Pontifícia Universidade Católica
do Rio de Janeiro

Daniel Tenorio Martins de Oliveira

On the Identification and Analysis of Refactoring-related Modifications

Tese de Doutorado

Thesis presented to the Programa de Pós-graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática.

Advisor : Prof. Alessandro Fabricio Garcia
Co-advisor: Prof. Wesley Klewerton Guez Assunção

Rio de Janeiro
September 2024

Pontifícia Universidade Católica
do Rio de Janeiro

Daniel Tenorio Martins de Oliveira

On the Identification and Analysis of Refactoring-related Modifications

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Informática. Approved by the Examination Committee:

Prof. Alessandro Fabricio Garcia
Advisor
Departamento de Informática – PUC-Rio

Prof. Wesley Klewerton Guez Assunção
North Carolina State University - NCSU

Prof.ª Juliana Alves Pereira
Departamento de Informática – PUC-Rio

Prof José Alberto Rodrigues Pereira Sardinha
Departamento de Informática – PUC-Rio

Prof. Baldoino Fonseca dos Santos Neto
Universidade Federal de Alagoas - UFAL

Prof. Rafael Maiani de Mello
Universidade Federal do Rio de Janeiro - UFRJ

Rio de Janeiro, September 24th, 2024

Daniel Tenorio Martins de Oliveira

I am a PhD student in Computer Science at Pontifical Catholic University of Rio de Janeiro, and a part-time software engineer at Tecgraf Institute. My research focuses on providing developers with automated support for code smell detection and refactorings. Since my graduation, I always seek to collaborate in international projects in Software Engineering, having already been involved in projects with foreign universities, such as the University of Coimbra, the University of Florence and University College London. Additionally, I worked on research projects in partnership with companies such as BlackBerry. Finally, I have co-authored papers accepted in prestigious conferences including ICSE, EMSE, MSR, ICPC, CHASE, and ESEM.

Bibliographic data

Oliveira, Daniel Tenorio Martins de

On the Identification and Analysis of Refactoring-related Modifications / Daniel Tenorio Martins de Oliveira; advisor: Alessandro Fabricio Garcia; co-advisor: Wesley Klewerton Guez Assunção. – 2024.

111 f: il. color. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2024.

Inclui bibliografia

1. Informática – Teses. 2. Engenharia de software – Teses. 3. Refatoração. 4. Manutenibilidade. 5. Customização de refatoraçãoo. I. Garcia, Alessandro Fabricio. II. Assunção, Wesley Klewerton Guez. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

To my parents, brothers, family, and friends
for their support and encouragement.

Acknowledgments

I deeply thank everyone who contributed to the completion of this thesis. To my advisors, for their invaluable support and guidance; to my colleagues and friends, for their continuous collaboration and encouragement; and to my family, for their love and unconditional support. Without the effort and dedication of each one, this work would not have been possible.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Abstract

Oliveira, Daniel Tenorio Martins de; Garcia, Alessandro Fabricio (Advisor); Assunção, Wesley Klewerton Guez (Co-Advisor). **On the Identification and Analysis of Refactoring-related Modifications**. Rio de Janeiro, 2024. 111p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Refactoring is a well-established software engineering technique aimed at improving code structure without altering its behavior. Each refactoring consists of a set of default modifications in a program. A thorough analysis of what code modifications compose a refactoring is a prerequisite to reap the benefits of this technique. However, there are at least two characteristics of code refactoring in real-life software projects that complicates a thorough code analysis. First, refactorings are often applied in a *customized* fashion, i.e., developers manually tailor a pre-defined set of code modifications (associated with a *refactoring type*) by adding or removing modifications to suit specific contexts. Second, refactorings are often intertwined with other tasks like adding features or fixing bugs, known as *floss refactoring*. Since refactoring modifications are often performed with other unrelated modifications in the same commit, distinguishing them is time-consuming and error-prone. While previous research has scrutinized these two challenging characteristics of code refactoring, specialized tool support for its thorough analysis is still limited. This thesis aimed at developing a comprehensive approach to assist code refactoring analysis in the presence of these two challenging characteristics. To this end, we first performed a study with developers to understand whether they would require specialized tool support for customized refactorings. Then, we developed two tools to assist developers in identifying refactoring-related modifications and distinguishing them from other unrelated modifications. The effectiveness of these tools was assessed through a user study with experienced developers.

Keywords

Refactoring; Maintainability; Refactoring Customization.

Resumo

Oliveira, Daniel Tenorio Martins de; Garcia, Alessandro Fabricio; Assunção, Wesley Klewerton Guez. **Identificação e análise de modificações relacionadas à refatoração**. Rio de Janeiro, 2024. 111p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A refatoração é uma técnica bem estabelecida na engenharia de software, destinada a melhorar a estrutura do código sem alterar seu comportamento. Cada refatoração consiste em um conjunto de modificações estruturais em um programa. Uma análise minuciosa das modificações de código que compõem uma refatoração é um pré-requisito para colher os benefícios dessa técnica. No entanto, existem pelo menos duas características da refatoração de código em projetos de software da vida real que complicam uma análise detalhada do código. Primeiro, as refatorações são frequentemente aplicadas de maneira *personalizada*, ou seja, os desenvolvedores ajustam manualmente um conjunto pré-definido de modificações de código (associadas a um *tipo de refatoração*) adicionando ou removendo modificações para se adequar a contextos específicos. Em segundo lugar, as refatorações estão frequentemente entrelaçadas com outras tarefas, como adicionar funcionalidades ou corrigir bugs, conhecidas como *refatoração floss*. Como as modificações de refatoração são frequentemente realizadas com outras modificações não relacionadas no mesmo commit, distingui-las é demorado e propenso a erros. Embora pesquisas anteriores tenham examinado essas duas características desafiadoras da refatoração de código, o suporte especializado de ferramentas para sua análise minuciosa ainda é limitado. Esta tese teve como objetivo desenvolver uma abordagem abrangente para auxiliar a análise de refatoração de código na presença dessas duas características desafiadoras. Para isso, primeiro realizamos um estudo com desenvolvedores para entender se eles precisariam de suporte especializado de ferramentas para refatorações personalizadas. Em seguida, desenvolvemos duas ferramentas para ajudar os desenvolvedores a identificar modificações relacionadas à refatoração e distingui-las de outras modificações não relacionadas. A eficácia dessas ferramentas foi avaliada por meio de um estudo com usuários experientes.

Palavras-chave

Refatoração; Manutenibilidade; Customização de refatoração.

Table of contents

List of figures

List of tables

1
Introduction

Refactoring is a well-established software engineering practice aimed at improving the code structure without altering its behavior [1]. The structural improvements achieved with code refactoring are commonly intended to enhance program readability and maintainability [2]. Refactorings are organized in terms of *refactoring types* in existing catalogs (e.g., [1–4]) and tooling support [5,6]. In these catalogs, each refactoring type is defined as a default set of code modifications (or simply, *default modifications*) in the program. An example of refactoring type is Extract Method [2]; its default modifications include moving an existing code segment to a new method and adding a call to the new method in the old source method. The definitions of refactoring types in terms of default modifications provide guidance for developers applying them in their programs.

Despite the known benefits of code refactoring [2, 7, 8], this is a nuanced task that demands advanced code knowledge in order to prevent unintended side effects [9–12]. A key concern in this task is that developers must be able to identify which code modifications are part of the refactoring. In not being able to do so, developers will not capable to understand whether all required refactoring modifications (per type) were appropriately performed. However, in practical settings, this a not trivial task. One of the key reasons is that refactorings are often intertwined with other development tasks, such as adding new features or fixing bugs, which known as *floss refactoring* [13]. Each instance of floss refactoring (performed through a single commit) includes both refactoring and non-refactoring modifications. The latter modifications serve other purposes beyond code structure improvement and, thus, impact system's observable behavior [7, 13, 14]. A typical example of floss refactoring is one in which the developer performs an Extract Method refactoring together with modifications to include a new feature in a program.

Unfortunately, during each instance of floss refactoring, there are often no clear boundaries between refactoring-related and other modifications which can affect the refactoring review [18]. One of the key reasons is that some code modifications in a floss refactoring are simultaneously related to both refactoring and other development tasks in the same commit [13, 15]. For example, a

new extracted method may also be considered to be part of a new feature being added to the program when called by other methods different from the original one. As a consequence of floss refactoring (and other compounding factors), developers need to manually tailor a refactoring by adding or removing code modifications to/from the default ones mentioned in catalogues related to each refactoring type [1, 2]. By tailoring refactorings, developers create customized versions of the refactoring to suit specific contexts, also known as *refactoring customization* [15, 16].

Given the factors above, identifying and classifying different modifications as part of the refactoring can be challenging and requires thorough analysis of the code. Due to this challenge, distinguishing refactoring-related modifications manually is time-consuming and error-prone [15, 17, 18]. Indeed, a previous study [18] indicated that 94% of developers agreed that floss refactoring occurrences slow down the code review process. These developers reported that it is necessary to manually identify which modifications are related to the refactoring in order to properly review them. Therefore, it is crucial to provide automated support to developers not only for applying floss refactoring operations but also for understanding their specific impacts on code review. This would reduce the necessary manual effort allowing developers to focus their review on the crucial parts of the modified code. [17–19]. However, despite the challenges regarding floss refactorings and the need for automated support, developers tend to be hesitant in utilizing existing automated solutions due to their limitations. These limitations may arise from the need to use specific development environments, the lack of flexibility in these tools, or the absence of detailed information about the modifications during the review process [15, 18, 25].

To make matters worse, studies that explore how to automatically distinguish refactoring-related modifications are scarce [15, 20]. Studies in the literature are limited to investigate the frequency and the impact of floss refactoring on overall software quality [7, 8, 17, 21]. Therefore, there remains a significant gap in understanding which code modifications are related or not related to a refactoring instance. Thus, a research question remains: *How can code modifications in a commit be automatically identified and classified as refactoring-related or not?*

Given this challenge, it is crucial a automated tool provide comprehensive support for distinguishing refactoring-related modifications from others. For example, during code review, an automated tool should identify and highlight modifications that are part of the refactoring or not for each refactoring instance. [18]. Also, an automated tool would allow reviewers to understand the

impact of the refactoring modifications, reducing the effort and time required to manually locate them [18]. Finally, when customizing refactorings, developers should be assisted with means to create or remove statements or modifications that affect the code structure, such as creating new method declarations and invocations [15].

1.1 Problem Statement and Related Work Limitations

Although automated tools are needed to support floss refactoring [15, 18, 19, 22], several issues still hinder their effective application and the review of refactoring-related modifications. This leads us to the presentation of the three core research problems that must be tackled to better support floss refactoring practices. These problems are listed below:

1.1.1 Limited Understanding about Refactoring Customization

During floss refactoring, developers tend to customize refactoring instances. A *customized refactoring* includes additional modifications that cohesively contribute to the realization of a refactoring type. It occurs when developers adjust the refactoring instance to fit their specific needs, like adapting to the code or improving design. Frequently, these additional modifications share the same motivation as the refactoring itself. In some cases, they help connect the refactoring to other related modifications within the floss refactoring instance. [15, 16].

In our previous study [15], we investigated and listed frequent refactoring customizations for four different types of refactorings in practice [15, 23]. For this, we proposed an initial approach to identify refactoring-related code modifications during these four refactoring types. However, this study still presents some limitations in its methodological choices. These limitations should be addressed to properly support refactoring customization in practice.

First, the study considered only a limited range of code modifications. Only modifications in certain code locations were considered. For example, during Extract Method refactoring, the code modifications that occurred inside the extracted and original methods were not included in the analysis. This oversimplification may discard important modifications that directly impact the behavior of the refactoring, leading to an incomplete understanding of its effects. Additionally, there was no qualitative evaluation of motivations behind customized refactorings. In other words, we did not investigate why developers customize refactorings. Consequently, the study was limited in

capturing developers' concerns and thoughts regarding the need for refactoring customization support.

> **Problem 1:** The current literature lacks qualitative analysis, leaving the motivations behind developers' refactoring customizations unexplored. This gap hinders understanding and support for their practical needs.

1.1.2 Proper Identification and Classification of Refactoring-related Modifications

As previously discussed, refactorings are often intertwined with other modifications during software evolution, making their differentiation and evaluation complex [13, 18]. The primary issue lies in the difficulty of accurately identifying and classifying refactoring-related modifications, particularly when these modifications are mixed with other non-refactoring changes. In our previous study [15], the proposed approach used to identify code refactoring-related modifications was restrictive. For instance, during the Extract Method refactoring, we only classified modifications that called either the original method or the extracted one as refactoring-related. This restriction potentially overlooked other types of relationships, such as the use of common variables or method signature modifications. Failure to distinguish between refactoring-related and non-refactoring-related modifications can lead to misunderstandings about their purpose and structure. Without accurate distinction, unintended side effects may go unnoticed and unaddressed [15, 18].

Accurate classification is essential to assess the impact and correctness of each modification. Misclassifying modifications can obscure their true intentions, making it difficult to ensure that refactorings are applied effectively. As part of the task of classifying modifications, it is important to identify whether a code modification is *close* to a refactoring instance. The closer a modification is to the refactoring, the more likely it is part of the refactoring process. Modifications close to the refactoring are usually more directly aligned with refactoring purposes, while those farther away are often influenced by other motivations beyond refactoring.

> **Problem 2:** Accurate identification and classification of refactoring-related modifications is challenging when they are intertwined with other non-refactoring modifications.

1.1.3 Providing Tooling Support for Floss Refactoring Review is a Challenging Task

As software development practices evolve, the need for more sophisticated tools to support code review processes becomes increasingly apparent. Current tools provide some support for refactoring, but they are limited to highlighting only the default modifications (see Section 1) associated with refactorings [18, 26]. This limitation results in a significant gap when it comes to identifying and managing all refactoring-related modifications. Given the complexity and potential volume of changes, relying solely on these tools often leads to a labor-intensive and error-prone review process.

To address these challenges, it is essential to develop a tool that offers enhanced visualization of refactorings and related modifications. Without such a tool, reviewers must manually identify which modifications are part of the refactoring, a process that is both tedious and prone to errors [18]. An advanced visualization tool would address these issues by clearly distinguishing between refactorings and related modifications, thereby reducing the cognitive load on reviewers and improving the accuracy and efficiency of their code assessments.

Problem 3: Reviewing floss refactoring commits is demanding and requires significant cognitive effort, especially since current tool support may overlook related modifications.

1.2 Research Contributions

This thesis aimed to expand the understanding of the application and review support for floss refactorings. The expected contributions are listed as follows:

Contribution 1: We conducted a qualitative study into developers' motivations and concerns regarding refactoring customization. This study revealed why developers choose to customize refactorings, what specific challenges they face, and what kind of support they need to improve the process. By addressing these questions, the our research outcomes can lead to the development of more robust tools and methodologies that better support developers in their refactoring efforts.

This contribution sheds light on the underlying factors driving developers to adapt refactorings to their specific needs, which is often overlooked in traditional refactoring studies. By understanding these motivations and chal-

lenges, this study not only identifies gaps in existing tools but also suggests improvements for refactoring practices.

Contribution 2: We developed of a tool to identify and classify refactoring-related modifications by distinguishing them from non-refactoring modifications. This tool provides automated support to identify how close the modifications are to a refactoring instance. By accurately detecting the closeness of modifications, the tool helps developers to understand the relationship between different code modifications. This understanding ensures that the impact of each modification is properly assessed.

Additionally, our proposed code modification detection tool is descriptive and extensible. It categorizes and classify the code modifications, providing an explicit and detailed description of each modification. Its extensible design allow developers to include new refactoring types and new code modification types, making it adaptable to evolving development practices.

Contribution 3: We aimed at improving the practical applicability of tool-based approaches for reviewing floss refactorings. To this end, we developed another tool, called RefViewer, integrated into the code review process. This tool reduces the review effort by automatically identifying refactoring-related modifications. The tool allow reviewers to concentrate on the most critical aspects of the code. Additionally, it provides detailed insights into how each code modification relates to the refactoring, streamlining the review process, reducing effort, and ultimately improving code maintainability. Finally, we conducted a validation study to assess the tool's effectiveness, involving experienced developers, which demonstrated its positive impact on review activity and effort reduction. This contribution offers both a practical solution and empirical evidence supporting its benefits.

1.3 Thesis Propose Outline

The remainder of this thesis, which is a compilation of technical papers (accepted or under submission), is organized as follows.

Chapter 2: Presents the study titled *"The Untold Story of Code Refactoring Customizations in Practice"* published at the International Conference on Software Engineering (ICSE) 2023 [27]. In this study, we investigate: (i) the current knowledge about refactoring customizations and their occurrences in software projects, (ii) the need for adequate support for the application of refactoring customizations, (iii) an assessment of the current limitations of refactoring tools, and (iv) developers' perspectives and expected requirements from a customization support tool. This study represented a first empirical in-

vestigation towards understanding the concerns of the developers with respect to customized refactorings.

Chapter 3: In this chapter, we present the extended version of the study titled *"Digging Deeper: Automated Tooling Support to Identify and Analyze Modifications in Floss Refactoring"*. The paper reporting this study was recently submitted to a major international software engineering conference. In this study, we investigated the relationships between refactoring modifications and other code evolution modifications in the context of floss refactorings. In particular, we investigated how these types of modifications are intertwined and how they are spread in the source code.

Chapter 4: This chapter presents the extended version of the study titled *RefViewer: A tool to identify refactoring-related modifications*. This study introduces a visualization tool designed to help reviewers identify and understand code modifications related to refactoring. The tool highlights refactoring-related modifications and shows their connection to the refactoring, giving reviewers a more complete view. The study evaluates the proposed tool's accuracy in classifying refactoring-related modifications and its ability to reduce the effort involved in reviewing them. It also explores how effectively the tool simplifies code reviews and enhances the overall workflow.

Chapter 5: The last chapter summarizes our current results as well as implications and research publications.

2
The Untold Story of Code Refactoring Customizations in Practice

In this chapter, we present our empirical study published at ICSE'2023 [27], where we delve into several aspects related to refactoring customizations in software. Firstly, we explore the current state of knowledge regarding refactoring customizations as well as the nature of their occurrences in software development. We also identify the pressing need for adequate support in applying these customizations, considering their complexities and varied implementations.

Furthermore, we conduct a comprehensive assessment of the current limitations of available refactoring tools. We highlight areas where improvements are needed to facilitate the adoption and effectiveness of refactoring customizations. Lastly, we investigate developers' perspectives and their expectations from tools specifically designed to support refactoring customizations. We aim at understanding which functionalities and capabilities are most valued in the practical context of software development.

To explore these questions, our study focused on analyzing 13 Java-based open-source projects. We investigated common refactoring types, namely Extract Method, Inline Method, Pull Up Method, and Move Method. Using RefactoringMiner, we identified over 1,162 refactorings comprising more than 100,000 modifications. Insights from this study discuss the way for refining refactoring guidelines for developers. Such insights can also be used as the basis for designing recommender systems, which would assist developers in selecting appropriate refactoring modifications tailored to their coding contexts.

The untold story of code refactoring customizations in practice

Daniel Oliveira*, Wesley K. G. Assunção*§, Alessandro Garcia*,
Ana Carla Bibiano*, Márcio Ribeiro†, Rohit Gheyi‡, Baldoino Fonseca†
*Informatics Department – Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil
§Institute of Software Systems Engineering – Johannes Kepler University (JKU), Linz, Austria
†Computing Institute – Federal University of Alagoas (UFAL), Maceió, Brazil
‡Department of Computing and Systems – Federal University of Campina Grande (UFCG), Campina Grande, Brazil

2.1 Introduction

Code refactoring is a widely used practice to promote program maintainability and other quality attributes [2, 31, 32]. Each code refactoring type is composed of a set of one or more modifications that aim at improving the program structure [2]. To support refactoring, the literature provides a set of standard modifications for each refactoring type [2,33]. Despite the importance of refactorings as a strategy to keep internal software quality, developers remain reluctant on using IDE tools [28–30] to support these refactorings [13,25]. In fact, developers believe these tools have limitations to practical use [19,25]. These factors indicate that existing automated refactoring support may not be sufficient yet.

Previous studies observed that developers usually tailor the set of modifications associated with each refactoring type described in refactoring catalogs [16, 19]. These tailored modifications are named non-standard modifications and are part of refactoring customizations. A *customized refactoring* includes non-standard modifications that cohesively contribute to the realization of a refactoring type. Refactoring customization may be required to satisfy recurring developers' needs such as an adjustment to a local code structure, the removal of a certain poor structure, or even updating client methods [19].

We can observe some attempts to support developers in customizing refactorings. For instance, popular IDEs, such as Eclipse [28], NetBeans [29], and IntelliJ [30], allow developers to customize their refactorings through basic settings. However, previous studies [16,19,25] suggest that these settings are not aligned with the practice. Then, developers are induced to perform refactorings without the use of an IDE [23].

To the best of our knowledge, no study has analyzed in depth the typical customizations of refactoring types across multiple software projects. There are various open questions, including: (i) do developers indeed often customize their refactorings? (ii) what are the most common modifications related to

each customized refactoring? (iii) how to improve IDEs to properly support the application of customized refactorings? The answers to such questions are necessary to guide tool builders in supporting the application of customized refactorings. Also, adequate guidelines and tooling support aligned with the practice may reduce developers' efforts.

Based on these limitations, we conducted a study by mining 13 open-source projects developed in Java. We focused our analysis on four common refactoring types, namely *Extract Method*, *Inline Method*, *Pull Up Method*, and *Move Method* [7, 34]. We identified, by using RefactoringMiner (RMiner) [33], 1,162 refactorings composed of more than 100k modifications. The analysis showed that standard modifications were often accompanied by recurring additional modifications, thereby showing that refactorings are indeed frequently customized by developers. We noticed in commits' comments that their authors mentioned the need for additional modifications to ensure the program's correctness [35, 36].

We found 42 patterns of customized refactoring that occurred in various refactorings of the same type. For instance, various patterns include a similar structure of exception handlers and related method calls, which go against certain IDE tooling mechanics. Developers would not be able to safely reuse these frequent customizations if they are not correctly predefined and supported by the IDE. Even worse, developers would have to: (i) find out by themselves the IDE's transformation is not adequate, (ii) ensure the program's correctness by avoiding unexpected behavior, and (iii) manually apply this non-trivial pattern in their code. Thus, understanding customized patterns is the basis for guiding in-depth investigations of customized refactorings and cataloging the scenarios in which these customized refactorings are applied.

Based on our findings, we evaluated the existing tooling support for applying frequent customized refactoring with widely used IDEs, namely Eclipse, IntelliJ, and NetBeans [23, 37]. We then listed and discussed 12 limitations that hamper the application of found patterns using such IDEs. For example, a key prevailing limitation is the lack of flexibility for developers to choose which method should handle exceptions when performing a *Extract Method*. IDEs make certain rigid choices on the behalf of developers, *e.g.*, inducing an exception handling location, which may lead to bugs in the refactored code. Therefore, our study findings shed light on how to improve refactoring guidelines and tool support. Our findings also provide insights on the design of recommenders for assisting developers in properly selecting code modifications of a refactoring that best match the contextual needs.

Finally, to corroborate the results of our previous analysis, we performed

a survey with 40 developers. This survey was applied to investigate the relevance of the refactoring customization patterns and corresponding tool support. We found that 92.7% of the interviewed developers consider as important the addition of tooling support for customized refactoring in IDEs. Also, the interviewed developers provided additional arguments on the importance of these patterns.

2.2 Background and State of the Art

2.2.1 Refactoring Research and Practice

Code refactoring consists of applying modifications to code structures for enhancing program comprehensibility, maintainability, and other quality attributes [2, 19, 38]. The literature cataloged (*e.g.*, Fowlers' catalog [2]) various refactoring types and their *mechanics*. The mechanics for a refactoring type defines a set of standard code modifications, which guide developers in enhancing their code structure.

For our study, we focused on four popular refactoring types, chosen for two reasons. First, they have different scopes, *i.e.*, they cover both class-level modifications such as *Pull Up Method* and *Move Method*, and method-level ones such as *Extract Method* and *Inline Method*. Second, we focused on frequent, more complex, structural refactorings [38–40]. Simpler refactorings, *e.g.*, renaming, have less room for structural customization. Our selected refactoring types have wide scopes and allow a high number of customizations. Also, these four types of refactorings share structural similarities with other refactoring types, *e.g.*, *Move Method* moves a method from one class to another similarly to *Push Downs* and *Pull ups*.

Table 2.1 describes the refactorings with their corresponding source and target elements. These elements represent the main method modified, *i.e.*, the source, and the method produced after the refactoring, *i.e.*, the target. The standard modification sets are shown in the last column of Table 2.1. These modifications are aligned with Fowler's and Opdyke's refactoring mechanics [1, 2], being the basis for the design of refactoring tools [5, 13, 33, 41–43].

