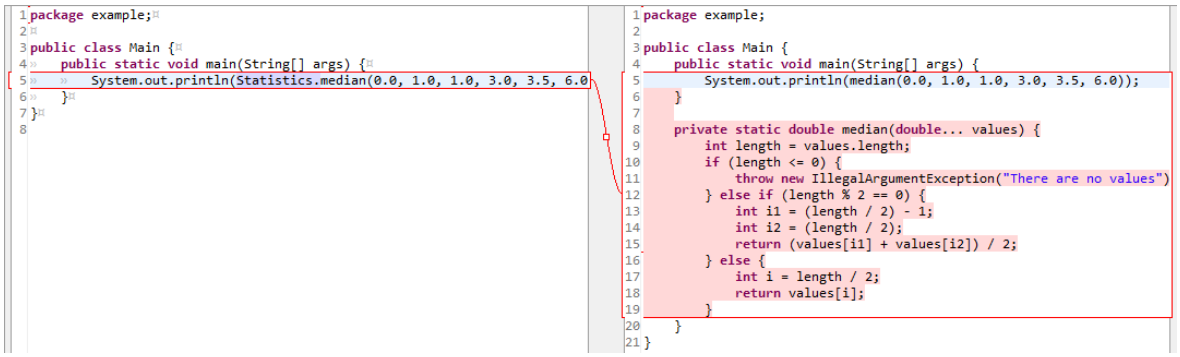


Figure 5.7: When we merge the changes from figures 5.5 and 5.6 we get a conflict

However, we can improve merging algorithms if we know the refactorings applied to the code. For example, Dig et al. [2008] proposed a refactoring-aware version control system that is able to resolve merge conflicts caused by refactorings using the following strategy: it identifies the refactorings applied between revisions, undo the refactorings, apply regular textual merge algorithms, and finally apply the refactorings again. As another example, Cavalcanti et al. [2017] propose an improvement in semistructured merge algorithms that relies on the Levenshtein distance to find renamed code elements. This way, it avoids reporting merge conflicts when a developer renames a method whose body is changed by other developer in parallel. Shen et al. [2019] also propose a refactoring-aware merging technique based on a semistructured representation of the source code. Their tool, called IntelliMerge, first transforms the source code of

each revision involved in the merge process into Program Element Graphs (PEGs). Then, nodes from these graphs are matched with their base version, i.e., in this step refactorings such as *Move* and *Rename* are found. Finally, textual merge is applied individually for each node, considering the matchings found in the previous step. The aforementioned tools depend on finding refactorings applied between revision. Thus, they are another direct application of refactoring detection approaches such as RefDiff.

Chapter 6

Conclusion

This chapter concludes the thesis by listing the main contributions of this work. Next, we present directions for future research.

6.1 Contributions

We summarize our main contributions as follows:

1. In Study 1, we found that code reuse is one of the main motivations for applying Extract Method refactoring (56.9% of the cases). This was an important finding for two main reasons. First, the relationship between Extract Method and code reuse, using actual refactorings mined from code repositories, had not been studied before. Second, although refactoring literature emphasizes the removal of code smells as motivations for refactorings, such finding reveals that it is incorrect to assume that this is the sole reason to refactor source code.

2. In Study 2, we compiled a catalogue of 44 distinct motivations for 12 well-known refactoring types, based on the actual explanations of developers on specific refactorings they have recently applied. Some of the motivations we found are the resolution of well-known code smells, but many others are related to code evolution, facilitating the implementation of a feature or a bug fix. We also investigated the frequency of each refactoring type and the usage of refactoring tools, confirming previous findings, and how the IDE affects refactoring tools usage. The findings of this study increased our knowledge about refactoring practice and offered important insights to researchers, practitioners, and refactoring tools builders.

3. We proposed RefDiff, a novel refactoring detection approach suitable to mine refactoring in version histories in large scale with high precision and recall, with the advantage of supporting multiple programming languages. When initially proposed, our approach improved precision and recall over existing approaches. Later, we extended RefDiff introducing a language agnostic core algorithm and improved its precision to be on par with RMiner, the current state-of-the-art in Java refactoring detection. Taking advantage of RefDiff's extensible architecture, we implemented plugins for three programming languages (Java, JavaScript, and C), and evaluated our approach in each of them. Our tool is publicly available at GitHub (`https://github.com/aserg-ufmg/RefDiff`), along with usage instructions.