In several contexts, developers may need to customize the standard sets of modifications [16], adding or removing modifications from this standard set to tailor refactorings [16, 38]. These customizations make the application of refactorings more complex [19, 44]. To make it worse, existing refactoring tools (*e.g.*, [6, 33, 45–48]) are mostly focused on only providing support for either the

Table 2.1: Refactoring Details and Standard Refactoring Mechanics

Type	Description	Source	Target	Standard modification set
Extract Method	Create a method based on statements extracted from an existing method	Method where the extraction was performed	Extracted method	• Create the target method with code extracted from the source method • Update variables' references • Add in the source method's body a call to the target method
Inline Method	Incorporate the body of a method into an existing method	Method to be inlined	Method that inlined the source	• Replace each call to the source method with its method body • (Optional) Remove the source's method declaration
Pull Up Method	Move a method from a child class to its parent clas	Method in the subclass	Pulled up method in the superclass	• Create target method in the superclass and copy the source's method body • Remove from all subclasses the source's method declaration • If possible, change source methods calls, with call to the target method
Move Method	Move a method from one class to another class	Method to be moved	Method after being moved	• Create target method with a copy of the source's body method • If removed source's method: replace calls to target method • If did not remove source's method: add target call in source's body

detection or the application of standard mechanics. In this way, there is a lack of tool support for these more complex refactorings, even though the interest of developers has been demonstrated in the literature [19, 25].

Previous studies investigated the motivations behind the refactoring application [49, 50]. Although these studies observed different reasons for performing refactorings, little is known about how refactorings are customized based on developers' needs or motivations. Some studies discuss the concept of *floss refactoring* [17, 38]. *Floss refactorings* are refactorings applied with other development activities, such as feature additions or bug fixes [38, 49]. The set of modifications in a floss refactoring may include some additional and non-standard modifications as part of the refactoring customization. However, these studies do not characterize which of these modifications are related to the refactoring itself. This characterization is necessary to properly support the application of customized refactoring through refactoring catalogs and tools.

A recent study investigated which modifications are combined with *Extract Method* [20]. However, this study focuses on only one refactoring type, besides investigating a limited scope of modifications. Also, the authors use a different AST diff with a higher granularity level. Finally, this study does not investigate the support of these additional refactoring-related modifications on popular IDEs. Another study speculated the need for customized refactoring according to the development context [16]. However, this study did not empirically investigate the occurrences of refactoring customizations in those projects as well as their characteristics and support required.

In summary, the knowledge about customized refactoring is quite limited.

It remains challenging and necessary to investigate: (i) in what ways refactorings are customized in practice, and (ii) whether and how to start improving IDEs refactoring tools to properly support refactoring customizations.

2.2.2 Refactoring Customization

Customized refactoring is a variation of the standard set of modifications defined for a type of refactoring [2]. This variation may occur due to the addition or even the removal of modifications from the standard set. Customization is often needed to tailor the refactoring to a program context. A customized refactoring includes only one or more non-standard modifications that have to be applied together with the standard ones to fully realize a refactoring. In other words, the non-standard modifications of a customized refactoring are also structural modifications required to implement a refactoring type. Non-standard and standard modifications of a customized refactoring interact and cohesively contribute to the realization of a refactoring type.

The conventional definition of refactoring assumes code behavior preservation [2]. This definition is in line with the notion of *root-canal refactoring* [38]. A *root-canal refactoring* occurs when the structural modifications of a refactoring are applied alone in a change and do not interact with co-occurring behavioral changes. However, certain recurring refactoring customizations may also be needed in the context of floss refactorings [38]; that is, the customized refactoring are applied in conjunction with other non-refactoring changes, such as feature addition. The customization may be required due to the interface of the refactored code with the new feature code.

The practical need for frequent floss refactorings does not make it possible for developers to always stick to the behavior-preserving aspect of the conventional definition of refactoring [19,38,51]. Recurring customizations may also exist in floss refactorings, and, as such, developers also need support to perform their frequent non-standard modifications for refactoring types in the context of floss refactoring. Thus, we classify the customized refactorings into two groups: (i) refactoring customizations that do not change the code behavior, *i.e, root-canal customizations*, and (ii) the refactoring customizations that change the code behavior, *i.e, floss customizations*.

Finally, we consider as *customization pattern* the recurring refactorings that satisfy both conditions: (i) they all have at least one structural modification that differs from the standard ones defined for a refactoring type; and (ii) this set of modifications, including the non-standard one(s), consistently occur together in multiple instances of that same refactoring type.

Listing 2.1 presents a refactoring customization, *e.g.*, a root-canal customization, of a *Move Method* that was applied to the Apache Tomcat project [52]. In this case, the developer moved a method called setAllowCasualMP from the Connector class to the StandardContext class. This example has the following modifications: (i) a method was moved from one class to another class, and (ii) a method signature of this method was created on the interface (Context) of the target class. The first modification is part of the standard set of modifications for *Move Method* (see Table 2.1). On the other hand, the second modification is an additional one that customizes the *Move Method*. This additional modification moved the setAllowCasualMP method to the target class and made it an abstract method of the interface implemented by the target class. This additional modification is important to pass the test in the class (TestStandardContext) that calls the setAllowCasualMP method directly from the Context interface. This example is an illustration of a customization pattern of the *Move Method* refactoring, in which the moved methods become part of an inherited interface.

Listing 2.1: Real Example of Customized *Move Method*

```
  public class Connector {
-   public void setAllowCasualMP(){ ...}    ...
  }
  public class StandardContext implements Context {
+   public void setAllowCasualMP(){ ...}    ...
  }
  public interface Context {
+   public void setAllowCasualMP(){ ...}    ...
  }
```

2.3 Study Settings

We investigated how developers apply and customize refactorings on their projects. We derived two research questions (RQs) that guided our study:

RQ_1: *In what ways are refactorings customized by developers?* RQ_1 aims at investigating how refactorings are applied in practice. We observe the most frequent root-canal and floss customizations by analyzing the modifications that compose each commit that includes a refactoring instance. This analysis enables us to identify and understand the most frequent customization patterns. We also discuss divergences between the modifications that compose the customized patterns and the standard mechanics of each refactoring type presented in Table 2.1. As result, we present a catalog of customization patterns

for each refactoring type. These patterns bring insights into how developers apply and customize refactorings.

RQ_2: *How to improve IDEs' automated refactoring tools to properly support customized refactorings?* Automated refactoring tools available in IDEs aim to support standard refactoring mechanics. Thus, they do not properly support customized refactorings both in the context of root-canal and floss refactorings. However, it is important that refactoring tools are in accordance with the practice; otherwise, developers may refuse to use them [19, 25]. In this way, RQ_2 aims at investigating what are the current IDE limitations and how their refactoring features should be improved to properly support the application of customized refactorings. For that, we replicated, using popular IDE refactoring tools, customized patterns from the catalog obtained as a result of the RQ_1. The result of RQ_2 provides a list of identified limitations. This list is the basis to recommend how IDE tools can improve the support for developers to perform customized refactorings.

2.3.1 Study Steps

This section details the steps performed to build the dataset and perform the data analysis in our study. All the dataset-building steps and analyses were conducted by at least two authors and then discussed with other authors. In the presence of conflicting views, further discussion was required to converge. The dataset can be found on our website [53].

Step 1: Project Selection. We selected 13 active Java open-source projects of different sizes and domains. These projects are often used in previous studies of refactoring [31, 39, 40] given their frequency and diversity of refactorings. We took into account the stars count to prioritize popular projects [54]. We focused on open-source projects to facilitate the replication of our study. Finally, choosing Java projects allow us to use RMiner, a refactoring detection tool with high recall and precision, as discussed in Step 2. The selected projects were Elasticsearch-hadoop [55], Hystrix [56], Fresco [57], Achilles [58], Ikasan [59], ExoPlayer [60], Signal-Android [61], Netty [62], MaterialDrawer [63], Derby [64], Tomcat [65], HikariCP [66] and Material-dialogs [67]. The domains are: (i) data search and analysis; (ii) Android systems such as messaging applications and visual design; (iii) application server; and (iv) database construction. These projects have from 1,121 (doanduyhai Achilles) to 17,787 (Tomcat) commits.

Step 2: Refactoring Detection. We used RMiner [33] to detect refactorings performed in the selected projects. We chose RMiner because it

is widely used in the literature [7, 39, 40, 68] and has high recall (87.2%) and precision (98%). With a high recall, the tool captures almost all refactoring instances performed in different contexts for each project. Thus, these instances may represent a variety of modifications used to customize refactorings for these diverse contexts.

We focused on four refactoring types (Table 2.1), which are frequent in multiple projects [7, 39] and are present in popular IDEs. These refactorings constantly occur in a unique commit, affecting the same code fragment, known as composite refactorings [40]. Thus, to avoid modifications of composite refactoring instances to be erroneously considered as part of a unique refactoring type, we selected only commits with one detected refactoring. Lastly, any customization exclusively occurring with particular refactoring compositions (*e.g.*, *Extract Method* with *Move Method*) would be an addition to the customizations already present in our study; in other words, they would complement but not invalidate our results.

Step 3: Modification Detection. We used Eclipse's JDT 3.10 to collect the code modifications [69]. This library parses Java code into an Abstract Syntax Tree (AST). ASTs are widely used in the literature to detect refactorings [33, 70]. The Eclipse JDT is also used by the Gumtree framework [71], a popular framework used in literature to compare ASTs in Java [72, 73]. We used JDT directly because it provides Java language syntax information, allowing us to distinguish the same node type in different contexts. For example, using JDT we could observe whether the SIMPLE_NAME node is associated with a class variable, interface, class name or other Java tokens. These differences are relevant to detecting refactorings customizations and their patterns.

For each refactoring detected in Step 2, we collected the information before (v) and after (v_{+1}) the refactoring occurrence. We collected information related to the classes affected by the refactoring and their clients. We classified a class as affected by a refactoring when the modifications occurred within that class. For instance, an *Extract Method* has at least one affected class. On the other hand, a refactoring of the type *Pull Up Method* or *Move Method* has at least two affected classes, once a method is moved from one class to another one. Finally, we classified as a client of a class or method every other class or method that interacts with the client, *e.g.*, importing it and/or calling a method of the affected class. Once we have two subsequent versions of a class, the AST nodes are defined as $AST_v = \{node_i, node_{i+1}, ..., node_n\}$ where AST_v is the set of nodes belonging to the AST in version v. The set of added nodes to the source code between two subsequent versions is given by the resulting

set of the difference between $AST_{v+1} - AST_v$. Similarly, the set of removed nodes from the source code is given by the difference of $AST_v - AST_{v+1}$.

Listing 2.2: Modifications between Two Subsequent Versions

```
+ public void clear() {
+   if (mAnimatedDrawableCachingBackend ! = null) {
+     mAnimatedDrawableCachingBackend.dropCaches();
+   }
+   ClosebleReference.closeSafely(mLastDrawnFrame);
+   mLastDrawnFrame = null;
+ }
  public void onInactive() {
-   if (mAnimatedDrawableCachingBackend ! = null) {
-     mAnimatedDrawableCachingBackend.dropCaches();
-   }
-   ClosebleReference.closeSafely(mLastDrawnFrame);
-   mLastDrawnFrame = null;
+   clear();
  }
```

Listing 2.2 illustrates the difference between two subsequent versions of a class from the Facebook Fresco project [74]. Table 2.2 presents a partial list of nodes obtained when analyzing the code in Listing 2.2, indicating the node type, scope, and whether the node was added or removed. We grouped the nodes based on semantic similarities of their modifications, creating coarse-grained categories, shown in Table 2.3. For instance, the nodes related to conditional control, such as SWITCH_STATEMENT, CONDITIONAL_EXPRESSION and IF_STATEMENT, were grouped into the group *Conditional*. These categories enabled us to perform analysis and comparison focusing on the semantics of the modifications.

Table 2.2: Nodes Detected in the Subsequent Versions

AST Node	Statement	Element	Status
METHOD_DECLARATION	Animated...Wrapper.clear()	Class	Added
IF_STATEMENT	mAnimated...Backend != null	clear()	Added
IF_STATEMENT	mAnimated...Backend != null	onInactive()	Removed
METHOD_INVOCATION	CloseableR...closeSafely(m...Frame) mAnimated...Backend.dropCaches()	onInactive()	Removed
METHOD_INVOCATION	CloseableR...closeSafely(m...Frame) mAnimated...Backend.dropCaches()	clear()	Added
METHOD_INVOCATION	clear();	onInactive()	Added

Step 4: Dataset Construction. The collected modifications of all refactoring instances might include modifications related to different software engineering activities, *e.g.*, feature addition. Thus, in this step, we focused on filtering out non-refactoring modifications, which are modifications not related to the refactoring activity. Then, in order to facilitate the identification and

Table 2.3: Grouped Modifications

Category	**AST Nodes**
Annotation	ANNOTATION_TYPE_DECLARATION, ANNOTATION_TYPE_MEMBER_DECLARATION, MEMBER_VALUE_PAIR, QUALIFIED_TYPE, NAME_QUALIFIED_TYPE, MARKER_ANNOTATION, NORMAL_ANNOTATION, SINGLE_MEMBER_ANNOTATION
Enum	ENUM_DECLARATION, ENUM_CONSTANT_DECLARATION
Method Declaration	FIELD_DECLARATION, METHOD_DECLARATION, INITIALIZER, LAMBDA_EXPRESSION, MODIFIER
Exception Handler	TRY_STATEMENT, CATCH_CLAUSE, THROW_STATEMENT, UNION_TYPE
Comments	JAVADOC, BLOCK_COMMENT, LINE_COMMENT, METHOD_REF, METHOD_REF_PARAMETER, MEMBER_REF, TAG_ELEMENT, TEXT_ELEMENT
Array Modifier	ARRAY_CREATION, ARRAY_INITIALIZER, ARRAY_ACCESS, ARRAY_TYPE, DIMENSION
Literal Modifier	BOOLEAN_LITERAL, CHARACTER_LITERAL, NULL_LITERAL, NUMBER_LITERAL, STRING_LITERAL, TYPE_LITERAL
Class Creation	CLASS_INSTANCE_CREATION, ANONYMOUS_CLASS_DECLARATION, TYPE_PARAMETER, CREATION_REFERENCE, TYPE_METHOD_REFERENCE
Conditional	CONDITIONAL_EXPRESSION, IF_STATEMENT, SWITCH_CASE, SWITCH_STATEMENT
Method Access	FIELD_ACCESS, METHOD_INVOCATION, SUPER_METHOD_REFERENCE SUPER_FIELD_ACCESS, SUPER_METHOD_INVOCATION, THIS_EXPRESSION, CONSTRUCTOR_INVOCATION, SUPER_CONSTRUCTOR_INVOCATION, EXPRESSION_METHOD_REFERENCE
Operator Expression	INFIX_EXPRESSION, POSTFIX_EXPRESSION, PREFIX_EXPRESSION, ASSIGNMENT
Cast	INSTANCEOF_EXPRESSION, CAST_EXPRESSION, INTERSECTION_TYPE
Variable Declaration	VARIABLE_DECLARATION_EXPRESSION, VARIABLE_DECLARATION_FRAGMENT, VARIABLE_DECLARATION_STATEMENT, SINGLE_VARIABLE_DECLARATION
Class Control	IMPORT_DECLARATION, PACKAGE_DECLARATION
Loop Flow Control	DO_STATEMENT, FOR_STATEMENT, BREAK_STATEMENT, WHILE_STATEMENT CONTINUE_STATEMENT, ENHANCED_FOR_STATEMENT
Type Modifier	SIMPLE_TYPE, TYPE_DECLARATION, TYPE_DECLARATION_STATEMENT, PRIMITIVE_TYPE, PARAMETERIZED_TYPE, WILDCARD_TYPE
Return Modifier	RETURN_STATEMENT

removal of non-refactoring modifications, we split the modifications into two groups based on their code location: (i) the *internal modifications* that occurred within the source and target methods, and (ii) *external modifications* that

occurred somewhere else.

The *internal modification* group includes the modifications within the source/target methods that are identified by the refactoring detection tool. These modifications are cataloged and represented on the RMiner detection rules [33]. For those modifications not detected by RMiner detection rules, we manually observed that different (non-)refactoring modifications depend on particular project aspects, such as design patterns and modularization. We concluded that these situational modifications were not frequent and, thus, did not follow any pattern. Therefore, we decided to discard these modifications from the internal group in our results.

The *external modification* group includes modifications performed externally to the source/target methods and that satisfy one of the following conditions: (i) the modifications are included in the standard mechanics of the analyzed refactoring type, *e.g.*, the creation of the target method during a *Extract Method*; or (ii) the modifications are related to additional software engineering activities, *e.g.*, feature addition, but interact with the source/target methods of the refactoring instance. We consider as interaction with the source/target any invocation of these methods in the source code of the refactoring modifications. For this later case, additional modifications are part of the refactoring activity once they only exist due to the structural change aimed by the refactoring. Those modifications typically determine "the interface" between the refactoring activity and the co-occurring software engineering activities. For instance, existing IDEs support developers in customizing an *Extract Method* refactoring by enabling them to qualify a method as public, protected or private, which is not a standard modification in the *Extract Method* definition, to bind the refactoring modifications with the non-refactoring modifications. This binding is made only due to the refactoring activity (and, therefore, is part of it) as a new method creation is an intrinsic goal of the refactoring. Making the method accessible is a compulsory modification to introduce method calls from client methods that compose the most frequent customizations.

For collecting external modifications, we applied a pattern matching algorithm. This algorithm visits all modifications related to a refactoring instance and collects the Java tokens, *e.g.*, variable and methods names. Then, tokens are filtered out based on whether there is a mention of the source/target method name such as the own method declaration, in the case of external modification (i); or method invocations, in the case of external modification (ii). When there is a mention of the source/target method name, the algorithm counts the number of parameters that were passed in the method invocation. In this way, we avoid misidentifying method invocations on refactoring instances

that have more than one method with the same name. For this to be true, we also needed to remove from our dataset instances that have the source or target methods with the same name and an equal number of parameters.

In summary, we considered as refactoring modifications the modifications that satisfy one of the following conditions: (i) are explicitly listed in refactoring mechanics [1, 2, 33], or (ii) occurred externally to the source/target methods along with other (non-)modifications, but that also interacts with the source/target methods through a method invocation. Altogether, we found 1,162 refactoring instances and more than 100K modifications related to those refactorings. We found the following amount of instances and modifications for each refactoring type: (i) 856 instances and 77,306 modifications related to *Extract Method*, (ii) 174 instances and 14,126 modifications related to *Inline Method*, (iii) 78 instances and 5,856 modifications related to *Move Method*, and (iv) 54 instances and 3,734 modifications related to *Pull Up Method*. Additionally, we collected the commits' comments related to each refactoring instance. In this way, we could observe if developers mentioned any reference to the customizations.

Step 5: Survey with developers. To complement the results of our study, we conducted a survey to evaluate the relevance of the most frequent patterns (RQ_1) and the need for tooling support for such patterns (RQ_2) based on the developers' opinions. We invited participants for the survey using convenience sampling, *i.e.*, developers who are easily accessible [75]. We invited developers using professional and academic mailing lists. The invitees were free to accept or not to participate and we have not provided any reward for participation. The participants answered questions regarding their experience with refactoring applications. In order to level the knowledge of participants, we introduced in detail the definition and mechanics of customized refactorings and described the current refactoring tooling support. For the survey questions, we selected the most frequent patterns that, together, include all modification categories observed in our catalog. Each question included a refactoring instance with the code fragments before and after the application of a selected customization pattern. Then, the developers were asked to analyze those code fragments and indicated whether the presented customization patterns are relevant and whether it would be necessary to have tooling support for their application. We made clear that the developers could ask for clarification during any part of the survey. The survey was composed of true/false and open-ended question types. The first question type allows us to precisely identify the interviewee's final decision regarding the customization support needed. The second question type allows us to understand which factors motivate their

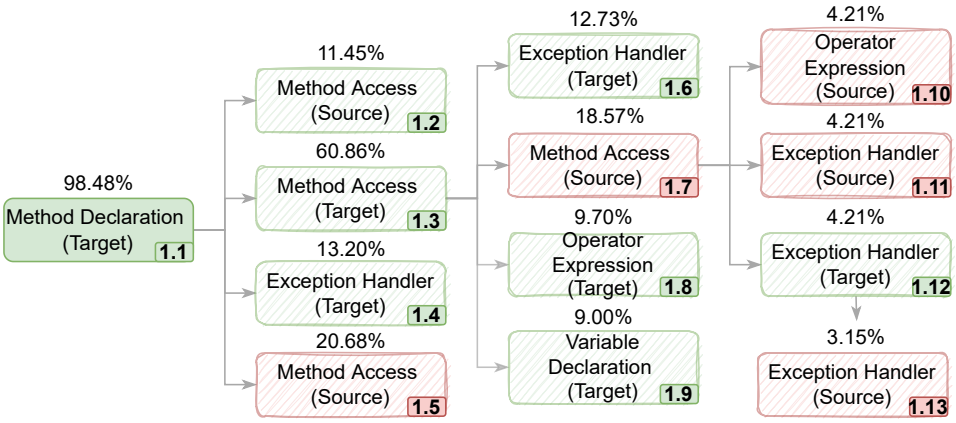

Figure 2.1: Most Common Patterns for *Extract Method*

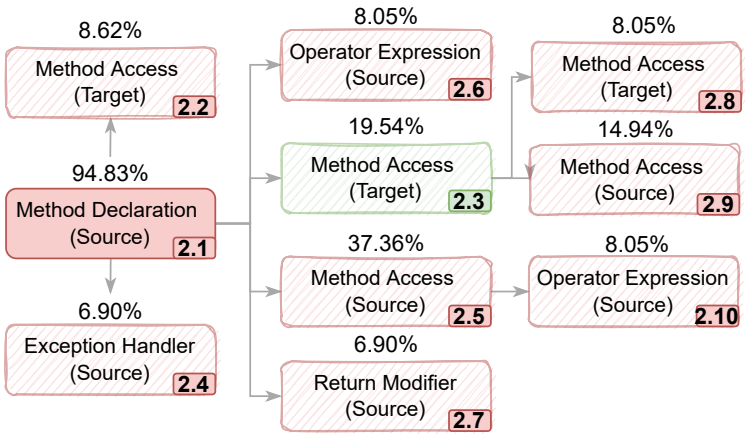

Figure 2.2: Most Common Patterns for *Inline Method*

answer. The complete survey including its questions and answers can be found on our website [53].

2.4 Results and Discussion

The following subsections present RQs' results and analysis.

2.4.1 Refactoring Customization in Practice

Figures 2.1 to 2.4 describe the most frequent customization patterns found for each refactoring type. These patterns are presented as a tree structure. Each node (box) in the figures represents a modification of a specific category performed to apply the customization. The nodes with dark background colors represent default modifications of Table 2.1. The nodes with light background colors represent additional modifications. The green and red colors indicate whether the modification is an addition modification (+) or removal modification (-). The labels source (S) and target (T) indicate

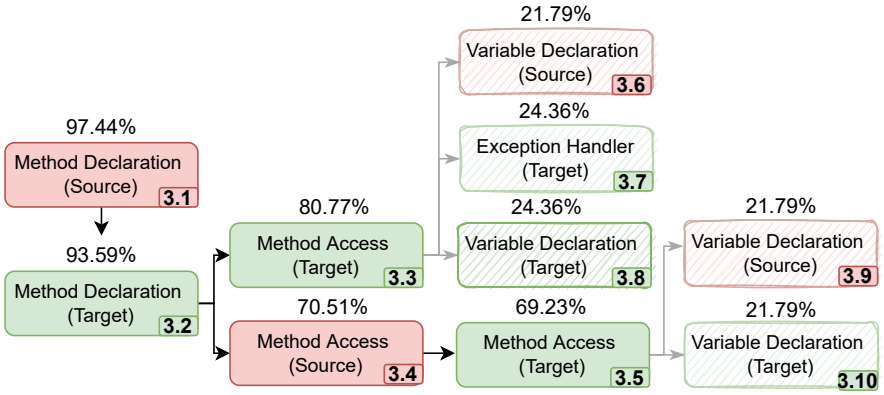

Figure 2.3: Most Common Patterns for *Move Method*

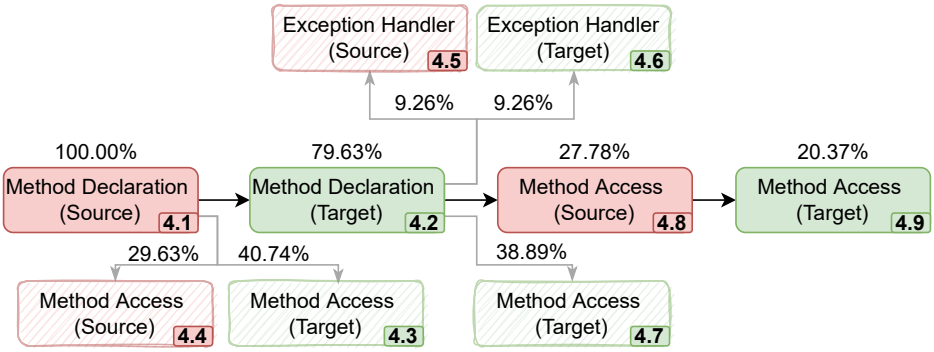

Figure 2.4: Most Common Patterns for *Pull Up Method*

whether the modification interacts with the source or the target method. Each path, starting from the root node, characterizes a pattern of the respective refactoring type. The nodes belonging to the path are the modifications that compose the respective pattern. For example, we have pattern 1.3, for the *Extract Method* type, composed by Method Declaration (target) and Method Access (target). This pattern is expressed in text as {Method Declaration.T+, Method Access.T+}. Finally, we also present the percentage of occurrence of the respective pattern. We consider an occurrence if the pattern is included among all the modifications of a commit.

Developers constantly apply non-standard modifications. We observed that *Pull Up Method* was the only refactoring type in which a modification from the standard set (removal of the source method) occurred in 100% of their instances. On the other hand, modifications in the standard set (the addition of the target method), only occurred in 79.63% of the instances. This means that developers, in their customizations, might even occasionally not perform some standard modifications. Not applying a default modification does not necessarily imply a change in code behavior. In the case of the *Pull Up Method*, the absence of the creation of the target method is due to the existence of a method with the same signature. For *Pull Up Method*, the existent method is found in one parent class of the hierarchy. A similar behavior

is also observed for the method declarations of Pattern 3.2 for *Move Method* and Pattern 1.1 for *Extract Method*. Thus, these patterns are supported in root-canal customization classification.

Patterns that simply added a single additional modification to the standard set occurred in 60.86% of *Extract Method* instances, 37.36% of *Inline Method* instances, 80.77% of *Move Method* instances, and 40.74% of *Pull Up Method* instances. More complex patterns, with at least two additional modifications, are less frequent. However, these patterns still occurred in over 10% of cases for all refactoring types. This is especially true for *Move Method*, which had patterns with five modifications that occurred in 21.79% of the instances. Thus, although the standard modification set is frequent, developers customize this set of modifications to include more possible modifications during the application of each refactoring type.

The most frequent additional modification among refactorings is *Method Access*. This modification indicates the addition or removal of a call to the source or target methods in the client method. The application of this modification unaccompanied by the replacement by the code of the source or target that had the call changed indicates a change in behavior, therefore a floss customization. Other additional modifications such as *Operator Expression* and *Variable Declaration* are related to code readability and thus do not affect the code behavior, being root-canal customizations. Finally, the modifications *Return Modifier* and *Exception Handler* do not exclusively indicate a change in behavior, since they tend to make the code more robust. Patterns with these two latter modifications can be floss or root-canal customizations, depending on the scenario.

Customization pattern modifications are similar for different refactoring types. Figure 2.1 presents the most frequent patterns for *Extract Method*. We observed that the addition of *Method Declaration* of the target method occurred in 98.48% of the *Extract Method* instances. In the remaining patterns, the refactoring mechanic differed from what is considered the default. In these cases, the developers extracted code statements and added them to an existing method. The addition of the extracted code elements to a method containing only the signature would not change the code behavior.

For Pattern 1.3, we observed the occurrence of a *Method Declaration* along with a *Method Access* in 60.86% of the cases. This means that client methods usually add a call to the target method after the extraction. This behavior reinforces the findings that developers extract fragments of code in order to be reused by new clients [7]. Also, for Pattern 1.7, the developers added a *Method Declaration* and *Method Access* to the target as well as removed

Method Access to the source. This pattern suggests a swap between the source and target call. However, only in 11.45% of the instances, the developers switched the call from the source to a call to the target, indicating a possible code change behavior.

Figure 2.2 presents the most frequent patterns for *Inline Method*. We observed that, for 5.17% of the instances, developers preferred to keep the source method when applying *Inline Method* contrasting what is considered the standard. The results also indicate that in 37.36% of the *Inline Method* instances the client methods removed a call to the source method (Pattern 2.5), but only in 14.94% there was also the addition of a call to the target method (Pattern 2.9). Thus, similarly to Pattern 1.7 of *Extract Method*, the client methods that removed the call to the source method and did not replace that call to a call to the target method had their functionality reduced. This reduction in functionality may be related to unexpected code behavior.

Finally, most of the modifications are of the removal type and interact with the source method. This indicates that the clients of the source method needed to be adjusted to remove the interactions that they have with the source method. However, this adjustment is more complex than just removing calls to the source method. We can observe that the client methods also needed to adjust logical expressions (8.05%) and exception handling (6.9%).

Figure 2.3 presents the most frequent patterns for *Move Method*. We observed that most of the patterns tend to add calls to the target method (80.77%, Pattern 3.3) and remove calls to the source method (70.51%, Pattern 3.4). A manual validation indicated that in 57.69% of the instances of *Move Method*, developers added a target method call in client methods that did not call the source method before the refactoring (floss customization). We also noticed that developers performed more complex patterns that include exception handler and variable declaration, both occurred in 24.36% (Patterns 3.7 and 3.8) of the instances. Finally, we noticed that developers were often aware of the need to move the source method in order to improve exception handling. That is, by moving this method, new methods could take advantage of this handling, avoiding unexpected behavior [35, 36].

Figure 2.4 presents the most frequent patterns for *Pull Up Method*. We observed that the removal of the source method together with the addition of the target method occurred in 79.63% of the instances (Pattern 4.2). A manual validation indicated that in the cases without the addition of the target method, the superclass in the hierarchy already had a method with the same signature or an abstraction of it. Based on the commit's messages, developers chose to perform this customization to simplify future implementations and

avoid code duplication [76, 77]. For that, they pulled up only the method's content into a superclass in order to create a standard implementation of this method. That way, each child class that implements this abstraction will no longer be forced to implement this method anymore. That is, this scenario required the customization of the *Pull Up Method* refactoring to fit in this different structure. This scenario is described by the commit's author [77], as follows:

'Move generic code to HttpOrSpdyChooser to simplify implementations. Motivation: HttpOrSpdyChooser can be simplified so the user not need to implement getProtocol(...) method.'

Similar to the other refactoring types, we also observed more complex patterns that also involve recurring exception handling modifications. In those cases, developers were concerned about ensuring the correct flow of the moved functionality, avoiding duplicate executions and unexpected behavior [77, 78]. When moving the handling to the superclass, new implementations of this superclass will have the appropriate standard treatment that already handles possible exceptions, avoiding further problems for users, as mentioned by the commit's author [78].

In general, the standard set of modifications for each refactoring type occurred frequently. However, most of the refactoring instances involved additional modifications, especially method calls for both the target and source methods, and exception handling. These additional modifications turn the refactoring application more complex. The comments of the commits indicated that developers were constantly aware of the need for customization motivated mainly by the addition of new features and the improvement of program correctness, avoiding unexpected behavior in the code. These customizations are recurring and focus on adjusting the refactoring to specific scenarios, *e.g.*, move a method across hierarchies.

RQ_1: In what ways are refactorings customized by developers? Several recurring refactoring customizations are consistently present in multiple projects. The standard refactoring modifications (Table 2.1) are far from being enough to address developers' needs. As such, developers frequently perform additional modifications, as those involving *Method Access* and *Exception Handler*, which extend or remove default refactoring modifications discussed in the literature [2]. Based on that, customized refactorings should be properly documented in order to better assist developers in performing code refactoring.

2.4.2 IDEs' Support for Customized Refactorings

In the previous RQ, we identified the most frequent patterns applied by developers when performing four refactoring types. In this RQ_2, we investigated how to improve the automated refactoring tools provided by the IDEs Eclipse, IntelliJ, and NetBeans to properly support the application of these patterns. We analyzed the source code of the instances of each pattern described in Figures 2.1 to 2.4. We minimally adapted the code to be reproducible in the IDEs' environment. Then, we manually invoked the IDEs' refactoring tools in order to reproduce the refactoring applied by the developer. For each IDE, we: (i) used the same code, (ii) selected the same statements, and (iii) applied the corresponding refactorings. Table 2.4 lists the main limitations (identified from 1 to 12) that hinder the application of custom refactoring patterns when using existing IDEs' refactoring tools. The limitations 1 to 8 occurred in more than one refactoring type.

All IDEs share similar customization impediments. Tables 2.5 to 2.8 present the IDEs support for each pattern and associate them to the limitations shown in Table 2.4. We classified the IDEs' support into three categories: (i) *Full Support*, the refactoring tool is able to reproduce the pattern completely for all reproduced scenarios; (ii) *Partial Support*, the refactoring tool is able to reproduce the pattern completely only if some preconditions are met; and (iii) *No Support*, the refactoring tool is not able to reproduce the complete pattern in any circumstance. Once the IDEs refactoring tools follow the standard modifications, we observed that all IDEs had the same limitations. Thus, we used only one column to indicate the support category for all of them. The last column indicates the limitation *id*.

Table 2.5 presents the limitations for applying *Extract Method*. Except for Method Declaration.T+, all the other patterns have *No Support* or *Partial Support*. Limitation 2 is the most frequent among the *Extract Method* patterns, since most of the patterns include the removal of a *Method Access* in a client method. Limitations 2 to 5 refer to the addition, removal, or swap of methods calls to the source or target method.

Limitation 5 is also related to Pattern 3.3, in which developers add more calls to the target method in Move Methods. We observed this limitation, mainly, when developers apply *Move Methods* to support a feature addition. In the commit FC14CA31CB36 [79] of the Netty project, the developer moved the SAFEEXECUTE method from the SINGLETHREADEVENTEXECUTOR class to the ABSTRACTEVENTEXECUTOR class. The developer also called this moved method in other classes, mainly in classes that were created to support the

Table 2.4: List of the Limitations of IDEs' Refactoring Tools

Id	Limitation
1	Modification only supported if occurred in source/target methods
2	It is not possible to remove source method invocation in client methods
3	It is not possible to remove target method invocation in client methods
4	It is not possible to add source method invocation in client methods
5	It is not possible to add target method invocation in client methods
6	There is no exception support for methods different than source and target ones
7	No exception handler is added if there is an exception error before the refactoring application
8	It is not possible to manage who should handle the exception
9	It is necessary that the extracted code is duplicated and the duplication recognized by the IDE -*Exclusive for Extract Method*
10	It is not possible to remove the modification without replacing it with the inlined method body -*Exclusive for Inline Method*
11	The swap of the call from source to target must occur in the same client -*Exclusive for Pull Up Method and Move Method*
12	It is mandatory to create the moved method, even if there is already a method with the same name in the destination class -*Exclusive for Pull Up Method and Move Method*

Table 2.5: Limitations of *Extract Method* Refactoring Tools

Patterns	IDEs' Support	Limitation Id
(1.1) Method Declaration.T+	Full support	
(1.2) Method Declaration.T+, Method Access.S+	No support	4
(1.3) Method Declaration.T+, Method Access.T+	Partial support	9
(1.4) Method Declaration.T+, Exception Handler.T+	Partial support	6,7,8
(1.5) Method Declaration.T+, Method Access.S-	No support	2
(1.6) Method Declaration.T+, Method Access.T+, Exception Handler.T+	Partial support	6,7,8,9
(1.7) Method Declaration.T+, Method Access.T+, Method Access.S-	No support	2,9
(1.8) Method Declaration.T+, Method Access.T+, Operator expression.T+	Partial support	1 (Operator Exp.),9
(1.9) Method Declaration.T+, Method Access.T+, Variable Declaration.T+	No support	1 (Variable Decla.),9
(1.10) Method Declaration.T+, Method Access.T+, Method Access.S-, Operator Expression.S-	No support	1 (Operator Exp.),2,9

Non Sticky Event Executor Group feature addition, as mentioned in the commit message [79]. A refactoring tool could mitigate this limitation by identifying when a *Move Method* is being applied in the feature addition context. For example, if the developer creates new classes after the *Move Method* application, then the tool can suggest the addition of a call to the previously moved method.

Limitations 6 to 8 affect the modification Exception Handler. For instance, if the selected statements for *Extract Method* throw an exception, the target method will throw this exception, even if the exception thrower is completely extracted. Thus, the IDEs do not allow developers to define which (source/target/client) method must handle that exception. This inflexibility forces all the methods that invoke the target to handle the exception themselves. Due to the lack of automated support, developers may not apply this

exception handling correctly, causing an unintended behavior change.

Tooling support for each refactoring type has particular limitations. IDEs' refactoring tools have the same limitations, as discussed for *Extract Method*, for the remaining refactoring types. However, there are some particularities for each refactoring type. For *Extract Method*, we observed the exclusive Limitation 9. This limitation indicates that it is not possible to manually choose two similar or equal fragments of code to be extracted in a new method. In this way, developers depend on the tool to consider the codes as duplicates, otherwise, developers will need to perform the extraction manually.

For *Inline Method* (Table 2.6), we have the exclusive Limitation 10. In this refactoring, developers can choose to replace the call to the source method with the body of the source method. However, the refactoring tool does not let the developer only remove the call to the source method or replace the call to the source method with a call to the target method, both modifications are often applied. Therefore, developers are forced to: (i) make these not supported modifications manually or (ii) apply the refactoring as suggested by the tool and then remove manually some modifications applied. In both situations, because of the manual step, more effort is needed. This limitation increases the misalignment between refactoring tools and custom refactorings, increasing tool misuse [19, 25].

Limitation 11, exclusive for both *Move Method* (Table 2.7) and *Pull Up Method* (Table 2.8), states that the refactoring tool allows developers to exchange a call to the source method for a call to the target. However, it does not allow only the addition of a call to the target method or only the removal of a call to the source method. For instance, developers may choose to call the target method on methods that did not call the source before refactoring because these methods did not have access to the source method or are in an inappropriate place. Inappropriate places are one of the main reasons why developers apply the *Move method* [7].

Finally, Limitation 12 is also exclusive for *Move Method* and *Pull Up Method*. This limitation indicates that it is not possible to move only the method content to a method with the same signature in the destination class. Thus, developers are forced to: (i) apply these refactoring manually, or (ii) force the method to be moved, leaving the destination class with two methods with the same signature.

We believe that the current tools are helpful for supporting refactoring activities. However, as hypothesized, these tools are not able to properly support the customizations performed by developers due to several limitations. In this way, developers are forced to manually apply the modification set of

Table 2.6: Limitations of *Inline Method* Refactoring Tools

Patterns	IDEs' Support	Limitation Id
(2.1) Method Declaration.S-	Full support	
(2.2) Method Declaration.S-, Method Access.T-	No support	3
(2.3) Method Declaration.S-, Method Access.T+	No support	5
(2.4) Method Declaration.S-, Exception Handler.S-	Partial support	6,7,8
(2.5) Method Declaration.S-, Method Access.S-	Partial support	10 (Method Access)
(2.6) Method Declaration.S-, Operator expression.S-	Partial support	10 (Operator Exp.)
(2.7) Method Declaration.S-, Return modifier.S-	Partial support	10 (Return modifier)
(2.8) Method Declaration.S-, Method Access.T+, Method Access.T-	No support	3,5
(2.9) Method Declaration.S-, Method Access.T+, Method Access.S-	No support	5,10
(2.10) Method Declaration.S-, Method Access.S-, Operator expression.S-	Partial support	10 (Operator Exp.), 10 (Method Access)

Table 2.7: Limitations of *Move Method* Refactoring Tools

Patterns	IDEs' Support	Limitation Id
(3.1) Method Declaration.S-	No support	12
(3.2) Method Declaration.S-, Method Declaration.T+	Full support	
(3.3) Method Declaration.S-, Method Declaration.T+, Method Access.T+	No support	5
(3.4) Method Declaration.S-, Method Declaration.T+, Method Access.S-	No support	2
(3.5) Method Declaration.S-, Method Declaration.T+, Method Access.S-, Method Access.T+	Partial support	11
(3.6) Method Declaration.S-, Method Declaration.T+, Method Access.T+, Variable declaration.S-	No support	1 (Variable Decla.), 5
(3.7) Method Declaration.S-, Method Declaration.T+, Method Access.T+, Exception Handler.T+	No support	5,6,7,8
(3.8) Method Declaration.S-, Method Declaration.T+, Method Access.T+, Variable declaration.T+	No support	1 (Variable Decla.),5
(3.9) Method Declaration.S-, Method Declaration.T+, Method Access.S-, Method Access.T+, Variable declaration.S-	No support	1 (Variable Decla.),11
(3.10) Method Declaration.S-,Method Declaration.T+, Method Access.S-, Method Access.T+, Variable declaration.T+	No support	1 (Variable Decla.),11

Table 2.8: *Limitations of Pull Up Method Refactoring Tools*

Patterns	IDEs' Support	Limitation Id
(4.1) Method Declaration.S-	Full support	
(4.2) Method Declaration.S-, Method Declaration.T+	Full support	
(4.3) Method Declaration.S-, Method Access.T+	No support	5
(4.4) Method Declaration.S-, Method Access.S-	No support	2
(4.5) Method Declaration.S-, Method Declaration.T+, Exception Handler.S-	Partial support	6,7,8
(4.6) Method Declaration.S-, Method Declaration.T+, Exception Handler.T+	Partial support	6,7,8
(4.7) Method Declaration.S-, Method Declaration.T+, Method Access.T+	No support	5
(4.8) Method Declaration.S-, Method Declaration.T+, Method Access.S-	No support	2
(4.9) Method Declaration.S-,Method Declaration.T+, Method Access.S-, Method Access.T+	Partial support	11

a customized refactoring either partially or completely, which is cumbersome and error-prone [19]. In general, the IDEs' refactoring tools present similar behavior. These tools do not allow users to change the modification set of a refactoring; that is, adding modifications besides those predefined by

the IDE for each refactoring type or even removing a predefined one. We agree that IDEs should prioritize supporting code behavior preservation as default. However, even modifications that are not supposed to change code behavior, such as *Variable Declaration* and *Operator Expression*, are not properly supported.

RQ_2: How to improve IDEs' automated refactoring tools to properly support customized refactorings? Refactoring tools should make the configuration of refactoring modifications more flexible, allowing developers to adjust it based on their needs [13]. Existing tools would better adhere to developers' needs if they were designed to (i) support a comprehensive catalog of a mutable set of code modifications; (ii) have a configuration that allows developers to handle the clients that will be affected by the refactoring; (iii) allow developers to choose which element(s) should handle possible exceptions; and (iv) allow developers to choose between creating new methods or using existing ones.

2.4.3 Developers' Opinion About Customization Refactoring

In the last step of our study, we conducted a survey to enrich RQs' results taking into account developers' opinions. All the survey details and results, including those not covered here, are presented on our study website [53]. Altogether the survey was answered by 40 developers. We observed that most of the respondents are familiar with the refactoring application. The majority of respondents (70.8%) declared themselves quite experienced with refactoring, performing refactoring constantly, whereas the remaining indicated applying refactorings periodically. Among the respondents, Eclipse is the most used IDE with 68.2% of them using it, followed by IntelliJ and Netbeans with 46.3% and 14.6%, respectively. Notably, 43% of them also indicated using multiple IDEs.

Respondents agree with the relevance of customization patterns. Survey answers indicate that the majority of the respondents agree with the relevance and support for customization patterns. Their answers were positive for all the types of non-standard refactoring modifications covered in the survey. For instance, the survey revealed 92.7% of agreement concerning both the relevance and the support needed for patterns that include addition and removal of *Method Access*. Interestingly, this modification category is present in the most frequent customization patterns. Also, this category is responsible to allow developers to select which method should access the source and target

methods after the refactoring; this issue is related to the IDE Limitations 1 to 5 (Section IV.B).

With an agreement of 87.8%, the respondents also mentioned the importance of supporting customizations for *Method Declaration*. They agreed that developers should be in charge of deciding whether the method should be entirely (including its declaration) or partially moved. Regarding code exceptions, 75.6% of the respondents agreed that developers should be also given the flexibility of selecting where *Exception Handler* is introduced; this issue is associated with Limitations 6 to 8. Lastly, the respondents also pointed out the importance of tool assistance for refactoring customizations involving *Variable Declaration*, *Return Modifier* and *Operator Expression* with agreement of 70.7%, 65.9%, and 63.4%, respectively.

Customization assistance: spontaneously mentioned positive and negative factors. Finally, we also asked the respondents to openly justify their answers with free text by explaining which factors motivate tooling support for customization patterns. We manually categorized and grouped their answers into positive (*i.e.*, motivating) factors and negative (*i.e.*, demotivating) factors emerging from their answers. These factors are listed in Table 2.9 with the their corresponding percentages of explicit mentions from developers.

We observed that positive factors were much more frequently mentioned than negative ones. Most importantly, the majority of the negative factors have to do with personal preferences or uncertainties of the respondents, including: (i) freely follow their specific programming styles (26%); (ii) preference to apply customizations manually (11%); and (iii) not able to determine if one of the customization patterns (addressed explicitly in the survey) was indeed a refactoring (11%). There were a few cases of respondents that concerning one particular case of customization pattern: (i) was too simple (11%) to be supported by the IDE, or (ii) could not tell whether it was relevant to software maintainability (22%).

Developers argue that explicit customization support would improve code quality and correctness. They reported that IDE assistance for refactoring customization would improve their awareness with respect to bug proneness (41%), guarantees of behavior preservation (19%) and other possible side effects (44%) as well as adherence to good coding practices (30%). Finally, some respondents also found interesting the possibility of they becoming aware of multiple refactoring configuration alternatives (26%).

Table 2.9: Factors motivating refactoring customization support

Positive factors		**Negative factors**	
Awareness of side effects	44%	Own refactoring style	26%
Less error prone	41%	Low relevance	22%
Good coding practice	30%	Simplicity of modification	11%
Awareness of refactoring alternatives	26%	May not be a refactoring	11%
Behavior preservation	19%	Manual preference	11%

2.4.4 Actionable Results

Until now, we discussed the practical occurrence of customized refactoring, the IDE limitations, and the developers' opinions regarding the need for refactoring customization support. Here, we discuss how to incorporate our findings into tooling support. A direct way is by integrating it into a semi-automated strategy of stepwise refactoring [80]. In this strategy, each refactoring modification is a step selected (or approved) by the developer, through which she/he can visualize, understand, and decide about each step. A graphical interface can support these steps by displaying the known alternatives of customizations, *e.g.*, our results shown in Figures 2.1-2.4, which can either be selected or adjusted based on developers' preferences, *e.g.*, following their code styles or team quality standards. Developers can also save their own performed customizations per refactoring type for later reuse.

The stepwise strategy is aligned with developers' expectations observed in our survey (Table 2.9), including (i) their "awareness of refactoring side effects" by tracking each of the refactoring modifications and their code effects; (ii) "reduced error-proneness", allowing the developer to reason about the impact of each modification individually on the behavior of the code; and (iii) "awareness of refactoring alternatives", as the modifications are progressively shown to the developer depending on their previously-selected options, *e.g.*, following a path in the refactoring trees of Figures 2.1 to 2.4. Stepwise customized refactoring favors those developers requiring full control and predictability [25, 81] of customized refactorings ((i) and (ii) above) as they decide on the refactoring application step by step, thereby making them feel more confident using tooling support. This strategy is also aligned with recent and emerging proposals for step-wise refactoring in a range of different contexts, *e.g.*, [16, 80].

2.5 Threats to Validity

We describe here the threats to validity and their mitigation.

Internal and Construct Validity. RMiner [7, 68] may yield false positives and false negatives. It has an effectiveness of 87.2% for recall and 98% for precision [33], which is the best effectiveness among detection tools. To alleviate this threat, we manually inspected some instances of our database. Although we are currently analyzing refactorings detected only by RMiner, it is possible to observe that this tool has detection rules quite flexible, allowing several customizations [33].

RMiner detects 15 types of refactorings in version 1.0 [33], but we analyzed only four types of refactorings. Although these four refactorings may not fully embrace all forms of refactoring customizations, they have been frequently applied [7,34]. Also, these refactorings affect the program structure differently at method-level and class-level. For instance, *Extract Method* is a method-level refactoring, affecting directly one class. Different from *Extract Method*, *Move Methods*, and *Pull Up Methods* affect at least two classes, including changes affecting a class hierarchy. Yet, these refactorings have similarities with other refactoring types, *e.g. Move Method* moves a method from one class to another, similarly to *Push Downs* and *Pull ups*. We chose *Pull Up* to understand this method movement in the context of a class hierarchy. We avoid textual refactorings such as renames. Given their simpler and lexical nature, they have less room for structural customization.