4. We contributed to the creation of the largest refactoring dataset to date, which keeps record of 3,248 refactorings mined from 538 commits of 185 Java repositories. Such dataset initially contained the data we collected in Study 2, and was later extended by Tsantalis et al. [2018], while evaluating RMiner. Finally, in our evaluation of RefDiff, we extended it once again with additional refactorings. This dataset serves as a reliable oracle that can be used to evaluate precision and recall of refactoring detection approaches, facilitating future research. The data is also publicly available at RefDiff's GitHub repository.

6.2 Future Work

The work developed throughout this thesis opens different research paths. First, there is still room for empirical studies on refactoring practice. Given the availability of reliable refactoring mining techniques, larger scale studies can be conducted to confirm previous findings and to investigate other research questions. In particular, there is much to learn about refactoring in programming languages other than Java. For example, JavaScript is an interesting case study because of its distinct characteristics. We suspect that refactoring practice in a dynamically-typed, interpreted language, might differ significantly from compiled languages, because developers should carefully consider the risk of introducing defects. Without compile-time type checking, renaming a function that is used across several files is much more dangerous. Thus, developers may avoid broader refactorings in favor of more localized modifications.

Besides empirical studies, there are potential practical problems worth exploring, such as the ones already discussed em Chapter 5. In particular, refactoring-aware diff visualizations are little explored by the current literature. We believe that such kind of

tool might increase developers' productivity, specially in the context of code reviewing. It is worth noting that the three problems discussed in Chapter 5—*Refactoring-aware diff*, *Tracking changes of a code element*, and *Resolving merge conflicts*—affect any programming language. Thus, the multi-language support provided by RefDiff is an important advantage for these applications.

Last, refactoring detection approaches can be further improved. In particular, although RefDiff was thoroughly evaluated for Java, the JavaScript and C evaluation have a much smaller scale, due to the lack of refactoring oracles in these languages. Thus, a larger scale JavaScript and C evaluation could be conducted, which would also open margin for improvements in precision and recall. The evaluation data could also be used as a starting refactoring oracle for future competing tools. Additionally, RefDiff could be extended to support popular programming languages such as Python, C#, and others.

Bibliography

Bavota, G., Carluccio, B., Lucia, A., Penta, M., Oliveto, R., and Strollo, O. (2012). When does a refactoring induce bugs? an empirical study. In *12th International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 104–113.

Bavota, G., Lucia, A., Marcus, A., and Oliveto, R. (2014a). Recommending refactoring operations in large software systems. In Robillard, M. P., Maalej, W., Walker, R. J., and Zimmermann, T., editors, *Recommendation Systems in Software Engineering*, pages 387–419. Springer Berlin Heidelberg.

Bavota, G., Lucia, A., and Oliveto, R. (2011). Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, 84(3):397–414.

Bavota, G., Oliveto, R., Gethers, M., Poshyvanyk, D., and Lucia, A. (2014b). Methodbook: Recommending Move Method refactorings via relational topic models. *IEEE Transactions on Software Engineering*, 40(7):671–694.

Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

Beck, K. (2003). *Test-driven development: by example*. Addison-Wesley Professional.

Bois, B., Demeyer, S., and Verelst, J. (2005). Does the "refactor to understand" reverse engineering pattern improve program comprehension? In *9th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 334–343.

Borges, H., Hora, A., and Valente, M. T. (2016). Understanding the factors that impact the popularity of GitHub repositories. In *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344.

Cavalcanti, G., Borba, P., and Accioly, P. (2017). Evaluating and improving semistructured merge. *Proceedings of the ACM on Programming Languages*, pages 1–27.

Chaparro, O., Bavota, G., Marcus, A., and Penta, M. (2014). On the impact of refactoring operations on code quality metrics. In *30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 456–460.