The collected modification types may not consider all possible modification types. We used Eclipse JDT library because this library has a very fine level of granularity. In this way, we could detect a large number of modifications. Besides, this library is commonly used to build automated refactoring tools for Eclipse and RMiner.

Finally, the use of other tools and a larger refactoring interval (considering more than one commit) could present complementary results, such as new customization patterns. However, these supplemental patterns do not invalidate the ones currently reported in our paper nor the limitations of the IDEs.

External Validity. We performed an in-depth analysis of refactoring instances from 13 Java projects. However, our results might not necessarily hold to other projects involving other primary programming languages and/or from domains not covered by our dataset. Moreover, we focused our analysis on open-source projects. The nature of refactoring in closed-source projects is not necessarily the same as refactoring in open-source ones. However, popular

open-source projects have a major concern with software modularity, tending to continuously refactor the source code. We analyzed projects with differing sizes/domains and all key findings were uniform. These projects have an active community, according to Github metrics.

2.6 Conclusion

We presented a study to understand in what ways developers customize refactoring in practice and how to improve refactoring tools to properly support these customizations. We investigated the most frequent customization patterns for four refactorings types in 13 Java projects. The results revealed that developers frequently added new modifications, or remove some, of the standard set for each refactoring type. These changes to the standard set customize the refactorings for the specific developer's scenarios. We then listed the current limitations of popular IDEs that should be improved to provide adequate support for these customizations. We also observed that developers agree with the relevance of customizations and show interest in having tool support for recurring customizations.

Finally, it is important to highlight that the modification sets currently considered standard ones are far from being enough to address practical needs. It is also important to consider the fact that the lack of support for refactoring customization might have serious side effects. Thus, as future work, we plan to design and implement tool support for better assisting developers in performing customized refactorings. We also intend to expand the number of refactoring types and projects being considered in future studies, reducing the threats to external validity.

3
Digging Deeper: Automated Tooling Support to Identify and Analyze Modifications in Floss Refactoring

Although we identified some kind of modifications that occur alongside specific types of refactorings in Chapter 2, the closeness and relationship of these additional modifications to the refactoring itself remain unclear. This understanding is crucial for improving developers' awareness of the potential impact of refactorings, helping to prevent unintended side effects during refactoring application. However, determining whether modifications are related to the refactoring is not straightforward [18]. It requires additional effort from developers and complicates the code review process. As a result, developers have expressed the need for tools that can automatically identify and classify modifications related to refactoring instances [18].

In Chapter 3, we address this gap by investigating floss refactoring and its relationship with code modifications. We developed a tool to detect and categorize modifications related to refactorings, building on the insights from Chapter 2. This tool enhances our understanding of how code modifications relate with refactorings, providing a more comprehensive view of their interactions. This new understanding goes beyond the definition of refactoring customizations, including all modifications related to the refactoring.

In this chapter, we investigate the occurrence of floss refactoring across six different refactoring types. We consider both method-level refactorings (e.g., Extract Method, Inline Method) and class-level refactorings (e.g., Move Method, Inline Method, Push Down Method). We also examined simpler refactorings, such as Rename Method. We propose the following: (i) automated method to support and identify the closeness of modifications to refactoring instances, (ii) a tool for categorizing refactoring-related modifications, and (iii) an analysis of code modifications frequently associated with six popular refactoring types. This chapter presents an extended version of the submitted paper titled *Digging Deeper: Automated Tooling Support to Identify and Analyze Modifications in Floss Refactoring.*

Digging Deeper: Automated Tooling Support to Identify and Analyze Modifications in Floss Refactoring

3.1 Introduction

Refactoring is a software engineering technique that aims to improve the internal structure of software without altering its observable behavior [1,2]. The primary focus of refactoring is to enhance the readability and maintainability of the source code [1, 2]. However, refactorings are often applied together with other development tasks, such as adding new features or fixing bugs [7, 17]. This practice, known as *floss refactoring*, serves purposes beyond the sole act of code improvement, consequently affecting code behavior [14, 17, 82].

Identifying whether a refactoring is "floss" is challenging because the refactoring and additional modifications occur in the same commit, requiring a thorough analysis and understanding of the code. Analyzing the code modifications manually can become a time-consuming, error-prone, and tedious task [17, 27]. Therefore, it is essential to equip developers with sufficient automated support to disentangle the modifications. This automated support should identify additional modifications that interact with the refactoring ones, called hereafter *refactoring-related modifications* [17, 19].

Addressing this gap is highly relevant to software engineering practice. Developers often combine default code modifications associated with refactorings, such as those described in refactoring catalogs [1, 2, 4, 90], with other modifications tied to different purposes [13, 27, 83]. Those code modifications can be so tangled with each other that they can make it harder, or even impossible, to provide proper support for carrying out complex floss refactoring analyses. This entanglement also hinders the application of other activities aimed at analyzing refactorings, such as code review, since the intertwined modifications can overshadow undesired modifications [83].

Studies in the literature have investigated floss refactorings, exploring their occurrence, frequency, and their effect on overall software quality metrics [7,8,17,21]. A few studies delve into identifying the additional modifications that occurred together with the refactoring [16, 20, 27]. Nonetheless, there is a prevailing gap in understanding how refactoring and non-refactoring modifications relate to each other during software maintenance and evolution. Thus, the challenge to fill this gap is: *For a given commit, how to automatically filter*

and classify modifications in refactoring and non-refactoring and identify their relationship?

The current methods for filtering code modifications in the context of refactoring have notable limitations. Refactoring detection tools like RefactoringMiner [33] and RefDiff [6] focus solely on identifying refactorings, without considering other concurrent modifications. Therefore, they do not provide insights into the complexity and scope (*i.e.*, the range of code elements affected by a code modification) of those additional modifications. Conversely, AST differentiation tools such as Gumtree [71] and ChangeDistiller [84], although capable of evaluating detailed modifications, are limited to AST-based actions (*e.g.*, adding, removing, moving, and updating nodes) and do not specifically highlight refactorings, making it difficult to relate additional modifications to refactoring activities. Additionally, recent studies [20, 27] that examined the modifications made during floss refactorings have some limitations. They do not take into account how close the modifications are to the refactoring instance. Modifications close to the refactoring are usually aligned with the refactoring purpose, whereas those farther away are related to other development tasks.

To effectively support the analysis of floss refactoring, it is thus necessary to identify refactorings in commits and untangle refactoring-related code modifications from non-refactoring ones. Therefore, in this work, we define and implement *a new automated method that provides a novel way to identify and analyze fine-grained modifications that are 'part of' or 'related to' floss refactorings*. Our method leverages two state-of-the-art tools, namely RefactoringMiner [33] and Gumtree [71]. However, differently from previous work that utilized those tools, our study aims to understand not only which code modifications occur in instances of floss refactoring, but also the closeness that the additional modifications have to the refactoring instance.

To evaluate our method, we conducted a large-scale study with a myriad of refactoring instances from 213 projects. Altogether, we collected 98,496 instances of six popular refactoring types: Extract Method, Inline Method, Rename Method, Move Method, Pull Up Method, and Push Down Method. Then, for each refactoring instance, we investigated the closeness of each code modification to the refactoring instance. The results of our method were validated by 11 developers, allowing us to measure its accuracy and receive feedback on potential opportunities for improvement.

Our study found several patterns in code modifications during refactoring (e.g., the introduction or removal of method calls, return values, and variables). Interestingly, we observed that refactoring-related modifications can cascade

throughout the codebase, affecting different classes, method calls, access modifiers, and renaming of code elements. This effect is more pronounced in refactorings like Pull Up, Push Down, and Move Method. These refactoring types may require additional attention during code reviews to prevent unintended consequences and ensure that changes do not introduce side effects in other parts of the code. For Extract and Inline Methods, developers usually focus on modifications within the refactored class. We also reveal that refactoring-related modifications tend to increase in complexity as they move away from the original refactoring location.

Our contributions include: (i) an automated method to identify the closeness to modifications pertaining to a refactoring instance; (ii) a code modification detection tool for categorizing refactoring-related modifications; (iii) an investigation of code modifications that frequently occur in conjunction with code refactorings from six popular refactoring types; and (iv) a replication package where all source code and collected data are available [89].

3.2 Motivating Example and Problem Statement

During software development, a developer may refactor code while implementing a new feature or fixing a bug [7, 17]. These modifications can become so entangled with the refactoring modifications that it becomes difficult to track or distinguish the refactoring-related ones.

As a motivating example, Listing 3.1 presents a code diff [85] between two subsequent versions of the class JDBCStore from the Apache Tomcat project [86]. The green background color indicates the coded added, and the red background color the removed code. We can notice that the modifications are mostly concentrated within the method keys(boolean). Among the modifications, we see the application of the Extract Method refactoring, involving the creation of a new private method called String[] keys(boolean expdOnly) from keys() method to encapsulate the logic previously present in the keys() method. Also, it is possible to observe an instance of Inline field refactoring targeting the prepKeysSql field, removing its declaration and adding the field in the method keys(boolean).

In addition to the refactoring modifications, the developer introduced modifications that alter the behavior of the method in conjunction with the Extract Method. These modifications rely on the new parameter expdOnly received in the extracted method to be applied. By passing a positive value to this parameter, the method will present a different behavior from the pre-refactoring version, creating additional filters in the SQL query (lines 468

Listing 3.1: Code modifications from Project Apache Tomcat

```
1   public class JDBCStore extends StoreBase {
    ...
161  protected PreparedStatement prepKeysSql = null;
    ...
456  @Override
457  public String[] keys() throws IOException {
458   <Method extracted: keys(boolean)>
459   keys(false);
460  }
461
462  private String[] keys(boolean expOnly) throws IOException {
463    try {
464     if (prepKeysSql == null) {
465      String keysSql = "<ommited query>";
466        prepKeysSql = _conn.prepareStatement(keysSql);
467      }
468     if (expOnly) {
469        keysSql += "<additional query>";
470     }
471     try (PrepStmt prepKeysSql = _conn.prepStmt(keysSql)) {
472       prepKeysSql.setString(1, getName());
473       if (expOnly) {
474         prepKeysSql.setLong(2, System.currentTimeMillis());
475       }
476       try (ResultSet rst = prepKeysSql.executeQuery()) {
477         // <code>
478       }
479     }
480   }
481   catch (SQLException e) {// <code>}
482   finally {release(_conn);}
483   return keys;
484  }
    ...
923  protected void close() {
924    try {
925      prepKeysSql.close();
926    } catch (Throwable f) {
927      ExceptionUtils.handleThrowable(f);
928    }
929    this.prepKeysSql = null;
930  }
931 }
```

to 470 and 473 to 475). Although these modifications are not part of the refactoring technique, they are directly related to the statements extracted from the KEYS() method, as they share common code elements. In this way, these additional modifications are considered refactoring-related. According to the developer's comment in the commit, the refactoring-related modifications make the method more efficient for specific scenarios and reduce the access control of the JDBCSTORE class.

The developer's comment indicates the intention to alter the code behavior in conjunction with refactoring. However, other changes implicitly impacting the code behavior were not mentioned. Different from the previous version, the new method KEYS(BOOLEAN) always invokes the statement _CONN.PREPSTMT(KEYSSQL) instead of only once. This statement sets a new value to the PREPKEYSSQL variable every time the KEYS(BOOLEAN) is called, impacting the code efficiency in contrast to the first developer's in-

tention. Also, these additional modifications affect functionalities that rely on keeping some state, since PREPKEYSSQL has its value reassigned every time KEYS(BOOLEAN) is called.

In the Inline Field refactoring, the developer moved the variable PREPKEYSSQL from the class level to become a method-level variable within the KEYS(BOOLEAN) method. This change leads to several side effects. For example, it reduces the scope where this variable can be accessed, thus, other methods or classes within this hierarchy will no longer have access to it. As a consequence of this access scope reduction a refactoring-related modification was necessary to remove the use of this variable in the method CLOSE() (lines 923 to 929). This modification in the CLOSE() method alters the exception handling when closing the use of this variable, being the try-with-resources (line 471) the new one responsible for handling the closure of this variable. However, this new closure does not invoke the EXCEPTIONUTILS.HANDLETHROWABLE(...) method (line 927) as used to do before the refactoring affecting the code robustness. The removal of the PREPKEYSSQL declaration as part of the Inline Field refactoring led to this behavioral change in the CLOSE() method.

This scenario highlights the importance of clearly understanding the refactoring-related modifications along such refactorings to ensure that subtle changes do not unintentionally alter the system's behavior without being noticed. In this example, the developer focused the commit comment on the KEYS methods, not mentioning the code behavior change in the method CLOSE(). This oversight occurred because, although the refactoring was included in this commit, the other modifications that constitute the floss refactoring received the developer's primary attention, ultimately obscuring the impact of these code modifications on other parts of the code. This issue is further compounded by the fact that the commits modifications are numerous and spread. For example, the method CLOSE() is located at the end of the JDBCSTORE class, approximately 500 lines away from the KEYS method, which was the main focus of the developer. However, this method is semantically close to the refactoring, since it uses the refactored field PREPKEYSSQL. As a consequence, another commit was made modifying the JDBCSTORE class, with one of its objectives being to adjust the exception handling in the KEYS(BOOLEAN) method [91].

Identifying which code modifications occur in addition to refactoring code and how semantically close they are related helps to understand the impact of code refactoring on large systems [83]. Developers must be able to evaluate this impact based on the number, spread, and complexity of the necessary refactoring-related modifications. This evaluation reduces the introduction of faults [10, 11], unexpected behaviors, and allows the identification of increased

code complexity [87]. Finally, understanding the relation between refactorings and additional modifications provides insight for developing tools [88] and studies to support floss refactoring [27].

3.3 Related Work and Limitations

There are two approaches to detecting code modification in the context of code refactoring. The first type includes the refactoring detection tools, such as RefactoringMiner [33] and RefDiff [6]. These tools focus on identifying the occurrence of refactorings based on default refactoring modifications. As a result, they do not distinguish between detecting pure refactoring and floss refactoring. Consequently, they do not provide information on the relationships among modifications, which hinders the detection of side effects. Listing 3.2 presents an example of the RefactoringMiner output for the Pull Up Method refactoring. We can observe the list of modifications before (leftSideLocations) and after (rightSideLocations) the refactoring. This example indicates that methodA was moved up in the hierarchy, since the method location was moved from Class to its parent, called ClassParent in the example. However, there is no mention of any other modifications that may have occurred during the refactoring process along the default ones.

Listing 3.2: RefactoringMiner Output Example

```
{
    "type": "Pull Up Method",
    "description": "Refactoring summary",
    "leftSideLocations": [{
            "filePath": "Class.java",
            "codeElementType": "METHOD_DECLARATION",
            "description": "original method declaration",
            "codeElement": "private methodA(param ParamType) : void" // lines
    }],
    "rightSideLocations": [{
            "filePath": "ClassParent.java",
            "codeElementType": "METHOD_DECLARATION",
            "description": "pulled up method declaration",
            "codeElement": "private methodA(param ParamType) : void" // lines
    }]
}
```

The second type of code modification detection consists of tools focused on Abstract Syntax Tree (AST) differentiation, such as Gumtree [71] and ChangeDistiller [84]. Although these tools are not exclusively focused on refactoring, they are capable of identifying modifications. Consequently, they are used in refactoring detection studies, being compared to other proposed refactoring detection tools [93]. The Gumtree's output is purely AST-based actions such as adding, removing, moving, and updating nodes to convert from an AST to another AST, as shown in Listing 3.3 for a move operation.

This output is complex, requiring specific knowledge of AST for complete comprehension. Yet, Gumtree does not establish a relationship between the modifications and do not indicate whether refactoring has occurred in the analyzed code.

Listing 3.3: Gumtree Output Example

```
move-tree
ExpressionStatement [260,312]
    MethodInvocation [260,311]
        METHOD_INVOCATION_ARGUMENTS [279,310]
            InfixExpression [279,310]
                SimpleName: soma [306,310]
to Block [356,588] at 0
```

Finally, previous studies [20, 27] have investigated refactorings in conjunction with additional modifications butdid not explore how these modifications relate to each other. Moreira *et al.* [20] focused exclusively on the Extract Method refactoring, while Oliveira *et al.* [27] examined frequent code modifications across four refactoring types. However, Oliveira *et al.*'s study has notable limitations regarding the detected code modifications, since they excluded several modifications from the analysis based on preconditions and location. Additionally, both studies failed to evaluate the closeness between refactoring-related modifications. These limitations result in an incomplete understanding of how refactoring impacts surrounding code, potentially missing significant side effects and overlooking the relationships among modifications. Addressing these gaps can offer a more comprehensive view of how refactoring influences code and improve the detection and management of refactoring-related modifications.

3.4 Refactoring-Related Modification Detection Tool

To address existing limitations, we developed a tool to automatically detect refactoring occurrences and filter modifications based on their relation to the refactoring instance. Our tooling support builds on state-of-the-art tools and adds an extra dimension of abstraction. This additional dimension offers flexibility, allowing developers to choose which modifications to detect and create new strategies for identifying modifications they consider important. Additionally, it is descriptive by providing a detailed explanation for each code modification based on its semantic context.

Unlike a purely AST-based comparison, such as Gumtree [71], our tool evaluates a set of code modifications across several AST nodes. For example, consider the SIMPLE_NAME node, also present in Gumtree's output (Listing 3.3). Analyzing this node alone, we do not have enough

information to conclude what this change represents in the code, beyond the fact that a name changed. Therefore, it is necessary to examine the higher nodes in the tree, starting from the SIMPLE_NAME node. If the nearest higher node is a class declaration, this could indicate a change in the name of a class or even the addition or removal of a hierarchy. On the other hand, if the SIMPLE_NAME node is closer to a variable declaration, we understand that the change refers to that variable.

In the Gumtree output example 3.3, the SIMPLE_NAME is close to a METHOD_INVOCATION node, which means that this name is part of this invocation. Thus, simply knowing which AST node changed is not enough to understand the nature of the code modification or its relation to the refactoring. It is essential to combine this information with the code context, such as the location of the modified node and nearby nodes, to accurately determine what the modification represents to the code. Thus, instead of only using pure AST nodes as modifications, we proposed an initial catalog [89] of common code modifications based on previous studies [27], and AST tools and libraries [69, 71]. We refer to these cataloged modifications as *change models*. Each change model maps one or more AST node actions to a unique, self-explanatory element. Also, every change model is associated with a specific code element and has a logical rule. This rule specifies the set of code modifications that must occur for the change model to be considered present in a commit.

The rules imposed for the change models are not limited to operations in AST nodes (*i.e.*, insertion, deletion, update, or movement) but also include changes that arise from a set of modifications to one or more nodes. For example, within the scope of a single AST node, we can evaluate the characteristics of the code that the node represents. Consider the METHOD_DECLARATION node, which represents a method in Java. Taking two consecutive versions of this node, we can understand if its access modifiers, return type, parameters, or any other internal modifications to the method or its signature were changed. For the case of multiple nodes, consider the CATCH_CLAUSE node. This node represents a catch clause in Java code. By examining an individual catch clause, we can also determine if there was a change in the type of exception it catches. By looking at the set of catch clauses within the same try block, we can understand whether the developer increased or decreased the robustness based on the possible exceptions captured. In case the developer reduced the number of clauses or made the clauses more generic, it could indicate a potential problem with the code's robustness.

Listing 3.4 illustrates the structure of a change model in JSON for-

Listing 3.4: Change Model Example

```
{
 "name": "CHANGED_VARIABLE",
 "value": "{before:int userAge=30;,after:int userAge=45;}",
 "line": "45 - 45",
 "column": "13 - 29",
 "parentElement": "calc.InnerCalculator.wellcome()",
 "element": "calc.InnerCalculator.wellcome().varName",
 "description": "A variable had its values changed",
 "elementType": "VariableDeclaration",
 "variables": "[<variable list>]"
}
```

mat. It has the name of the modified model, followed by its corresponding value. The value is a summary describing the previous and after values for the element. The modified element is denoted by its complete identifier (*e.g.*, "`package.class.method.var`"). The parent element specifies the nearest method or class declaration where the model is contained. Our tool also provides the change models as Java objects, allowing the developer to manipulate or access the element and the AST nodes related to them, or even create a new customized change model using a provided code interface. This approach provides a detailed and structured view of each code modification, related or not, making it easier to detect and understand refactoring-related modifications. The context and description of a modification with the relationship to other elements in the code allow developers to analyze how the code modifications interact with the refactoring process.

Figure 3.1 shows the steps for collecting and categorizing code modifications as refactoring-related. Each refactoring-related modification is represented by a tuple containing three elements: a change model, a relation type that describes how the change model interacts with the refactoring, and a *layer* number. A *layer* indicates the closeness, in terms of significance, of a change model to the refactoring occurrence. The lower the layer number, the closer and more significant the change model is to the refactoring occurrence. Each

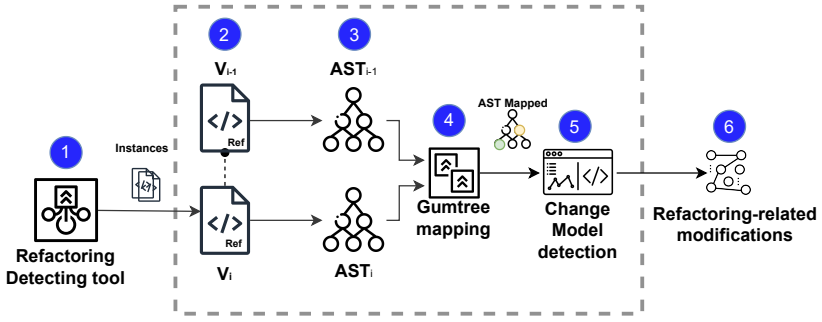

Figure 3.1: Modification Detection Steps

layer interacts exclusively with the previous layers. In this way, the layer one interacts with the layer zero. Similarly, the layer two interacts with the layer one, that by transitivity, interacts with the layer zero.

In Step ❶, to identifies the refactoring-related modifications, our method obtains the refactoring instances from a refactoring detection tool. This list must contain the commit where the refactoring occurred and the respective refactoring. We collect the default code modification used to detect the refactoring instance and the source and target methods for each instance. The source and target methods are the methods before and after the refactoring occurrence. For example, for the Extract Method refactoring, the source method is the other method that had its content extracted, and the target method is the new method containing the extracted code. Table 3.1 presents the complete list of evaluated refactoring, as well as the source and target description.

In Step ❷, for every identified refactoring instance, we obtain the code versions immediately before (v_{i-1}) and after (v_i) the refactoring. This step ensures we have two consecutive versions to identify the modifications that occurred during the refactoring. For this step, we considered the code of all files that had their content changed during the version represented in v_i. We then use both refactoring file versions as input for the next step.

In Step ❸, our method constructed the Abstract Syntax Tree (AST) for the versions v_{i-1} and v_i of the modified files. ASTs are widely used to evaluate changes and are a part of various static analyses [33, 70] in source code. To build the ASTs, we used the Eclipse JDT library (Eclipse's JDT 3.10) [69]. This library is used by other tools and frameworks for static analysis and code change detection [71–73].However, we applied an additional abstraction to the AST structure to keep it closer to the Java language's structure. This additional abstraction allows us to organize and describe non-intuitive or ambiguous nodes, such as the SIMPLE_NAME previously exemplified. As

Table 3.1: Refactoring Details

Type	**Description**	**Source Method**	**Target Method**
Rename Method	Changes a method's name	Method before renaming	Method after renaming
Extract Method	Creates a new method from statements, and to call the new method in the source one	Method that suffered the extraction	Extracted method
Inline Method	Adds a method's body to an already-existing method	Method to be incorporated	Method that incorporated
Move Method	Moves a method from a class to another	Method before being moved	Method after being moved
Pull Up Method	Moves a method up the hierarchy	Method before being moved	Method after being moved
Push Down Method	Moves a method down the hierarchy	Method before being moved	Method after being moved

discussed, this type of node is used in various locations within the AST, with each usage having a different meaning. A SIMPLE_NAME can indicate, but is not limited to, a simple variable name change or the introduction of a hierarchy to a class, depending on where it occurs in the AST.

With ASTs of the two subsequent versions, for Step ❹, our method uses Gumtree (Version 3.0.0) [71] to map the nodes between AST_{i-1} and AST_i. Gumtree identifies which nodes were updated, inserted, removed, or moved between the two ASTs. The AST mappings are input to Step ❺, the change model detection algorithm. This algorithm interprets the mapped AST node pairs and verifies whether the set of code modifications in each pair satisfies the change models rules. If it does, the change model is associated to the closest element associated with the highest modified AST node used to detect the change model.

Finally, in Step ❻, we obtained the list of refactoring-related modifications by assigning each change model its corresponding layer and relation type. Algorithm 1 presents the step-by-step procedure to determine the layer based on four distinct *relation types*, as follows:

Default modification relation: The first relationship is included in the *layer zero*, since the change models on this layer are the most important for the refactoring application. The layer zero exclusively consists of all change models that represent default modifications. These change models are derived from the tool's output when contains modifications used to identify the refactoring. For example, in an Extract Method refactoring, the source and target elements, with the set of lines representing the statements extracted from the source method to the target method, are classified as default modification relation. The change models in this category are part of the algorithm input, which processes the source and target elements along a complete list of change models.

Method signature relation: This relation includes all change models associated to the declarations of the source and target methods that do not have a default modification relation. Since these change models relate with the source and target method directly, *i.e.* change models from layer zero, they are included in the *layer one*. For example, if an Inline Method refactoring results in a private method becoming public after incorporating another method, we consider the change from private to public in the method declaration as a refactoring-related modification of layer one. To obtain the modifications with this type of relation, we first obtained all modifications that do not have a default modification relation (lines 2-3). Then, for each change metric (lines 4-13), we obtained the parent of this change in the AST and verified if the parent is the source or target element. If it is not, we repeat the step until

Algorithm 1 Identifying Related Changes and Layers

```
 1: procedure getRelatedChanges(allChanges, sourceElements, targetElements)
 2:   notRelated ← filter NOT_RELATED changes
 3:   for all change in notRelated do
 4:     parent ← change.closestElement.getParent()
 5:     while parent is not (null or block) do
 6:       for all sourceElement in sourceElements do          ▷ Repeat to targetElement
 7:         if parent equals sourceElement then
 8:           change.setRelation(SIGNATURE, 1)                ▷ Relation type and layer
 9:           continue to next change
10:         end if
11:       end for
12:       parent ← parent.getParent()
13:     end while
14:     for all sourceElement in sourceElements do            ▷ Repeat to targetElement
15:       checkIfCallMainMethod(sourceElement, closestElement, change, CALL_SOURCE)
16:     end for
17:   end for
18:   repeat
19:     hasNewRelated ← false
20:     notRelated ← filter NOT_RELATED changes
21:     newUserVars ← empty List
22:     for all change in notRelated do
23:       usedVars ← getUsedVarsPerChangeType(change)
24:       relatedVar ← null, layer ← MAX_DISTANCE
25:       for all var in usedVars do
26:         if change.usedVariables contains var and usedVars.get(var).layer < layer then
27:           relatedVar ← var
28:           layer ← usedVars.get(var).layer
29:         end if
30:       end for
31:       if relatedVar is not null then
32:         hasNewRelated ← true
33:         change.setRelation(VAR_RELATED, layer + 1)
34:         newUsedVars.putAll(change.usedVariables)
35:       end if
36:     end for
37:     usedVars.putAll(newUserVars)
38:   until hasNewRelated is false
39: end procedure
```

the parent is NULL, or it is a block statement. The NULL value means that the parent reached the root of the file, and the block statement means that the parent is inside a declaration. Thus, it is not part of the declaration signature.

Method call relation: The third relation includes all change models that contain a call to the source and target methods. Different from previous relations, for this relation type, the change model can be included in layer one or above. The layer associated with the change model depends on how far, in terms of AST nodes, the evaluated model is from the call. For instance, if the main element of the change model is a METHOD_CALL, the layer defined will be one (*i.e.*, the minimum possible to this relation). In case the main element is a variable declaration (one node above) used to assign the method call return, the layer defined would be two. This relation is assigned in lines 14-16 in Algorithm 1. For each source and target method, the evaluation checks whether the primary element of the change model calls the target or source method by analyzing the source code, method signature, location, as well as the number and types of parameters.

Common variable relation: Finally, the last relation, checked in lines 18-38 in Algorithm 1, includes change models based on the use of variables in common to previous refactoring-related modifications. First, we obtain the current list of used variables from previous refactoring-related modifications (line 23), then we verify if a not-related modification uses some of these variables (line 26). If so, we classify this previously not related modification as refactoring-related (lines 31-35) and its layer will be one above the layer number that it relates to. The algorithm continuously reapply this loop until no further modifications are included in the set of refactoring-related modifications. For example, in the Extract Method, after the extraction, the source method used to call the extracted one. The return of this call might be assigned to a variable. If a subsequent statement starts using this variable, this modification is considered a refactoring-related one.

3.5 Study Design

This section presents the research questions (RQs) and data collection for our study.

RQ1: What is the performance of the refactoring-related detection tool? Understanding the accuracy of our proposed tool is critical for ensuring its practical utility in real-world scenarios. However, determining a single accuracy value for evaluating the proposed tool is challenging. This challenge comes from the fact that change models represent all code modifications from token level (*e.g.*, variable names, types, and parameter) to entire classes and packages. Thus, in a typical commit, the number of detected change models can reach thousands, making it difficult or infeasible to manually determine whether all relations are perfectly accurate. To address this challenge and answer the RQ1, we implemented a two-step assessment evaluation. We invited experienced developers in refactoring, including practitioners and academics. The developers invited to participate in the experiment were selected based on availability and experience. The developer's expertise in refactoring is essential for providing meaningful feedback and ensuring the reliability of the results. Altogether, 11 developers performed the evaluation. The evaluation details are present in Section 3.5.2. This evaluation provides insights into the tool's performance in accurately identifying refactoring-related modifications, helping us understand how well it performs in various scenarios. Additionally, it may highlight areas where the tool may require further refinement, ensuring that it meets the developer's needs.

RQ2: To what extent are change models related to refactoring instances? This question investigates the most frequent refactoring-related modification and their relationship to refactorings in practice. Specifically, it evaluates which change models are most common and how their frequency and distribution vary with different layer numbers. By answering this research question, we can compile a list of change models modifications that frequently occur, related or not, to refactoring instances. This analysis is essential for reducing unintended side effects during refactoring by helping developers anticipate the impacts of their changes. Understanding which change models commonly accompany refactorings allows developers to identify potential risks and take preemptive measures, making the refactoring process more controlled.

RQ3: How widespread are refactoring-related modifications throughout the code? This question investigates the extent to which refactoring-related modifications affect different parts of the source code. Unlike RQ2, which focuses on the relationship and frequency of change models, RQ3 examines how these modifications are distributed across various source code locations, such as different classes and methods. By answering this question, we aim to gain insights into the complexity of floss refactorings and how modifications in different classes and methods are interconnected. Understanding this distribution not only helps identify which types of refactorings impact a broader range of code elements but also assists developers in being aware of the scope of changes during code reviews. This awareness allows for more thorough and informed review processes, helping identify potential issues that might arise from widespread modifications by focusing on areas prone to be affected by the reviewed refactoring type.

The answers to these research questions can lead to a more systematic and predictable application of floss refactorings, helping developers understand the impact in terms of scope and the necessary efforts before deciding to apply refactoring. The results can also serve as a foundation for developing tools and methodologies to support automated floss refactoring.

3.5.1 Data Collection

The data collection of our study had two steps, as follows:

Project Selection. The first step involved selecting software projects based on criteria commonly found in the literature [7, 39, 40]. We considered projects meeting four criteria: (i) *At least 3500 stars.* This number is much higher than what is typically used in earlier studies [27, 39]. We chose this higher threshold because it is a determining characteristic of a project's

popularity [54]. (ii) *At least 500 forks*. Forks indicate that other developers are interested in the project, whether they are contributing, suggesting changes, or creating variations without affecting the original repository. (iii) *A minimum of 500 commits*. We set this minimum to avoid including projects that are still in their infancy or are just toy programs. (iv) Finally, *Presence of commits in the last 6 months*. This indicates that the projects were active at the time of the data collection. We collected data from 213 projects from different domains, sizes, and characteristics.

Refactoring Detection. After selecting the projects, the next step was identifying the occurrence of code refactorings. For this, we opted to use RefactoringMiner [33] due to its extensive use in the literature [7, 39, 40, 68] and its high recall (87.2%) and precision (98%) values. In our study, we used version 2.4. We considered six types of refactorings, as listed in Table 3.1. We prioritized these six types because most of them are among the ones that occurs with higher frequency in many projects and are widely used in literature [7, 16, 17, 21, 27] and involve different expected code scopes for the default modifications. Among them, Rename Method has the smallest scope, focusing into a single method. Next, the Extract Method and Inline Method, in which default modifications are expected to pertain to a single class. Finally, Move Method is expected to affect two classes, while Pull Up Method and Push Down Method interact with multiple classes in a hierarchy. This set of refactoring types allows us to understand how refactorings with different scopes affect other elements of the code, whether within the same class or external classes.

Altogether, we analyzed 98,496 instances of the six refactoring types (Table 3.1) from a diverse set of 213 Java projects. The number of instances per type is: Push Down: 3,836; Pull Up: 10,556; Inline Method: 4,126; Rename Method: 35,690; Extract Method: 26,281; Move Method: 18,007.

3.5.2 Tool evaluation

For this evaluation, we randomly selected 48 refactoring instances from the previously collected (8 instances per type). In total, 28 different projects were included in this sample. Two different developers evaluated each instance, ensuring thorough review and resulting in a total of 96 evaluations. This approach balanced the need for statistical rigor with practical constraints, such as the availability of the 11 developers and the manageable workload for each developer.

Before the evaluation: All necessary concepts related to the definitions

and relationships between modifications were provided to the developers. They were encouraged to clarify any questions regarding the instances evaluated or any misunderstandings that might arise during the experiment. Once prepared, the developers were presented with the GitHub diff page corresponding to a commit containing one or more refactorings. Refactoring data, including the type and location, were also provided. Finally, the code modifications were highlighted in different colors based on their classification as refactoring-related, default, or non-refactoring modifications.

During the evaluation: In the first step of the evaluation, the developers were asked to provide their perspective on whether the detection tool correctly classified the evaluated instances by indicating if the tool failed to classify any refactoring-related modifications. Then, in the second step, developers evaluated the extent of misclassified tokens by rating their level of agreement with the tool's classification on a scale from 1 (completely disagree) to 5 (totally agree).

3.6 Results and Discussion

3.6.1 Performance Evaluation of the Refactoring-related Detection Tool

Regarding the first question, we observed that in 77 out of 96 instances (80.20%), developers indicated that the detection tool was able to identify as related all the modifications that the developers considered to be related to the evaluated refactoring. Then, when answering the second question, we noticed that in 20 out of 96 instances (33.3%), developers listed at least one code modification they disagreed was related to refactoring.

Based on the developers' responses, we observed two reasons for the extra modifications categorized as refactoring-related. First, some code modifications interact with the refactoring but do not necessarily represent refactored code functionality, such as annotations. Developers consider this set of modifications as false positives. Second, the detection tool provides to the developers refactoring-related for all detected layers, which can result in very high distant code modifications far from the refactorings, either by lines of code or by semantic context of the code. This increases the likelihood of developers indicating some code changes as non-refactoring.

Figure 3.2 presents the results of agreement. We observed that developers exhibit a high level of agreement with the classification. The median agreement is four, suggesting that a substantial portion of developers strongly agree

with the indicated relationships. The interquartile range spans from 3 to 5, indicating that most evaluations were highly favorable. The overall results reflect that the model aligns well with developer expectations, showcasing its effectiveness in identifying and classifying these relationships. For some cases, some evaluations strongly disagree with the classification proposed by the tool. A manual validation revealed that some developers disagreed with the presence of certain refactorings, which meant that no related modifications could exist. In a few instances, the tool identified certain modifications, such as annotations or method comments, as related, though developers did not consider them part of the refactoring. This feedback highlights areas where the tool could be improved to better align with developer expectations.

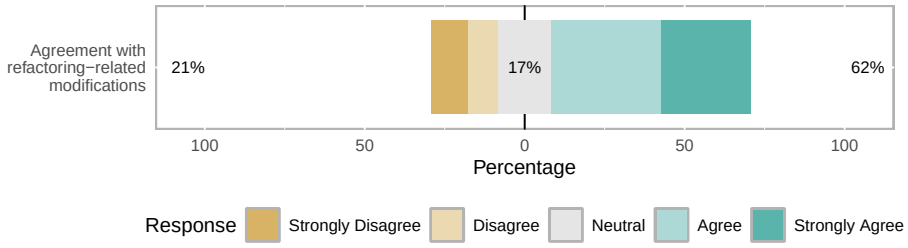

Figure 3.2: Relationship Agreement by Developers

Answering RQ1: The results show that the detection tool is highly effective, with developers agreeing that 80.20% of the refactoring-related modifications were correctly identified. The tool earned a median agreement rating of four out of five, indicating strong alignment with developer expectations. It's important to note that the modifications identified by RefMiner, which are considered default refactoring modifications, are already included in our tool. However, what sets our approach apart is its ability to detect additional modifications beyond these defaults, which other tools like GumTree initially capture but do not filter as related. Developers confirmed that these extra modifications are indeed relevant to refactoring, demonstrating the tool's added value in identifying a broader range of modifications and offering a more comprehensive view of the refactoring process.

3.6.2 Refactoring-related Modifications in Refactoring Instances

Table 3.2 presents a list of modifications for different layers. The first column lists the types of refactorings, and the remaining columns indicate which refactoring-related modifications were found in a specific layer. The

percentage next to each type of modification indicates the frequency with which that modification appeared relative to the total number of refactorings involving that layer. For instance, consider the Rename Method refactoring in the column referring to layer two. In this case, we see the modification *add_method_call* occurred in 56.57% of the Rename Method instances among all instances of this refactoring type that reached layer two.

Layer Zero Analysis. In general, we can observe that, for layer zero, the default modifications consistently appear in almost all refactoring instances, indicating that the results are aligned with the modifications assigned to each refactoring type in literature catalogs [1, 2, 4, 90]. For instance, in the case of Extract Method, modifications such as *add_method_call* and *add_method* occur in over 95% of instances. This aligns with the fundamental purpose of Extract Method, which involves creating a new method and invoking it where the original code block resided. Similarly, Move Method, Pull Up Method, and Push Down Method show high frequencies of *add_method* and *rmv_method* modifications, reflecting their core operations of relocating methods within the class hierarchy. In the case of Inline Method, we observed, in addition to the removal of the method, the removal of method calls, which coincides with the purpose of this type of refactoring.

An interesting observation arises with the Rename Method refactoring, where the *rename_element* modification appears in 84% of instances. The slightly lower frequency compared to other refactoring default modifications suggests complexities in accurately detecting renames, possibly due to variations in method implementations or detection tool limitations. Recognizing these nuances is essential for tool developers to enhance refactoring detection accuracy and for practitioners to be aware of potential inconsistencies during code reviews.

For Extract Method and Inline Method refactorings, we see that the addition and removal of method calls are the most frequent, respectively. However, it is important to note, especially for Inline Method, that the addition and removal of method modifications were not as frequent as method calls. This result indicates that in many instances of these types of refactorings, only the extracted or inlined content was moved between methods. Practically, this means that during Inline Method refactoring, the addition of modifications to variables and control structures are less common, indicating that developers focus on simplifying the code.

Additionally, for frequent modifications, Inline Method tends to counterbalance Extract Method regarding extracted and inlined content. Examples include the addition of infix expressions and return values, which are removed

Table 3.2: More Frequent Modifications per Layer

	Layer 0	Layer 1	Layer 2	Layer 3 and beyond
	rename_element (84.18%)	add_method_call (66.91%)	add_method_call (56.57%)	add_method_call (17.29%)
	add_method_call (39.57%)	rmv_method_call (58.99%)	rmv_method_call (44.64%)	rmv_method_call (12.01%)
	rmv_method_call (38.38%)	rename_element (33.07%)	add_variable (42.63%)	add_variable (11.22%)
	add_return_value (23.60%)	add_method (24.86%)	rmv_variable (35.08%)	add_method (10.48%)
Rename	rmv_return_value (22.86%)	add_return_value (23.48%)	add_infixexpression (27.28%)	add_infixexpression (9.39%)
Method	add_variable (17.60%)	rmv_return_value (20.58%)	add_expressionstatement (27.15%)	add_if_statement (8.74%)
	add_method (17.33%)	add_expressionstatement (20.08%)	add_return_value (25.92%)	rmv_variable (8.51%)
	rmv_variable (17.03%)	rmv_expressionstatement (19.07%)	rmv_infixexpression (21.87%)	add_expressionstatement (7.62%)
	rmv_method (15.35%)	changed_fielddeclaration (18.35%)	add_if_statement (20.64%)	add_return_value (7.40%)
	add_javadoc (11.85%)	rmv_method (17.49%)	rmv_expressionstatement (20.24%)	rmv_infixexpression (6.84%)
	add_method_call (99.79%)	add_modifier (97.24%)	add_method_call (47.10%)	add_method_call (20.80%)
	add_method (95.96%)	add_method_call (72.99%)	add_variable (46.77%)	add_variable (14.02%)
	rmv_method_call (87.90%)	rmv_method_call (49.73%)	rmv_method_call (42.70%)	add_method (13.01%)
	rmv_variable (49.41%)	add_return_value (37.20%)	rmv_variable (32.15%)	add_infixexpression (11.34%)
Extract	add_return_value (48.98%)	rmv_variable (30.25%)	add_if_statement (25.73%)	add_if_statement (10.42%)
Method	add_variable (47.55%)	add_variable (26.13%)	rmv_infixexpression (25.40%)	add_expressionstatement (8.81%)
	add_infixexpression (42.40%)	add_parameterizedtype (24.49%)	add_infixexpression (25.10%)	add_return_value (7.24%)
	rmv_infixexpression (40.06%)	add_if_statement (20.84%)	add_expressionstatement (23.66%)	add_class_instance (6.85%)
	add_if_statement (37.24%)	rmv_if_statement (18.99%)	rmv_expressionstatement (18.93%)	add_classinstcreation (6.85%)
	add_class_instance (36.69%)	add_method (18.61%)	add_return_value (18.33%)	rmv_method_call (6.63%)
	rmv_method_call (99.37%)	rmv_modifier (97.40%)	add_method_call (59.43%)	rmv_method_call (19.62%)
	rmv_method (87.62%)	rmv_method_call (64.32%)	rmv_method_call (47.81%)	rmv_variable (14.18%)
	add_method_call (82.79%)	add_method_call (56.68%)	rmv_variable (47.77%)	add_method_call (13.99%)
	rmv_return_value (62.70%)	add_variable (31.36%)	add_variable (41.21%)	rmv_method (12.32%)
Inline	rmv_infixexpression (44.13%)	rmv_parameterizedtype (24.23%)	add_infixexpression (33.72%)	rmv_infixexpression (12.18%)
Method	rmv_variable (42.49%)	rmv_return_value (24.16%)	add_expressionstatement (30.07%)	add_variable (9.71%)
	rmv_if_statement (38.78%)	add_if_statement (23.41%)	rmv_if_statement (28.37%)	rmv_expressionstatement (9.64%)
	add_variable (37.78%)	add_modifier (23.11%)	rmv_expressionstatement (27.79%)	rmv_if_statement (9.41%)
	rmv_class_instance (34.66%)	rmv_method (23.09%)	rmv_infixexpression (27.19%)	rmv_return_value (8.11%)
	rmv_classinstcreation (34.66%)	rmv_textelement (22.51%)	add_if_statement (26.74%)	add_infixexpression (7.92%)
	add_method (97.92%)	rmv_method_call (64.89%)	add_method_call (63.46%)	add_method (19.26%)
	rmv_method (97.32%)	add_method_call (63.85%)	add_variable (58.77%)	add_method_call (17.63%)
	rmv_modifier (93.90%)	add_fielddeclaration (42.33%)	rmv_method_call (55.11%)	rmv_method_call (17.54%)
	add_modifier (93.59%)	add_class (40.58%)	add_method (50.58%)	rmv_method (15.69%)
Move	add_method_call (79.75%)	add_method (39.39%)	rmv_variable (50.42%)	add_variable (14.24%)
Method	rmv_method_call (79.37%)	rmv_method (37.94%)	add_return_value (42.93%)	rmv_variable (13.92%)
	add_return_value (57.08%)	rmv_fielddeclaration (37.77%)	add_expressionstatement (41.99%)	add_infixexpression (11.69%)
	rmv_return_value (56.70%)	add_expressionstatement (26.93%)	add_infixexpression (39.51%)	rmv_infixexpression (11.59%)
	rmv_variable (45.79%)	rmv_expressionstatement (25.25%)	add_fielddeclaration (38.77%)	add_if_statement (11.31%)
	add_variable (45.60%)	rmv_class (20.75%)	rmv_expressionstatement (36.23%)	add_return_value (10.91%)
	rmv_method (96.87%)	rmv_method_call (60.96%)	add_method_call (66.91%)	add_method (19.86%)
	add_method (93.42%)	add_method_call (60.87%)	add_method (60.18%)	rmv_method_call (17.62%)
	rmv_modifier (92.80%)	add_fielddeclaration (54.54%)	add_return_value (56.34%)	add_method_call (17.58%)
	add_modifier (89.88%)	rmv_fielddeclaration (52.89%)	rmv_method_call (55.68%)	rmv_method (17.11%)
Pull Up	add_method_call (66.45%)	add_class (50.14%)	add_variable (53.71%)	rmv_variable (13.38%)
Method	rmv_method_call (65.20%)	add_method (49.51%)	add_fielddeclaration (51.24%)	add_variable (13.25%)
	add_return_value (54.77%)	rmv_method (46.94%)	add_expressionstatement (51.24%)	rmv_infixexpression (11.80%)
	rmv_return_value (54.38%)	add_expressionstatement (37.89%)	add_infixexpression (48.05%)	add_infixexpression (11.33%)
	rmv_markerannot (51.96%)	rmv_expressionstatement (37.01%)	rmv_variable (44.58%)	add_return_value (11.17%)
	add_markerannot (45.42%)	add_return_value (29.34%)	rmv_expressionstatement (40.69%)	rmv_expressionstatement (11.14%)
	add_method (99.17%)	rmv_method_call (72.33%)	add_method_call (70.52%)	add_method (20.10%)
	add_modifier (93.64%)	add_method_call (69.92%)	rmv_method_call (67.02%)	rmv_method_call (17.90%)
	rmv_modifier (84.80%)	rmv_fielddeclaration (63.50%)	add_variable (60.94%)	add_method_call (17.30%)
	rmv_method_call (68.48%)	add_fielddeclaration (59.54%)	add_method (59.43%)	rmv_variable (14.18%)
Push	add_method_call (68.40%)	add_class (57.19%)	add_return_value (58.95%)	rmv_method (14.11%)
Down	rmv_return_value (62.04%)	add_method (56.62%)	add_expressionstatement (57.25%)	add_variable (13.57%)
Method	add_return_value (62.04%)	rmv_method (56.48%)	rmv_variable (56.37%)	rmv_infixexpression (12.52%)
	rmv_method (61.91%)	rmv_expressionstatement (49.32%)	add_infixexpression (53.50%)	add_infixexpression (11.80%)
	add_markerannot (51.38%)	add_expressionstatement (44.77%)	add_fielddeclaration (52.36%)	rmv_expressionstatement (11.75%)
	rmv_variable (31.00%)	rmv_return_value (42.16%)	rmv_expressionstatement (52.27%)	add_expressionstatement (11.72%)

during Inline Method and added during Extract Method. Recognizing these patterns allows developers to adjust their refactoring practices to address specific modification patterns. For instance, knowing that method calls are the most frequently modified element in Extract Method refactorings can help developers anticipate the scope of modifications and manage them more effectively. By understanding that Inline Method frequently results in the removal of infix expressions and return values, developers can proactively review and adjust these elements to avoid missing critical functionality. Finally, the addition and removal of variables are common for both refactorings due to significant changes in method content and structure. Developers must monitor these modifications to maintain consistency and prevent side effects.

For refactorings involving method movement, such as Move Method, Pull Up Method, and Push Down Method, we observed a high incidence of method additions and removals, as well as modifications to method modifiers, such as public, final, and static. Specifically, for Push Down, the frequency of method removal is notable at 61.91%, indicating that developers often retain the method signature in the superclass and perform push-down operations to override this signature. This is supported by the frequency of marker annotation modifications, representing the addition of @override annotations on methods whose hierarchy has been altered. In practice, understanding these patterns helps in anticipating and managing changes to method signatures and annotations, allowing developers to reapply common and effective Push Down strategies.