Chierichetti, F., Kumar, R., Pandey, S., and Vassilvitskii, S. (2010). Finding the jaccard median. In *21st annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 293–311.

Cruzes, D. S. and Dyba, T. (2011). Recommended steps for thematic synthesis in software engineering. In *5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 275–284.

Demeyer, S., Ducasse, S., and Nierstrasz, O. (2000). Finding refactorings via change metrics. In *2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*, pages 166--177.

Dias, M., Bacchelli, A., Gousios, G., Cassou, D., and Ducasse, S. (2015). Untangling fine-grained code changes. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 341–350.

Dig, D., Comertoglu, C., Marinov, D., and Johnson, R. (2006). Automated detection of refactorings in evolving components. In *20th European Conference on Object-Oriented Programming (ECOOP)*, pages 404–428.

Dig, D., Manzoor, K., Johnson, R. E., and Nguyen, T. N. (2008). Effective software merging in the presence of object-oriented refactorings. *IEEE Transactions on Software Engineering*, 34(3):321–335.

Dit, B., Revelle, M., Gethers, M., and Poshyvanyk, D. (2013). Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95.

Ferreira, I., Fernandes, E., Cedrim, D., Uchôa, A., Bibiano, A. C., Garcia, A., Correia, J. L., Santos, F., Nunes, G., Barbosa, C., et al. (2018). The buggy side of code refactoring: Understanding the relationship between refactorings and bugs. In *40th International Conference on Software Engineering (ICSE): Companion Proceeedings*, pages 406–407.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Henkel, J. and Diwan, A. (2005). Catchup!: capturing and replaying refactorings to support API evolution. In *27th International Conference on Software Engineering (ICSE)*, pages 274–283.

Hill, E. and Black, A. P. (2008). Breaking the barriers to successful refactoring: Observations and tools for Extract Method. In *30th International Conference on Software Engineering (ICSE)*, pages 421–430.

Hill, E., Zimmermann, T., Bird, C., and Nagappan, N. (2015). The design space of bug fixes and how developers navigate it. *IEEE Transactions on Software Engineering*, 41(1):65–81.

Hintze, J. L. and Nelson, R. D. (1998). Violin plots: A box plot-density trace synergism. *The American Statistician*, 52(2):181–184.

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

Hotta, K., Higo, Y., and Kusumoto, S. (2012). Identifying, tailoring, and suggesting Form Template Method refactoring opportunities with program dependence graph. In *16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 53–62.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2014). The promises and perils of mining GitHub. In *11th Working Conference on Mining Software Repositories (MSR)*, pages 92–101.

Kawrykow, D. and Robillard, M. P. (2011). Non-essential changes in version histories. In *33rd International Conference on Software Engineering (ICSE)*, pages 351–360.

Kim, M., Cai, D., and Kim, S. (2011). An empirical investigation into the role of API-level refactorings during software evolution. In *33rd International Conference on Software Engineering (ICSE)*, pages 151–160.

Kim, M., Gee, M., Loh, A., and Rachatasumrit, N. (2010). Ref-Finder: A refactoring reconstruction tool based on logic query templates. In *18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 371–372.

Kim, M., Zimmermann, T., and Nagappan, N. (2012). A field study of refactoring challenges and benefits. In *20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 50:1–50:11.

Kim, M., Zimmermann, T., and Nagappan, N. (2014). An empirical study of refactoring challenges and benefits at Microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649.

Mahmoudi, M., Nadi, S., and Tsantalis, N. (2019). Are refactorings to blame? an empirical study of refactorings in merge conflicts. In *26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 151–162.

Meng, N., Hua, L., Kim, M., and McKinley, K. S. (2015). Does automated refactoring obviate systematic editing? In *37th International Conference on Software Engineering (ICSE)*, pages 392–402.

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

Murphy, G. C., Kersten, M., and Findlater, L. (2006). How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83.

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

Negara, S., Chen, N., Vakilian, M., Johnson, R. E., and Dig, D. (2013). A comparative study of manual and automated refactorings. In *27th European Conference on Object-Oriented Programming (ECOOP)*, pages 552–576.