Layer One Analysis. Layer one contains refactoring-related modifications that are indirectly triggered by the initial refactoring, such as adjustments to dependent code elements and resolution of resulting inconsistencies. For the Rename refactoring, we have the addition and removal of method calls. This highlights the cascading effect a simple rename can have across the code, especially in large or tightly coupled systems. Additionally, the existence of *rename_element* in 33.07% of instances suggests that renaming a method often leads to renaming related entities for consistency, such as variables, methods, and classes. This result indicates that even method-level refactorings, such as Rename Method, can affect class-level structures.

In the context of Extract Method and Inline Method, the frequent changes to modifiers and method calls. Unlike layer zero, these calls are not necessarily within the main method itself (*i.e.,* the extracted or inlined method) but rather in other methods (client methods) that interact with the main method. Thus, this result indicates adjustments to access levels and invocation patterns to maintain functionality and encapsulation after refactoring.

Notably, the imbalance between additions and removals of method calls in Extract Method (a 23.26% difference) raises concerns about potential side effects, as the introduction of new functionalities or reuse of extracted methods in additional contexts could inadvertently alter the program's behavior. This highlights the importance of careful review to ensure that such modifications do not introduce unintended consequences.

For method-moving refactorings (*i.e.*, Move Method, Pull Up Method, Push Down Method), similar to Extract and Inline Methods, the additions and removals of method calls are among the most frequent changes. However, the frequency of the opposite call modifications differs from them. For these method-moving refactorings, the addition and removal of calls occur at similar frequencies, in the worst case being only 2.41% for Push Down Method. Regarding other frequent modifications, we observe that the three refactorings share exactly the same modification types. The only exception is Move Method, which includes the class removal as refactoring-related. Our hypothesis is that the class became unnecessary once the methods contained within it were moved. Finally, consistent modifications to field declarations and class structures reflect the need to adapt surrounding code to the method's new location. This adaptation includes updating or relocating related fields and ensuring that class hierarchies correctly represent the new design intentions.

Higher Layers (Two and Beyond) Analysis. Layers two and beyond represent more complex and less direct modifications resulting from or related to the initial refactoring. These layers often involve iterative adjustments and refinements as developers integrate refactored code into the broader system context. The persistent prominence of *add_method_call* across all refactorings in layer two suggests ongoing integration efforts, where refactored methods are increasingly utilized or their usage patterns evolve. For refactorings like Inline Method and Pull Up Method, where these refactoring types intent to simplify or consolidate code, the unexpected frequency of added method calls may indicate scenarios where new abstractions or functionalities emerge, necessitating additional method invocations. Moreover, the diverse range of modifications in higher layers, even occurring less frequently, underscores the varied and context-specific nature of how code evolves after refactoring. These refactoring-related modifications often involve adding or removing complex structures like infix expressions, if statements, and entire classes, which may reflect ongoing design improvements and feature expansions.

3.6.3 Distribution of Refactoring-related Modifications

In the discussion above, we presented the frequency in which different refactoring-related modifications occur per layer for each type of refactoring. Table 3.3 shows an average of how modifications are distributed between related and not related modifications across commits. We observe that not related modifications are much greater in terms of quantity, reaching 92.97% of code changes containing the Rename Method refactoring type. The refactoring with more average refactoring-related modification is Inline Method, where 16.72% are related modifications.

Table 3.3: Distribution (non-)Related Modifications

	Move Method	Extract Method	Rename Method	Inline Method	Pull Up Method	Push Down Method
Non-Related	86.44%	85.6%	92.97%	83.28%	86.18%	84.05%
Related	13.56%	14.40%	7.03%	16.72%	13.82%	15.95%

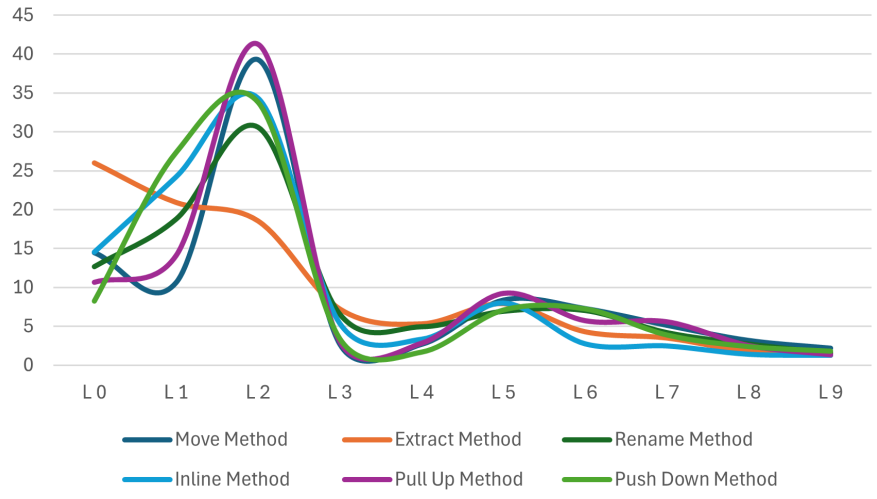

Figure 3.3: Modification Distribution Throughout the Layers

Regarding the concentration of refactoring-related modifications throughout the layers, we present Figure 3.3. The x-axis represents the layer and the y-axis represents the percentage of the total of modifications for the respective layer. In this figure, we can observe that, except for Extract Method, refactoring-related modifications are concentrated in layers one and, especially, layer two. This means that additional modifications interacting with the resulting code from the refactoring are more prevalent. As for Extract Method, it exhibits a different distribution from the others, its highest concentration is in layer zero. This result suggests that commits involving

Extract Method tend to focus on the refactoring itself. Additionally, all types of refactorings also have a higher concentration in layer five or six, following a pattern of decay for higher layers.

Answering RQ2: The analysis reveals that refactoring often triggers various modifications across different code layers, highlighting the interconnected nature of code modifications. Most of the refactoring-related modifications are concentrated in the first two layers. For example, the frequent addition of method calls during Extract Method refactoring, showing a 23.26% imbalance, underscores the importance of anticipating the scope and impact of these modifications for more effective planning. Patterns such as modifications of complex structures in higher layers, even in method-level refactorings, can guide the development of smarter, context-aware refactoring tools, helping manage intricate code modifications.

3.6.4 Spreading of Refactoring-related Modifications throughout the Source Code

We also analyzed the location of modifications related to refactorings in the source code to understand their impact in terms of scope. For each refactoring type, Table 3.4 presents three columns indicating different source code locations. The column "Same Method" indicates the percentage of modifications that occurred in the source or target method in the respective layer of the first column. Similarly, the column "Same Class" indicates modifications in the class where the source and target methods reside. The column "Outside" indicates if the modifications in classes other than those where the source and target methods are located.

In general, the distribution of modifications related to refactoring reveals distinct patterns based on the type of refactoring being applied. First, we observed that for all refactoring types, modifications tend to spread and affect a wider range of elements as the layer number increases. Second, refactorings with a similar focus, such as Extract and Inline Methods, which are concerned with adding or removing methods, tend to exhibit similar distribution patterns for related modifications. Similarly, refactorings involving method movement, such as Move Method, Pull Up Method, and Push Down Method, show comparable distribution behaviors. Finally, Rename Method also displays similarities with method-movement refactorings, displaying a similar modification distribution.

Table 3.4: Refactoring-related modifications Distribution in the Source Code

Layers	**Extract Method**			**Inline Method**			**Rename Method**		
	Same Method	Same Class	Outside	Same Method	Same Class	Outside	Same Method	Same Class	Outside
L 0	95.46%	4.53%	0.01%	92.58%	7.41%	0.01%	85.83%	14.16%	0.01%
L 1	42.47%	53.74%	3.79%	26.36%	71.05%	2.60%	0.30%	84.65%	15.05%
L 2	18.63%	73.25%	8.12%	7.58%	86.97%	5.45%	0.05%	86.76%	13.19%
L 3	15.71%	69.91%	14.38%	12.57%	67.26%	20.18%	0.08%	59.90%	40.02%
L 4	10.90%	62.91%	26.19%	10.00%	58.87%	31.13%	0.06%	43.42%	56.53%
L 5	5.68%	62.43%	31.89%	6.56%	60.31%	33.13%	0.03%	44.27%	55.70%
L 6	5.90%	62.82%	31.27%	5.53%	56.17%	38.31%	0.02%	32.91%	67.07%
L 7	5.52%	56.29%	38.19%	2.40%	54.89%	42.71%	0.02%	36.88%	63.10%
L 8	7.81%	62.50%	29.69%	5.68%	49.14%	45.19%	0.03%	30.65%	69.32%
L 9	8.28%	57.20%	34.52%	4.19%	50.83%	44.98%	0.01%	32.61%	67.38%
Not Related	1.54%	14.84%	83.62%	0.95%	19.06%	79.99%	0.00%	12.24%	87.76%
	Move Method			**Pull Up Method**			**Push Down Method**		
Layers	Same Method	Same Class	Outside	Same Method	Same Class	Outside	Same Method	Same Class	Outside
L 0	94.45%	5.55%	0.00%	92.61%	7.39%	0.00%	93.59%	6.41%	0.00%
L 1	0.12%	91.39%	8.49%	0.19%	93.12%	6.70%	0.35%	96.78%	2.87%
L 2	0.01%	90.59%	9.40%	0.00%	93.81%	6.19%	0.00%	92.20%	7.80%
L 3	0.02%	66.81%	33.17%	0.00%	60.49%	39.51%	0.00%	68.27%	31.73%
L 4	0.09%	63.28%	36.63%	0.00%	47.73%	52.27%	0.00%	55.36%	44.64%
L 5	0.00%	67.18%	32.81%	0.00%	63.37%	36.63%	0.00%	67.78%	32.22%
L 6	0.01%	67.16%	32.83%	0.00%	54.52%	45.48%	0.00%	63.98%	36.02%
L 7	0.00%	65.02%	34.97%	0.00%	60.29%	39.71%	0.00%	58.63%	41.37%
L 8	0.00%	70.55%	29.45%	0.00%	49.69%	50.30%	0.00%	67.24%	32.75%
L 9	0.00%	64.71%	35.29%	0.00%	45.85%	54.14%	0.00%	63.86%	36.14%
Not Related	0.00%	22.88%	77.12%	0.00%	26.83%	73.17%	0.00%	20.99%	79.01%

For Extract and Inline Methods, the modifications are predominantly concentrated within the source and target methods in layer zero, where more than 90% of modifications are observed. This high concentration indicates that these refactorings predominantly affect the methods directly involved. However, there are noticeable modifications in the same class, with occurrences of 4.53% and 7.41% for Extract and Inline Methods, respectively. This pattern arises due to the frequent association of method addition and removal with class-level changes, as observed in Table 3.2. Even in higher layers, for these two refactoring types, modifications within the source and target methods persist, with 8.28% for Extract Method and 4.19% for Inline Method in layer nine. Additionally, we can observe a considerable increase from layer three onward in occurrences of modifications in the outside column. The occurrence of modifications on outside classes remains above 30% of in higher layers, reaching 44.98% in layer nine for Inline Method.

The occurrence of refactoring-related modifications on outside classes from layer three onward is more intense in method-movement refactorings

and Rename Method. Furthermore, for these refactorings, the concentration of modifications in the source and target methods is solely in layer zero, with values nearly zero from layer one onward. Lastly, we observe that layers one and two show the highest incidence of modifications in the same class as the source and target methods.

These results suggest that, for each type of refactoring, developers should adopt a targeted approach to ensure comprehensive refactoring application and review. For Extract and Inline Methods, developers should focus primarily on modifications within the source and target methods. They should also pay particular attention to refactoring-related modifications in the same class, as these refactorings often affect class-level elements. For method-movement refactorings and Rename Method, developers should focus on class-level modifications. These refactorings can introduce modifications across multiple classes. This broad impact may lead to unintended side effects elsewhere in the code, requiring careful review.

Answering RQ3: Refactoring-related modifications are primarily concentrated in the source and target methods at layer zero across all refactoring types. Extract Method and Inline Method show a continued presence of modifications in these methods even in higher layers. In contrast, modifications in outside classes become more prevalent from layer three onward, indicating that even simpler refactorings, such as Rename Method, can lead to dispersed changes in the codebase. This suggests that developers should be vigilant during code reviews, particularly for refactorings with broader impacts.

3.7 Threats to Validity

Internal and Construct Validity. Our method depends on two key detection tools, namely GumTree [71] and RefactoringMiner [93]. While these tools are reliable (achieving an 87.2% recall and 98% precision in the case of RefactoringMiner), false positives pose challenges for both validation and the correct identification of refactoring-related modifications. To mitigate these issues, we manually inspected instances to ensure accuracy. In constructing the tool, we established four distinct types of relationships between modifications and refactorings. However, we acknowledge that more complex relationships may exist that are not yet addressed by our approach.

External Validity. Our study evaluated the tool exclusively on Java projects, which may limit its applicability to other programming languages.

Nevertheless, we ensured a broad and diverse sample, analyzing 213 commits across projects from various domains and sizes. While this large-scale evaluation strengthens the generalizability of our findings within the Java ecosystem, we acknowledge that results might vary if applied to projects using other programming languages or in different development environments.

3.8 Conclusion and Implications

We explored the extent of refactoring-related modifications in floss refactoring occurrences. Altogether, we analyzed 98,496 instances of six popular refactoring types across 213 projects. First, we proposed a tool for detecting and categorizing refactoring-related modifications. Using this tool, we cataloged frequent modifications that occur alongside refactorings and developed an approach to determine their closeness to refactoring instances.

Our results reveal distinct patterns in code modifications during refactoring, such as the introduction or removal of method calls, return values, and variables. We found that refactoring-related modifications often cascade through the codebase, particularly in refactorings like Pull Up, Push Down, and Move Method, which may require closer scrutiny to prevent unintended side effects. While most refactoring-related modifications are concentrated in lower layers, especially within the source and target methods, we also observed frequent modifications at higher layers, such as class field declarations, even in method-level refactorings like Extract and Inline Methods.

Our findings emphasize the importance of distinguishing between refactoring-related and unrelated modifications, which is significant for researchers and practitioners. For practitioners, we contribute to the understanding of how widespread refactoring-related modifications are and the complexity of refactoring. For researchers, our study offers tools for examining refactoring-related modifications in floss commits. Additionally, the catalog of frequent modifications and their closeness to refactoring instances can help create tools that better support developers in recognizing and managing the full scope of modifications during the refactoring process.

Data Availability

All source code, collected data, instances, and project list are available online [89].

4
RefViewer: Visualizating Refactoring-related Modifications

In Chapter 3, we examined the challenges developers face when distinguishing between refactoring-related modifications and other code modifications within the same commit. In order to address these challenges, we proposed a tool to automate the detection of these refactoring-related modifications. However, while the proposed tool offers significant advantages, its practical integration and utilization during the code review process remain a concern. Studies have highlighted that floss refactoring can considerably slow down code reviews, as developers must manually identify which modifications pertain to refactoring. This manual process not only increases effort but also heightens the risk of errors, ultimately affecting productivity [18, 19]. Chapter 3 focused on the tool's detection capabilities but does not address how such tools can be effectively applied in real-world code reviews. Therefore, exploring the integration of automated tools into code review workflows is crucial for simplifying the review process, reducing efforts, and avoiding overlooked modifications.

To address this issue, Chapter 4 builds on the insights from previous chapters by proposing a practical solution: an automated visualization tool integrated into the code review process. This tool identifies and highlights refactoring-related modifications, offering a streamlined approach for developers to distinguish between refactoring-related modifications during the review process. By providing clear visual indicators, the tool aims to reduce the effort involved in reviewing floss refactoring instances, minimize the risk of overlooking important modifications, and ultimately enhance developer productivity.

In this chapter, we present an extended version of the paper entitled *RefViewer: Visualizing Refactoring-Related Modifications*. Here, we introduce RefViewer, our proposed GitHub diff tool that automatically highlights refactorings and their related modifications. Building upon the tool proposed in Chapter 3, RefViewer provides a comprehensive view of the closeness and relationships between the code modifications. We assess the tool's effectiveness in reducing code review effort during floss refactoring and improving developers' understanding of refactoring impacts. The findings underscore RefViewer as a valuable addition to the development workflow, complementing

the automated detection methods introduced in Chapter 3. Finally, we provide a more detailed discussion of the validation process for the refactoring-related detector, as introduced in the previous chapter.

RefViewer: A tool to identify refactoring-related modifications

4.1 Introduction

Refactoring is a common practice in software engineering aimed at improving the internal structure of code without altering its external behavior [1, 2]. The goal is to enhance code readability and maintainability, facilitating future modifications [2]. Various types of refactorings are cataloged in the literature [1, 2, 4], with each type associated with a set of refactoring-related modifications considered *default* for its respective refactoring type. For example, the default modifications for the Extract Method refactoring involve moving a code segment into a new method and replacing the original segment with a call to that new method [2].

Despite its benefits, the practical application of refactoring is fraught with challenges [10–12]. A significant issue is that refactoring is often performed concurrently with other software development tasks, such as feature additions or bug fixes. This is a practice referred to as "floss refactoring" [14, 17]. In a floss refactoring, developers intertwine both refactoring and non-refactoring modifications in a single commit, with the latter aimed at altering the system's observable behavior [17]. Therefore, along with code reviews, developers need to discern which modifications pertain to the refactoring to ensure that all required refactoring-related modifications are adequately executed [18].

Distinguishing and classifying different modifications within each instance of floss refactoring is complex and demands meticulous code analysis [18, 27]. Manually identifying refactoring-related modifications is labor-intensive and prone to errors [18]. Studies have shown that floss refactoring can significantly slow down the code review process, as developers must manually determine which modifications are related to refactoring for proper review, leading to increased effort and potential mistakes [18, 19].

Despite these challenges, existing studies have focused only on identifying the refactorings themselves, overlooking the intertwined nature of floss refactoring [18, 26]. As a result, developers are only shown the default modifications with no support or visual identification to other possible refactoring-related modifications. For instance, a recent study [27] found that developers often add or remove method calls or adjust exception handling as a direct result of performing a refactoring. These refactoring-related modifications, closely

tied to the refactoring process itself, lack proper tool support and can lead to unobservable side effects if not carefully reviewed [27].

Given these challenges, automated support is essential to assist developers on understanding the specific impacts of refactoring on the code along code reviews. To address this need, we propose RefViewer, a tool designed to integrate into the code review process. The tool automatically identifies not only refactoring-related modification considered as default but also related modifications that occur as a result of the refactoring. By highlighting these modifications, RefViewer allows reviewers to focus on the broader implications of the refactoring, reducing the need for manual effort and minimizing the risk of overlooking important behavior changes. Our solution enhances the current state of the art by providing a comprehensive view of all relevant code modifications, which helps make code reviews faster and more thorough.

We conducted a comprehensive evaluation of RefViewer to understand its impact for understanding code modifications in commits with refactorings. The study involved assessing the tool's performance in correctly classifying refactoring-related modifications and understanding the effort reduction in reviewing floss refactorings. Additionally, we analyzed the extra overhead introduced by the tool and explored developers' perceptions of refactoring-related modifications. To achieve this, 11 developers, with solid experience on refactoring, participated in the evaluation. They assessed 48 distinct floss refactoring instances (classified as easy, medium, and difficult) from 28 Java open-source projects. Each instance was reviewed by two developers, resulting in a total of 96 evaluations. The study aimed to balance statistical rigor with practical constraints, ensuring a thorough assessment of RefViewer's integration into the development workflow.

The evaluation results showed that developers confirmed the tool identified refactoring-related modifications correctly in 80.2% of the instances. Also, RefViewer consistently reduced the effort required to identify modifications, saving time across all difficulty levels, without adding overhead to the review process (execution time was consistently under 9 seconds). Finally, our tool positively aided developers on understanding the refactoring impacts in nearly half of the cases, highlighting its potential to enhance code review efficiency.

4.2 RefViewer: Refactoring Tool Visualizer

Identifying and visualizing code modifications related to refactorings in a commit can be challenging, particularly when dealing with many code modifications [92]. To address this challenge, we present RefViewer, a

browser extension that enables the visualization and highlighting of code modifications related to refactorings directly within GitHub commits. All source code is available at [96].

4.2.1 RefViewer Architecture

Figure 4.1 presents RefViewer's architecture. The architecture is composed of three parts, the Browser Extension, an intermediate HTTP Server, and the RefViewer Classification. In a nutshell, once the developer opens a commit, the browser extension automatically detects and makes a request to RefViewer's HTTP Server to obtain the refactoring-related modifications. Then, the HTTP Server runs the RefViewer's classifier. Finally, all the classification data is returned to the extension, which displays the classification to the developer.

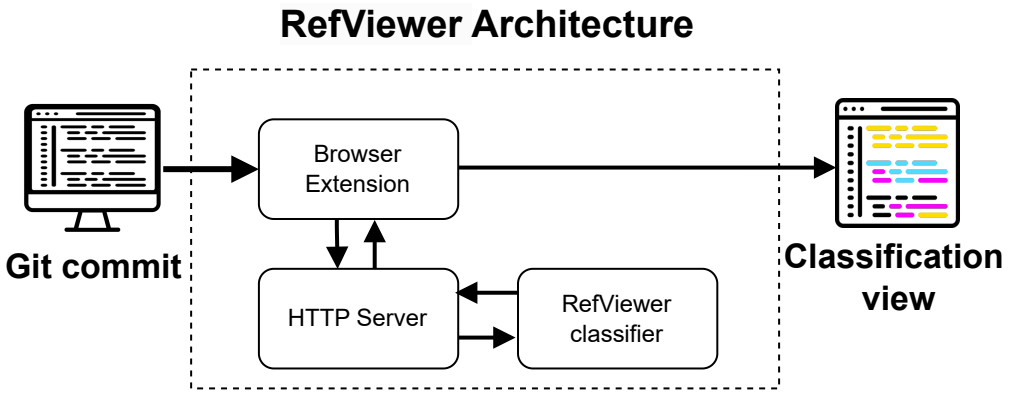

Figure 4.1: RefViewer Architecture

Browser Extension. The browser extension modifies the developer's browser view to include the RefViewer interface. It alters the HTML by identifying code modifications indicated by GitHub and updating their appearance. Additionally, the extension provides an interactive interface that allows developers to configure settings, such as selecting the refactoring instance and enabling or disabling certain visualizations. Finally, the extension communicates with the intermediate HTTP server, passing the commit hash and the Git URL to retrieve the necessary data for visualization.

HTTP Server. The server facilitates communication between the extension and the classifier. This setup allows the classifier to be used in any development environment, such as GitHub or even plugins for integrated development environments (IDEs).

RefViewer Classifier. Finally, the classifier is the core of the tool. It relies on two primary tools, *RefactoringMiner* [93] and *GumTree* [71]. RefactoringMiner identifies refactoring instances within the code, while GumTree detects code modifications in terms of the abstract syntax tree (AST). Once the refactoring is located and all AST-level modifications are identified, the classifier categorizes the code modifications into three groups: refactoring, refactoring-related, and non-refactoring modifications.

1. *Refactoring modifications* are the modifications considered default for each refactoring type. These modifications are part of refactoring catalogs [1, 2]. For instance, the creation of a new method containing the extracted code during an Extract Method.

2. *Refactoring-related modifications* are code modifications that interact with a refactoring modification and can be affected if the refactoring is not completely correct. These modifications are not necessarily listed as part of the refactoring in popular catalogs, and can include any type of modification. In this category, it is not mandatory to preserve the behavior. For this study, we considered that the interaction between refactoring-related and refactoring modifications can occur in three different ways: (i) The modification is part of the declaration of a refactored method. For instance, in an extract method refactoring, when the original method is changed from public to private. (ii) Sharing variables in common with the refactoring modifications. For instance, when a new modification responsible for providing a new functionality uses a variable involved in the refactoring modifications. (iii) Calling one of the refactored methods of the evaluated refactoring type.

3. *Non-refactoring modifications* are the modifications that do not interact with any other modification from the previous groups. Most of these modifications are purely related to other purpose, such as new functionalities or bug fixing.