Negara, S., Codoban, M., Dig, D., and Johnson, R. E. (2014). Mining fine-grained code changes to detect unknown change patterns. In *36th International Conference on Software Engineering (ICSE)*, pages 803–813.

Oliveira, J., Gheyi, R., Mongiovi, M., Soares, G., Ribeiro, M., and Garcia, A. (2019). Revisiting the refactoring mechanics. *Information and Software Technology*, 110:136–138.

Opdyke, W. F. (1992). *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA.

Prete, K., Rachatasumrit, N., Sudan, N., and Kim, M. (2010). Template-based reconstruction of complex refactorings. In *26th International Conference on Software Maintenance (ICSM)*, pages 1–10.

Rachatasumrit, N. and Kim, M. (2012). An empirical investigation into the impact of refactoring on regression testing. In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 357–366.

Ratzinger, J., Sigmund, T., and Gall, H. C. (2008). On the relation of refactorings and software defect prediction. In *5th International Working Conference on Mining Software Repositories (MSR)*, pages 35–38.

Sales, V., Terra, R., Miranda, L., and Valente, M. (2013). Recommending Move Method refactorings using dependency sets. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 232–241.

Salton, G. and McGill, M. (1984). *Introduction to Modern Information Retrieval*. McGraw-Hill.

Shen, B., Zhang, W., Zhao, H., Liang, G., Jin, Z., and Wang, Q. (2019). Intellimerge: a refactoring-aware software merging technique. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):170.

Silva, D., Terra, R., and Valente, M. T. (2014). Recommending automated Extract Method refactorings. In *22nd International Conference on Program Comprehension (ICPC)*, pages 146–156.

Silva, D., Tsantalis, N., and Valente, M. T. (2016). Why we refactor? confessions of GitHub contributors. In *24th International Symposium on the Foundations of Software Engineering (FSE)*, pages 858–870.

Silva, D. and Valente, M. T. (2017). RefDiff: Detecting refactorings in version histories. In *14th International Conference on Mining Software Repositories (MSR)*, pages 1--11.

Silva, H. and Valente, M. T. (2018). What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129.

Singer, J., Sim, S. E., and Lethbridge, T. C. (2008). *Guide to Advanced Empirical Software Engineering*, chapter Software Engineering Data Collection for Field Studies, pages 9–34. Springer London, London.

Soares, G., Gheyi, R., Hill, E., and Johnson, B. (2013). Comparing approaches to analyze refactoring activity on software repositories. *Journal of Systems and Software*, 86(4):1006–1022.

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

Tairas, R. and Gray, J. (2012). Increasing clone maintenance support by unifying clone detection and refactoring activities. *Information and Software Technology*, 54(12):1297–1307.

Tsantalis, N. and Chatzigeorgiou, A. (2009). Identification of Move Method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367.

Tsantalis, N. and Chatzigeorgiou, A. (2011). Identification of Extract Method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782.

Tsantalis, N., Guana, V., Stroulia, E., and Hindle, A. (2013). A multidimensional empirical study on refactoring activity. In *2013 Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, pages 132–146.

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., and Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In *40th International Conference on Software Engineering (ICSE)*, pages 483--494.

Tsantalis, N., Mazinanian, D., and Krishnan, G. (2015). Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering*, 41(11):1055–1090.

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

Wang, Y. (2009). What motivate software engineers to refactor source code? evidences from professional developers. In *25th IEEE International Conference on Software Maintenance (ICSM)*, pages 413–416.

Weißgerber, P. and Diehl, S. (2006a). Are refactorings less error-prone than other changes? In *3rd Workshop on Mining Software Repositories (MSR)*, pages 112–118.

Weißgerber, P. and Diehl, S. (2006b). Identifying refactorings from source-code changes. In *21st International Conference on Automated Software Engineering (ASE)*, pages 231–240.

Xing, Z. and Stroulia, E. (2005). UMLDiff: An algorithm for object-oriented design differencing. In *20th International Conference on Automated Software Engineering (ASE)*, pages 54–65.

Xing, Z. and Stroulia, E. (2008). The JDEvAn tool suite in support of object-oriented evolutionary development. In *30th International Conference on Software Engineering (ICSE)*, pages 951–952.