Finally, for each modification, the classifier determines a *distance* value. This value range from 0 to 99. The distance value is not determined by number of lines but depends on how many steps are necessary to a modification "touch" any refactoring modification. Low distance values mean that the modification is strongly related to the refactoring instance. For example, all the modifications categorized as refactoring ones are distance 0. Then we have the refactoring-related modifications that are in the distances from 1 and above. For instance, if a modification invokes a refactored method, this modification will be classified

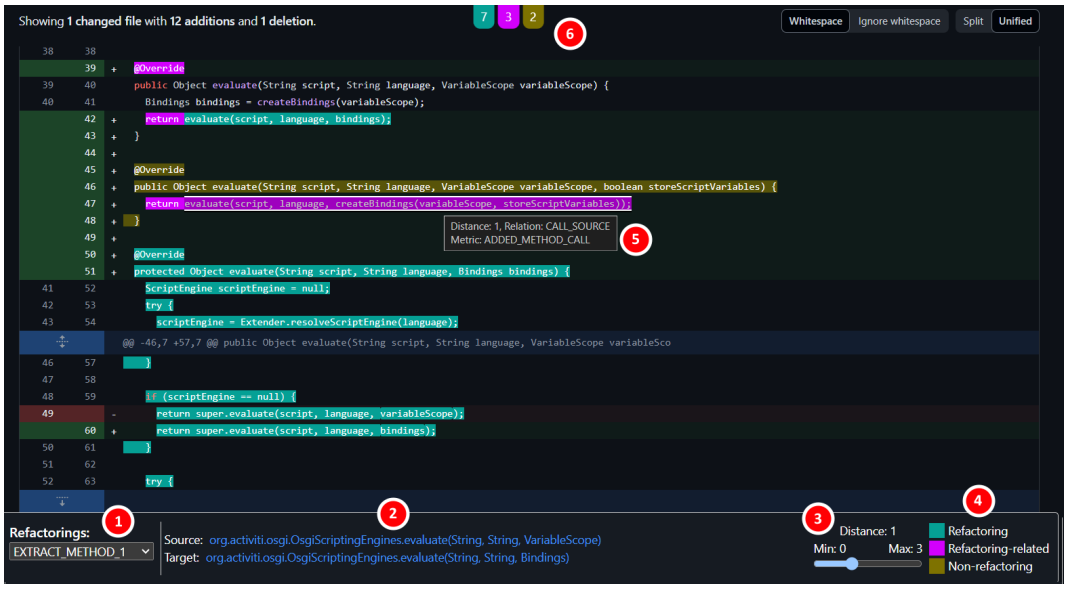

Figure 4.2: RefViewer Browser Extension Interface

as distance 1, since this call directly touches the refactoring modification. In this way, if the return of this invocation is attached to a new variable, the creation of this variable will be classified as refactoring-related with distance 2. This is because it will first touch the method invocation (distance 1) before touching the refactored method (refactoring modification), requiring 2 steps. Finally, we have the non-refactoring modifications that are in the distance 99.

4.2.2 Interface and Functionalities

The extension's interface is designed to present the classification information in a clear and intuitive manner. Users can view the modifications highlighted in the code, with refactorings, refactoring-related modifications, and non-refactorings all color-coded for easy identification. Figure 4.2 presents the interface provided by the extension when a refactoring is detected. The circles numbered from one to six indicates the possible interaction, described as follows:

- ❶ **Refactoring Instance Selection:** The extension allows the selection of a specific refactoring instance to be analyzed. Even if multiple instances are detected, only one will be analyzed per time.
- ❷ **Source and Target Methods:** The interface displays the main methods related to the refactoring, with the source method representing the code before the refactoring and the target method representing the code after the refactoring. Users can click on the method names to

navigate directly to them, even if they are collapsed by GitHub. In this case, the navigation will be to the closest line to the respective method.

- ❸ **Distance Selector:** The distance between refactored and non-refactored modifications can be adjusted. The interface initially presents the modifications within a maximum distance of 1, but this distance can be increased to include more related modifications as needed for the experiment.
- ❹ **Modifications Viewer:** A legend is presented in the interface, indicating the colors corresponding to each type of modification. Additionally, it is possible to hide the highlights for each category by clicking on their respective names in the legend.
- ❺ **Modification Tooltip:** When moving the mouse over a modification it will display a tooltip indicating the distance, the relation type and the metric name (type of modification) associated to that code modification.
- ❻ **Not Visible Modification Indicator:** When the visible screen does not show all classified modifications, an indicator will be shown on the top or bottom of the screen. There will be three possible indicators with the respective colors. It is possible to click on the indicator to navigate to the closest respective modification type in the top or bottom.

These functionalities allow developers to quickly analyze code modifications without the need for extensive reading or deep understanding of the commit itself. Also, they enable the navigation directly to specific methods and modifications, simplifying the code review process. Then, the distance selector offers flexibility to adjust the depth of the analysis, allowing developers to focus only on the most relevant modifications or expand the analysis as needed. Finally, the visual indicators for non-visible modifications ensure that no important modifications are overlooked, providing a comprehensive view of the modifications introduced in the code.

4.3 Study Design

We conducted a comprehensive evaluation of RefViewer to understand the impact of our proposed tool on the refactoring process. This evaluation aimed to assess multiple facets of the tool's integration and utility in the development workflow. Thus, to observed how effectively the tool facilitates the visualization of refactoring-related modifications without introducing overhead on the code review practice, our study was guided by the following research questions (RQs):

RQ1: What is the performance of the RefViewer? Despite being a key question for evaluating the proposed tool, determining a single accuracy value to answer it is not straightforward. This complexity arises mainly due to two factors. First, the classification process considers various code segments at the token level (*e.g.*, variable names, types, parameters, reserved words, among others). For a typical commit, the number of tokens can reach thousands, which makes it challenging, especially in terms of effort, to determine whether a classification is perfectly accurate or if there are misclassified tokens. Moreover, the concept of refactoring-related modifications is subjective to the developers' perspectives. Each developer may have a different maximum distance they consider as part of the refactoring, which might not align with the tool's initial visualization. To mitigate these factors and establish a method for evaluating the tool's performance, we adopted a two-step assessment. First, developers provided their perspective on whether the tool correctly classified the evaluated instances by indicating whether the tool failed to classify any related-modification. Then, to evaluate the extent of misclassified tokens, developers rated their level of agreement with the classification proposed by the tool.

RQ2: How does RefViewer affect the effort to review floss refactorings? To address this question, we evaluated the challenges developers face when identifying the occurrence of refactoring. We also measured the time developers would spend performing this task manually, which could be saved by using the tool. This evaluation will highlight the tool's potential to streamline the review process and reduce the effort required from developers. By minimizing the manual effort, the tool not only speeds up the development process but also allows developers to focus more on high-level tasks.

RQ3: What is the additional overhead when using RefViewer? In this question, we investigated what is the overhead that the RefViewer introduces in terms of time. We investigate the time the tool needs to collect data, execute its functions, and present the visualization to the developer. Understanding this overhead is crucial for assessing the tool's practicality and how it integrates into the development process without causing significant delays.

RQ4: What are developers' perceptions about refactoring-related modifications? The final research question focuses on understanding the qualitative aspects through developers' thoughts regarding reviewing refactoring instances, as well as how the RefViewer affected them. We discussed the main challenges developers face when reviewing floss refactoring instances. Additionally, we investigated the extent to which modifications are

still considered part of the refactoring in terms of *distance*. This result provides important insights for improving tool support in code review, enabling developers to more accurately determine the extent to which modifications are still part of the refactoring, thereby streamlining and enhancing the review process.

4.3.1 Data Collection

We evaluated 48 distinct refactoring instances from 28 Java open-source projects. To ensure comprehensive coverage, we included eight instances for six different refactoring types, encompassing method-level refactorings such as Extract Method, Inline Method, and Rename Method, as well as structural refactorings like Move Method, Pull Up Method, and Push Down Method. This selection was made based on the broad scope and frequent use of these refactorings in practice [7, 39, 40, 68]. Method-level refactorings are commonly employed to enhance code readability and maintainability within individual methods, while structural refactorings address the organization and hierarchy of classes, reflecting their widespread application in codebase management.

Each refactoring instance was assessed by two different developers, resulting in a total of 96 evaluations. Altogether, 11 developers, including practitioners and academics, participated in the evaluation process, ensuring that each instance was reviewed thoroughly. This approach was chosen to balance the need for statistical rigor with practical constraints, including the availability of the 11 developers and the manageable workload per developer. The following sections describe each step of the experiment.

For each refactoring type, the eight chosen refactoring instances was carefully divided based on difficulty into three easy, three medium, and two hard instances. The difficulty classification was determined using the following criteria:

$$\text{Difficulty} = \begin{cases} \textit{easy} & \text{for } \textit{rPercent} > 80 \text{ and } \textit{refInstances} = 1 \\ \textit{medium} & \text{for } \textit{rPercent} \geq 50 \text{ and } \textit{refInstances} \leq 2 \\ \textit{hard} & \text{for } \textit{rPercent} < 50 \text{ and } \textit{refInstances} \geq 2 \end{cases}$$

In this context, *rPercent* represents the percentage of code modifications that are not associated with the refactoring being analyzed. These modifications are either part of other refactorings than the evaluated one (in the case of medium and hard difficulties) or other software activities. Also, *refInstances* represents the number of refactorings that occurred in the same commit. By structuring our evaluation in this manner, we ensured that our tool was validated across a range of difficulty levels for each type of refactoring. This

approach provides a thorough assessment of the tool's performance, demonstrating its applicability and robustness in identifying refactoring-related modifications under various conditions.

Project selection. We selected projects based on their relevance and popularity using specific Git metrics [54, 95]. Our selection criteria covered a range of projects, from new to old, while excluding non-software, personal, or toy repositories. The criteria included: (i) *Projects with a minimum of 500 commits*, ensuring substantial development activity and avoiding toy projects; (ii) *Projects that had commits within the last six months*, to ensure that the projects are actively maintained and reflect current development practices; (iii) *Projects with at least 500 forks*, indicating significant community engagement and frequent refactoring efforts from the community, which enhance the code's structure and maintainability; and (iv) *Projects with a minimum of 3500 stars*, a threshold much higher than those in previous studies [7, 39, 40], serving as an indicator of popularity.

Refactoring instances selection. Regarding the refactoring instances, We opted to use RefactoringMiner [93] again to collect the instances used in the evaluation phase. The motivation for using RefactoringMiner is its high precision and recall [33, 93]. Also, for the evaluation phase, we initially gathered over one thousand of refactoring instances. We then randomly selected instances according to their difficulty level in order to have a similar amount between each difficulty. Finally, a last manual verification step was conducted to ensure that the instances encompassed varying commit sizes, with different numbers of modifications and affected files for each difficulty level.

Table 4.1 presents the list of instances used in our experiment. The first two columns indicate the refactoring type and the instance difficulty level. The remaining columns present the instance *Id*, the project name and the commit short hash, respectively.

4.3.2 Developer Characterization

Before the evaluation, each developer received a unique ID and fill out a characterization form. They answered questions about their experience with code development, as well as their expertise in both refactoring practice and review. Regarding the expertise, the developers answered the questions considering a scale from 1 (beginner) to 5 (very experienced).

We observed that 81.8% (nine) of the developers possess at least five years of experience in software development, with 54.5% (six) of these developers having more than ten years of practice in software development. The results

also indicate a high level of expertise among the participants in refactoring practices. The majority of responses were 4 or 5, with 36.4% (four developers) for each of these ratings. Only 27.3% (three developers) rated their experience as 3. Regarding their experience with code reviews, the responses similarly reflect a strong background. Three developers (27.3%) rated their experience as 5, while four developers (36.4%) rated it as 4 and another four (36.4%) rated it as 3. This distribution demonstrates a robust familiarity with refactoring practices among the developers, highlighting their confidence and capability in this area. Also, the results suggest that the developers maintain a substantial level of expertise for code review.

4.3.3 Evaluation Experiment

The experiment was divided into three stages. All questions and answers are available at [96].

Preparation stage. In this step, we provided developers with supporting material that detailed each concept necessary for understanding the questions. Following this, we offered an explanation of the interface of the tool, as well as the entire procedure required to set up and use the tool. Finally, each developer was also provided with a list of commits in which at least one refactoring had been detected and a step-by-step guide for each task they needed to perform. The developers were free to make any question and have any necessary information that they considered important before starting the experiment. Once the developers were ready to initiate, they repeated the first and second stage for each received commit.

Control stage. Upon opening a specified commit, the proposed tool indicated on the developer's screen only the type of refactoring they needed to analyze, along with details of the instance, such as the classes and methods involved in the refactoring. This information was provided similarly to the output of RefactoringMiner, focusing only on the default code modifications for each type of refactoring. As the first task, the developer indicated, on a scale of 1 (Strongly Disagree) to 5 (Strongly Agree), the level of agreement with the refactoring identified by the proposed tool. Next, the developer listed all the code modifications they considered related to the refactoring. To measure the effort required for this activity, developers were instructed to perform the task without breaks and to record the start and end times. After listing the modifications they deemed related to the refactoring, the developer answered questions regarding the difficulty level of the task, the main factors that made the task easy or difficult, and their confidence level in their response.

Treatment stage. After answering the questions in the first stage, the developers were instructed to unlock the full view of the proposed tool. This highlighted the code modifications indicated as default to refactoring, related to refactoring, and unrelated modifications, as shown in Figure 4.2. Before answering the next set of questions, the developers took the necessary time to evaluate and navigate through the proposed visualization to familiarize themselves with the classifications provided by the tool. Once ready, the developers answered questions regarding their level of agreement with the proposed classification and which modifications they believed were incorrectly classified. Additionally, developers were asked to adjust the visible distance value to find a distance at which modifications are still exclusively related to the refactoring instance. Finally, they were asked if they changed their opinions about the previous classifications after viewing the proposed classification and their opinion on the reduction of effort when using this visualization.

4.4 Results and Discussion

This section focus on answering and discussing the four research questions.

4.4.1 RQ1. RefViewer Performance

To evaluate the effectiveness and answer RQ1, developers answered two distinct questions. The first question aimed to understand whether developers believed that any modification related to refactoring was mistakenly classified as non-refactoring. Furthermore, we asked to the developer whether any code modifications were classified as related when they should not have been. Regarding the first question, we observed that in 77 out of 96 instances (80.20%), developers indicated that RefViewer was able to identify as related all the modifications that the developers considered to be related to refactoring. On the other hand, when answering the second question, we noticed that in only 20 out of 96 instances (33.3%), developers were able to list at least one code modification they disagreed was related to refactoring.

Based on the developers' responses, we can define three reasons that explain the extra modifications categorized as refactoring-related. The first reason is that some code modifications interact with the refactoring but do not necessarily represent any refactored code functionality, such as annotations. Thus, this set of modifications tends to be considered as false positives by developers. Second, the visualization allows developers to see all *distances*, which

Table 4.1: Instances Evaluated During Experiment

		Id	Project	Commit
Extract Method	Easy	T1-F1	Activiti-Activiti	1218270
		T1-F2	Activiti-Activiti	14721cd
		T1-F3	airbnb-lottie-android	f970d3a
	Medium	T1-M1	abel533-Mapper	0bbdd89
		T1-M2	alibaba-arthas	7442fba
		T1-M3	alibaba-arthas	ec6456c
	Hard	T1-D1	azkaban-azkaban	b0adb99
		T1-D2	azkaban-azkaban	bd9e6e4
Inline Method	Easy	T2-F1	discord-jda-JDA	36ee5ab
		T2-F2	dreamhead-moco	0e6a290
		T2-F3	dreamhead-moco	41fe372
	Medium	T2-M1	dreamhead-moco	424f653
		T2-M2	Genymobile-gnirehtet	660d6a8
		T2-M3	google-android-classyshark	e436fc4
	Hard	T2-D1	google-auto	3f69cd2
		T2-D2	google-google-java-format	33fc5ba
Move Method	Easy	T3-F1	signalapp-Signal-Server	33c88ec
		T3-F2	PhilJay-MPAndroidChart	8e193e2
		T3-F3	osmandapp-OsmAnd	c861265
	Medium	T3-M1	swagger-api-swagger-core	7416ff8
		T3-M2	StarRocks-starrocks	9a26f19
		T3-M3	StarRocks-starrocks	7646554
	Hard	T3-D1	Activiti-Activiti	00a1451
		T3-D2	NanoHttpd-nanohttpd	6aa9777
Pull Up Method	Easy	T4-F1	apache-storm	f594c20
		T4-F2	apache-zookeeper	6664679
		T4-F3	facebook-fresco	9395f70
	Medium	T4-M1	Tencent-APIJSON	8592367
		T4-M2	alibaba-jetcache	e22b1ae
		T4-M3	apache-shenyu	0cf7713
	Hard	T4-D1	apache-shenyu	0f3a09d
		T4-D2	apache-shenyu	32a4229
Push Down Method	Easy	T5-F1	Activiti-Activiti	af10b56
		T5-F2	discord-jda-JDA	2b0eff5
		T5-F3	discord-jda-JDA	a77218c
	Medium	T5-M1	discord-jda-JDA	b579b3d
		T5-M2	Doikki-DKVideoPlayer	fbb79c0
		T5-M3	Activiti-Activiti	a48199c
	Hard	T5-D1	Activiti-Activiti	47cfe93
		T5-D2	Activiti-Activiti	4c22d56
Rename Method	Easy	T6-F1	abel533-Mapper	32fd500
		T6-F2	Konloch-bytecode-viewer	b2f7fcb
		T6-F3	LMAX-Exchange-disruptor	f0fa2f8
	Medium	T6-M1	Activiti-Activiti	3756580
		T6-M2	mybatis-mybatis-3	2188eb2
		T6-M3	mybatis-mybatis-3	2a68f4f
	Hard	T6-D1	Tencent-QMUI_Android	ab82080
		T6-D2	Tencent-QMUI_Android	8149f13

can result in very high distant code modifications far from the refactorings, either by lines of code or by semantic context of the code. This increases the likelihood of developers indicating some code modifications as non-refactoring. Finally, the third reason is caused by a current limitation of the tool in displaying code modifications related to AST nodes that contains a block of code, for example, the modifications `ADD_CLASS`. In this code modification, the tool may understand that all the code created in the new class would be related to refactoring if the creation of the class is also related. Similarly, smaller modifications that still have a block of code such as if and while statements can have the same behavior when displayed.

After indicating which code modifications related to refactoring were misaligned, developers were asked about their level of agreement, on a scale of 1 (Strongly Disagree) to 5 (Strongly Agree), with the overall classification made by the tool. To answer this question, we grouped the data by difficulty level (Figure 4.3) and by type of refactoring (Figure 4.4).

Regarding agreement by difficulty level we observed that, in instances classified as easy, developers exhibit a high level of agreement, with most values concentrated in four and five. The median agreement is four, suggesting that a substantial portion of developers strongly agree with the indicated relationships. Regarding the medium level, The agreement is more evenly distributed between two and four, but with the same median (four) then the Easy difficulty level. This indicates a moderate consensus among developers, with a broader range of opinions on the relationships indicated by the tool. Finally, for the hard level, the agreement level is varied, ranging from 1 to 5, with a median of three, slightly lower than the other levels. This spread suggests that developers have more diverse opinions on the relationships at this difficulty level, with some strong disagreements and agreements.

Analyzing by refactoring type (Figure 4.4), we observed that the developers' agreement with the relationships indicated by the tool varies depending on the type of refactoring. For instance, when evaluation an Extract Method, developers show a high level of agreement, with most ratings falling between 3 and 5 with the median of four, indicating strong consensus among developers about the tool's relationship suggestions. On the other hand, some refactoring type such as Pull Up Method shows a wide distribution from one to five, with a median of four. This indicates a broad range of opinions, but with a tendency towards agreement. The same behavior is observed for the other refactoring types.

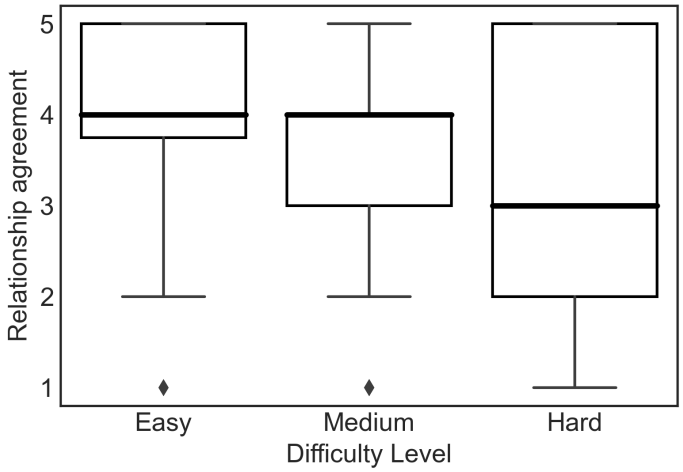

Figure 4.3: Relationship Agreement by Difficulty Level

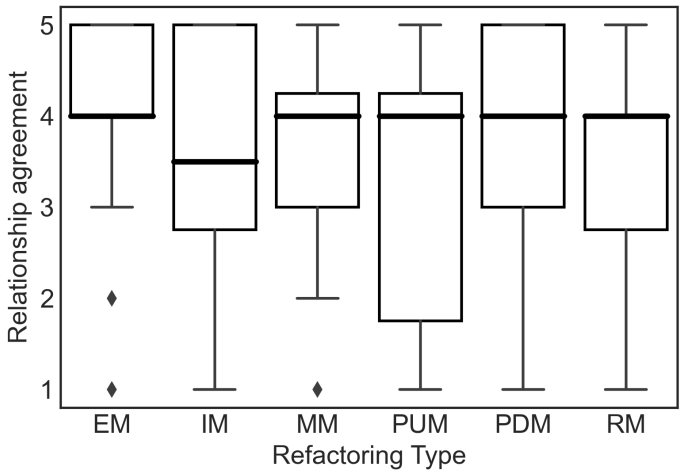

Figure 4.4: Relationship Agreement by Refactoring Type

RQ$_1$: The results indicate that in 80.20% of the instances, developers believed the tool identified all modifications related to refactoring. The relationship agreement analysis revealed that developers generally showed a high level of agreement with the RefViewer classification at the Easy difficulty level, with increasing difficulty levels correlating with a greater dispersion of opinions at higher difficulty levels. For refactoring types, Extract Methods had higher level of agreement among developers, while Pull Up Method showed more dispersed opinions and Inline Methods presented a lower median agreement, indicating a more mixed response from developers regarding these refactoring classifications.

4.4.2 RQ2. RefViewer Effort Reduction

In order to evaluate effort, we observed two aspects. First, we observed the necessary time to analyze and identify the refactoring-related modifications manually. Then, we asked the developer following the same scale, than the previous RQ, from 1 to 5, their opinion about their perception regarding the effort reduction.

Figure 4.5 indicates the amount of time developers spent manually classifying code modifications as related or not to the refactoring, which could be avoided by using the proposed tool. The median times for all three difficulty levels are 3.5 for easy instance and 4 for medium and hard instances. This result indicated that the median time savings are relatively consistent irrespective of the complexity of the task. However, at hard level, we observed that this activity can take up to 14 minutes at the upper limit. Finally, the presence of numerous outliers, particularly in the Medium and Hard categories, suggests that some developers experience significantly longer classification times. This variance indicates that while the tool can generally reduce the manual classification effort, the extent of this benefit can vary considerably among developers and tasks. This variance is potentially due to individual differences or the specific context of the code affected by the refactoring. Finally, for some medium and hard instance levels, some developers mentioned to be impossible or extremely hard to classify the modifications manually.

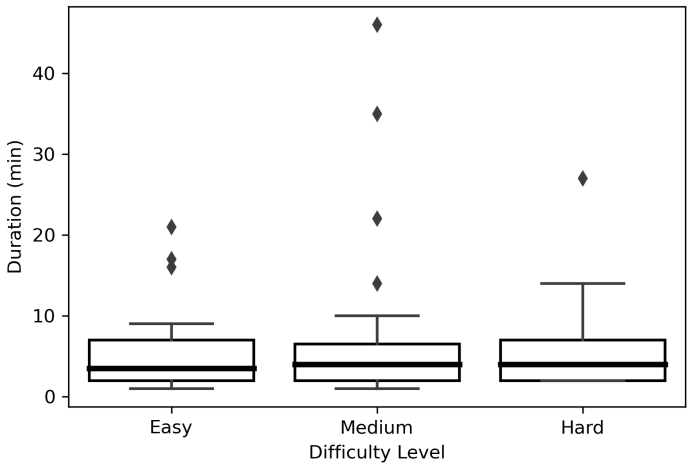

Figure 4.5: Manual Classification Required Time

Regarding the perceived reduction in effort by developers when using the provided visualization, Figure 4.6 indicated that the interquartile range and median is equal for all levels. This figure suggests that the variability in effort

reduction ratings is consistent. This result indicated that developers generally agree the tool reduces effort effectively, regardless of difficulty. However, the lower whisker extends to 1 in all categories, showing that some developers perceive a minimal effort reduction for some instances. For some of these instances, the developers mentioned that the refactoring was to simple, thus the use of a tool was not necessary. On the other hand, in few instances, developers did not agree with the classification proposed by RefViewer or with refactoring existence. In these instances, therefore, the visualization did not help the developer to identify the refactoring-related modification.

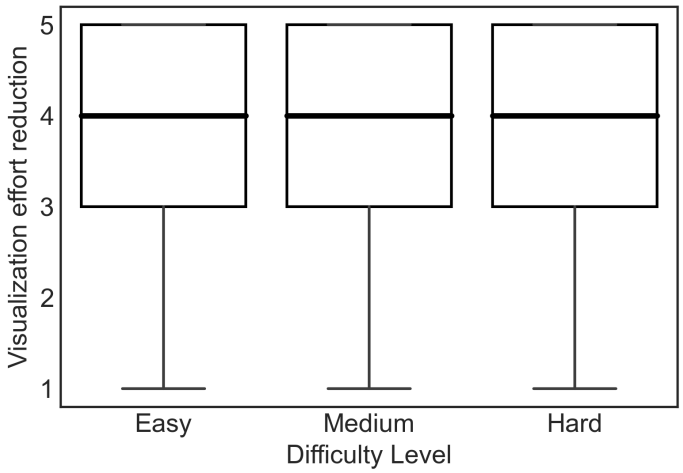

Figure 4.6: Effort Reduction by Difficulty Level

Finally, Figure 4.7 presents the reduction in effort brought by the RefViewer visualization for the six different refactoring types. The data indicates a consistent reduction in effort across the different types of refactorings, with most medians being equal to 4. However, there are variations in the distribution of effort reduction for each refactoring type. For instance, Pull Up Method, Push Down Method, and Rename Method, we observed a broader range of perceived effort reduction, suggesting that the impact of the visualization tool on these refactoring types can vary more significantly among developers. This variability might be due to the different complexities inherent in these types of refactoring or the specific circumstances of each task. For instance, developers indicated that for Pull Up and Push Down methods, the visualization only included one moved method in the hierarchy when it was expected to include all moved method. Outliers in the Inline and Move methods categories also indicate that, in some cases, developers experienced extreme differences in effort reduction, pointing to occasional discrepancies in how the visualization tool impacts different refactoring scenarios.

RQ$_2$: The proposed visualization reduces the effort needed to identify refactoring-related modifications, enabling developers to save time and providing a consistent perception of reduced effort. While the median time savings are similar across all difficulty levels, the reduced effort can vary depending on the type of refactoring and the specific circumstances of each task.

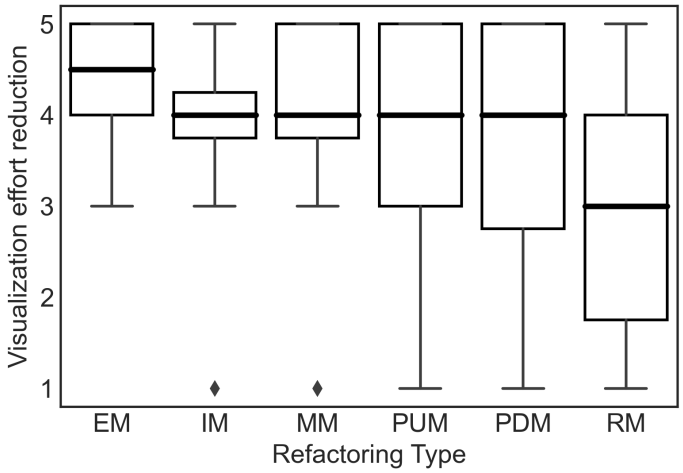

Figure 4.7: Effort Reduction by Refactoring Type

4.4.3 RQ3. RefViewer Runtime Overhead

To answer RQ3, we ran the tool three times for all 48 instances. Our testing was conducted on a computer with a Core i7-11800H processor and 24 GB of RAM. We chose to execute it three times to mitigate any potential communication bottlenecks between the browser extension and the server. After completing all executions, we collected the time required from the server request to the construction of the visualization.

Figure 4.8 presents the results of the executions. In this figure, we divided the required time by difficulty level and indicated the minimum time, maximum time, and average of all times across the three executions for each difficulty level. The analysis of execution times, measured in seconds, reveals only minor differences. The Hard instances have the highest mean execution time (4,97 seconds), followed closely by Easy (4,63 seconds), with Medium instances having the shortest times on average (4,57 seconds). Although Easy instances have a slightly higher maximum execution time (8,91 seconds) compared to Hard and Medium, the minimum times are quite similar across all levels.

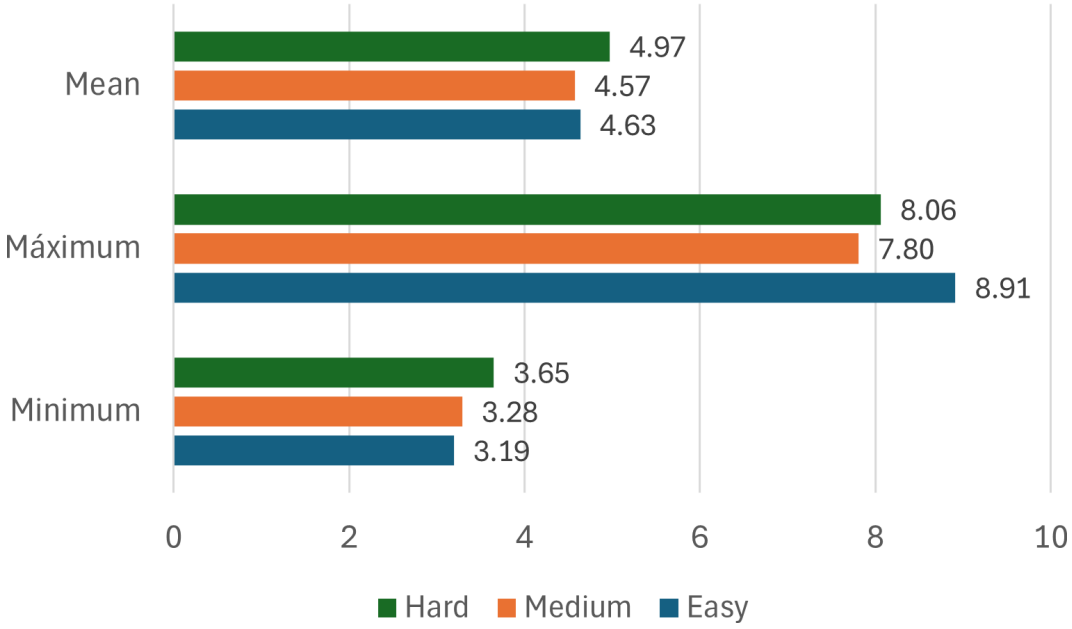

Figure 4.8: Execution Time in seconds by Difficulty Level

This suggests that the tool performs consistently, with only minor variations regardless of difficulty. Such minimal differences indicate that the tool is robust across different scenarios, highlighting its stability and efficiency.

Despite the rapid initial interface setup, some developers have reported bottlenecks with the tool when redrawing the interface as they adjust the maximum distance considered by the tool. In our tests, this delay was less than one second; however, in larger instances, it may hinder developers from accurately selecting the intended *Distance* value.

> **RQ$_3$**: In summary, the tool's execution times are fairly uniform across different difficulty levels, indicating that it can handle varying complexities with reliable performance. Also, the necessary time is not over 9 seconds, which is not a practical delay to the start of the code review step.

4.4.4 RQ4. Developers Thoughts About Refactoring Review

The final research question aimed to explore the concerns and challenges that developers have when reviewing commits that include refactorings and how the proposed visualization tool influenced their code review process. Initially, developers were asked to share their thoughts on the importance of evaluating refactorings in isolation during a code review. The majority of the developers, 72.8% (8 developers), indicated that they consider very important

to evaluate the occurrence of refactoring in isolation before reviewing other code modifications.

We even asked the developers to provide what are their concerns when reviewing refactoring instances. Figure 4.9 highlights the key concerns developers. Developers' were concerns about Behavior and Affected Code, with 8 and 7 mentions, respectively. This suggests that developers prioritize ensuring that refactorings do not alter the intended behavior of the code and that they are particularly mindful of which parts of the code are impacted by the modifications.

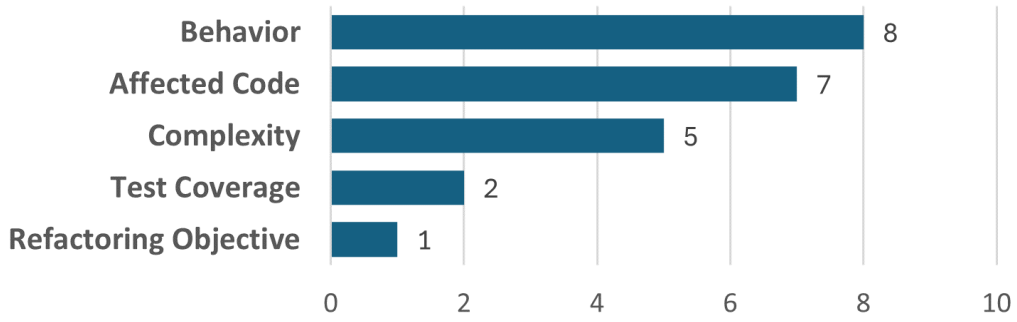

Figure 4.9: Developers' Concerns About Refactoring

Other concerns include Complexity, mentioned 5 times, indicating that developers are also aware of the complexity introduced or reduced by the refactoring. Test Coverage and Refactoring Objective were mentioned less frequently, with 2 and 1 mentions, respectively, suggesting these aspects are of lower immediate concern but are still considered in the review process. These results underscores the necessity of having a dedicated tool, like the proposed visualization, to help developers clearly distinguish refactorings from other modifications. Such a tool enables a more structured and focused review, allowing developers to assess the refactoring's impact isolated.

Once we understood the developers' concerns when reviewing refactorings, we aimed to explore the challenges faced by them. To investigate this, we asked developers about the difficulties they encountered in manually identifying refactoring-related modifications. Figure 4.10 lists the challenges as long as the percentage of the total instances in which developers mentioned each challenge. These insights are crucial for understanding how to better support developers in identifying related modifications and reducing the manual effort required in this process.

Developers pointed out that one of the main challenges when reviewing refactorings is dealing with the number of altered files and lines of code. This

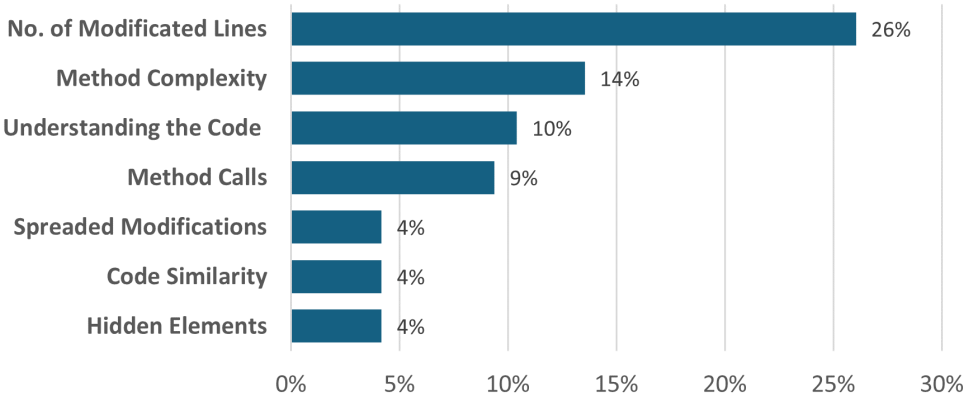

Figure 4.10: Developers' Concerns About Refactoring

challenge was faced in 26% of the instances. The following challenges were the complexity of the refactored method (14%) and the need to thoroughly understand the code (10%) to effectively review the refactoring application. These challenges imply that without adequate tool support, code reviews can become overwhelming and less effective. The tool mitigates the main challenge by visually distinguishing the refactoring-related modifications. This visualization helps developers to focus on the most relevant code modifications. Additionally, it can streamline the review process for complex methods and improve code comprehension, making it easier for developers to navigate and assess the refactoring more efficiently.

Finally, we asked developers whether using RefViewer to visualize the refactoring changed their thoughts. Specifically, we wanted to know if RefViewer was able to expand or alter their understanding of the relationships between modifications in the context of refactoring. Our results showed that all developers mentioned that, in hard instances, the visualization changed their initial opinion about the impact and affected code of the refactoring in terms of refactoring-related modifications. This shift in perspective occurred in 46.9% of the total refactoring instances reviewed, demonstrating how the tool can reveal overlooked modifications and help developers better assess the full extent of refactoring, leading to more informed decisions during code review.

RQ$_4$: The results of this research question underscore the importance of evaluating refactorings in isolation and highlight the effectiveness of the visualization tool in improving the refactoring code reviews. The tool not only addressed developers' main concerns and challenges but also influenced their understanding of the impact of refactoring in nearly half of the cases, showcasing its potential to enhance code review processes.

4.5 Related Work

Refactoring is a critical topic that has been extensively studied in the literature concerning quality, complexity, and potential adverse effects [8–12,19], as well as within the context of floss refactoring [14,17]. However, studies aimed at understanding and identifying refactoring-related modifications remain limited. Oliveira *et al.* [27] introduced the concept of *customized refactorings*, where code modifications serve as connectors linking refactorings within the refactored code context. In addition to this study, Moreira *et al.* [20] evaluated the Extract Method refactoring within the context of floss refactoring, seeking to understand which code modifications accompany this refactoring. However, these studies did not aim to understand the relationships between these modifications, nor did they offer a visualization of these additional modifications in the context of code review.

Regarding the detection of refactorings in a project's history, two tools have been widely used in the literature: RefDiff [6] and RefactoringMiner [33, 93]. While RefactoringMiner supports more types of refactorings with better efficiency and accuracy for most of them, RefDiff stands out for its support for multiple programming languages. Although these tools are essential for detecting refactorings, they have limitations when applied to code review since their primary goal is historical detection. Consequently, these detection tools have been used as the foundation for developing other tools that assist in code review.

RefDiff was employed in the Brito *et al.* study [26], which proposed the RAID tool to support refactoring reviews. Like RefViewer, RAID is a browser extension aimed at reducing cognitive effort in detecting and reviewing refactorings from textual diffs. Regarding the use of RefactoringMiner, the literature presents two distinct tools. First, Tsantalis *et al.* proposed a tool called *Refactoring Aware Commit Review*, also as a browser extension. In this tool, Tsantalis identifies refactorings in open-source Java projects and lists the refactoring activities in GitHub diffs, highlighting the identified refactoring-related code modifications. Additionally, the IntelliJ IDEA plugin proposed by Kurbatova *et al.*, called RefactorInsight [94], also relies on RefactoringMiner. Through RefactorInsight, developers can access a visualization that auto-folds refactorings, allowing them to focus on behavior-altering code modifications.

Finally, Ge *et al.* [18] conducted a study with 35 developers to evaluate and understand the motivations and challenges developers face when reviewing code refactorings. In this study, developers highlighted the importance of reviewing refactorings in isolation, as found in our study. Additionally, they

indicated that the presence of code refactorings in commits could slow down the code review process, increasing the effort required. Furthermore, in the same study, Ge *et al.* also introduced the ReviewFactor plugin as part of the Eclipse IDE. Through this plugin, developers could evaluate code modifications by choosing to highlight refactoring code modifications or other types of code modifications.

Although tools like ReviewFactor and RefactorInsight aid in visualizing refactorings either in commits or specific IDEs, they exclusively focus on default refactoring modifications. However, in the context of floss refactoring or customized refactorings [27], these tools are limited and may not provide the necessary information for an adequate refactoring analysis, as they do not indicate the relationships among refactoring-related modifications. Thus, unlike previous studies, RefViewer introduces new concepts such as a new classification of *refactoring-related modifications* and the notion of *distance* required for application in the context of floss refactoring. By classifying and highlighting modifications as refactoring-related, developers become aware of the impact that the refactoring under review has, which is crucial in the analysis and decision-making process based on the different relationships these modifications have. Moreover, the visualization with different distances allows developers to assess the scope of the refactoring. Lastly, RefViewer also presents markers at the top and bottom of the screen, indicating refactoring-related code modifications that are outside the current code view, enabling developers to navigate more practically and preventing any modifications from being overlooked.

4.6 Threats to Validity

We describe here the threats to validity and their mitigation.

Internal and Construct Validity. RefViewer relies on state-of-the-art tools to collect all the information it needs. Thus, the accuracy and efficiency of these tools directly affect RefViewer's performance. To minimize this impact, we opted for RefactoringMiner [7, 68], which has an effectiveness rate of 87.2% and 98% precision [33], proving to be superior in detecting refactorings in Java projects. Furthermore, we evaluated the level of agreement among developers who participated in the study regarding the existence of the refactoring they were analyzing. Except for Inline Method, which had a median of 4, all other refactorings had a median of 5, indicating a high level of agreement among developers regarding the existence of the refactoring.

Additionally, in our experiment, developers focused on one refactoring

at a time. This approach might not fully capture scenarios where multiple modifications occur simultaneously [40]. In these cases, refactorings can affect the same method or class. However, grouping the set of changes related to each individual refactoring could provide a comprehensive view of modifications in a multi-refactoring scenario. An improvement could involve enabling a complete visualization of this set without needing to switch between refactorings.

Finally, there are scenarios where refactorings occur across several commits. In such cases, the proposed visualization tool might not related code modifications to the refactoring. This limitation mainly arises because RefactoringMiner might also struggle to detect the refactoring occurrence. However, in these scenarios, developers would also not be necessary or requested to evaluate the refactoring based on the split commits, making visualization unnecessary.

External Validity. Despite RefactoringMiner version 2.2 being able to detect more than 50 types of refactorings [93], the majority of these refactorings are simple modifications with limited scope. Our hypothesis is that such refactorings tend to have fewer relationships with other modifications. Therefore, our study considers only six types of refactoring. These chosen types of refactorings affect the program structure differently at the method and class levels. Additionally, refactorings like Push Down Method and Pull Up Method directly impact hierarchies in the object-oriented programming (OOP) paradigm, which is one of the primary advantages of OOP languages.

In addition to the limited number of refactorings, our experiment included 48 evaluated instances. This number of instances might not be sufficient to ensure the external validity of our tool. To mitigate this issue, we considered different levels of difficulty and 28 distinct projects. This design choice allows the evaluated instances to cover various scopes and complexities, and distinct quality pattern from the different projects.

4.7 Conclusion

In this study we propose RefViewer, a tool, integrated into a GitHub environment, designed to assist developers in identifying and reviewing refactoring-related modifications within code commits. We evaluated the performance and impact of our proposed visualization tool through a experiment involving 11 developers from industry and academy. During the experiment, the developers reviewed code modifications without and with observing RefViewer visualization. Each developer was tasked with evaluating up to nine instances of code modifications, with each instance being reviewed by at least

two different participants. The scenarios presented in the experiment were carefully crafted to simulate realistic code review situations, including varying levels of difficulty and complexity in the code modifications.

Throughout the experiment, we measured several key metrics, such as the performance of identifying refactoring-related modifications, the time spent on reviews, and the cognitive load experienced by the developers. The results indicate that our tool significantly enhances the code review process by enabling developers to more easily distinguish between refactoring and non-refactoring modifications. The visualization provided by the tool not only improved the accuracy of the reviews but also reduced the time required to assess the impact of refactoring. Moreover, developers reported that the tool changed their initial perception of certain code modifications, leading to more informed and precise decision-making.

In summary, our study demonstrates that the integration of a visualization tool for refactoring identification can greatly improve the quality and efficiency of code reviews. The tool's ability to highlight and differentiate refactoring-related modifications ensures that developers can focus their attention on the most critical aspects of the code, thereby reducing the likelihood of overlooking important modifications.

5
Final Conclusions

Refactoring is a fundamental software engineering practice aimed at enhancing the structure and quality of code without altering its behavior [1, 2]. However, applying refactorings is a complex task that requires advanced knowledge of the code to prevent unintended side effects [9–12]. The practice known as floss refactoring, where refactorings are intertwined with other development tasks such as adding new features or fixing bugs, further complicates the identification of refactoring-related modifications [7, 14, 17].

Studies have investigated the frequency of floss refactoring occurrences their influence on overall software quality [7, 8, 17, 21]. However, there is a notable gap in understanding the code modifications beyond pure refactoring purpose and how these modifications relate to the default refactoring modifications during software maintenance and evolution [16, 20, 27]. This gap underscores the pressing need for automated tools that can accurately classify code modifications within a commit as either part of a refactoring or not [18], while also identifying their relationships.

In this thesis, we investigated the challenges developers face when performing and reviewing refactorings. We focused on understanding refactoring customizations and refactoring-related modifications. We also investigated the role of automated tools in facilitating the identification and review of these modifications. Each chapter contributes to a more comprehensive understanding of the refactoring process in real-world software development, offering both theoretical insights and practical tools to support developers.

First, we explored how developers customize refactorings, deviating from the default sets of modifications associated with specific refactoring types. Our study revealed that these customizations are not only common but also crucial for addressing specific project needs. We identified several limitations in existing IDEs, which currently do not provide adequate support for recurring customizations, potentially increasing the risk of unintended side effects. This result highlights the importance of enhancing refactoring tools to better accommodate the diverse scenarios developers face. In fact, we found that developers argue that explicit customization support would improve code quality and correctness.

Also, we examined and developed a tool to detect the occurrence of refactoring-related modifications in floss refactoring instances. Through the analysis of over 98,000 refactoring instances across 213 projects, we cataloged frequent code modifications that occur alongside refactorings. We also developed a method to measure their closeness to refactoring instances. This study showed that refactoring-related modifications are often concentrated in the refactored methods. However, these modifications can also extend the scope, affecting class-level modifications even during method-level refactorings. This finding underscores the importance of distinguishing between refactoring-related and unrelated modifications to better understand and manage the complexity of refactoring tasks. This distinction is also necessary to reduce the code review effort and complexity [18].

Finally, we developed a second tool called RefViewer. This tool is a visualization tool integrated into GitHub that helps developers identify and review refactoring-related modifications. Through an experiment involving 11 developers, we demonstrated that the tool significantly enhances the code review process, enabling developers to better distinguish between refactoring and non-refactoring modifications. The tool improved developers' ability to accurately identify refactoring-related modifications and reduced the time required for reviews. This finding indicates the tool potential to make the review process more efficient and enhance overall productivity. By providing a clear visual representation of refactoring-related modifications, RefViewer empowers developers to make more informed decisions during code reviews, ultimately improving software quality.

In conclusion, this thesis advances the state of knowledge on refactoring practices, particularly in the context of customized refactorings and floss refactoring. The tools and insights presented here offer practical solutions for developers, helping them to navigate the complexities of refactoring in real-world scenarios. As software systems continue to evolve and grow in complexity, the need for automated support in managing refactorings will only become more critical. This work provides valuable insights for future research and tool development focused on refining refactoring practices, with the goal of supporting more maintainable and reliable software systems.

Research Publications. Finally, Table 5.1 provides an overview of both the main studies that are directly linked to this thesis, as well as the secondary publications that stem indirectly from the research. These secondary works build upon the insights and findings gained throughout the course of the study.

Table 5.1: Primary and Secondary Publications Derived from this Thesis

	Title	Status
Main	**"The untold story of code refactoring customizations in practice."** 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023.	Published
	"Digging Deeper: Automated Tooling Support to Identify and Analyze Modifications in Floss Refactoring" Submitted to a Major Conference. 2024.	Submitted
	"Floss visualizer: An extension to visualize refactoring-related modifications". To be submitted to a Major Visualization Conference	To be submitted
	"The untold story of code refactoring customizations in practice: extended version". Empirical Software Engineering. 2024.	To be submitted
Secondary	**"Developers' perception matters: machine learning to detect developer-sensitive smells"**. **Oliveira, D**., Assunção, W. K., Garcia, A., Fonseca, B., & Ribeiro, M. (2022). Empirical Software Engineering, 27(7), 195.	Published
	"Applying machine learning to customized smell detection: a multi-project study". **Oliveira, D**., Assunção, W. K., Souza, L., Oizumi, W., Garcia, A., & Fonseca, B. (2020). XXXIV Brazilian Symposium on Software Engineering (pp. 233-242).	Published
	"Composite refactoring: Representations, characteristics and effects on software projects". Bibiano, A. C., Uchôa, A., Assunção, W. K., **Tenório, D.**,..., & Garcia, A. (2023). Information and Software Technology, 156, 107134.	Published
	"Look ahead! revealing complete composite refactorings and their smelliness effects". Bibiano, A. C., Assunção, W. K., ..., Soares, V., Gheyi, R., **Oliveira, D** & Oliveira, A. (2021). IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 298-308). IEEE.	Published
	"On the relation between complexity, explicitness, effectiveness of refactorings and non-functional concerns". Soares, V., Oliveira, A., Pereira, J. A., ..., Farah, P. R., **Oliveira, D**. & Uchôa, A. (2020). XXXIV Brazilian Symposium on Software Engineering (pp. 788-797).	Published
	"Recommending composite refactorings for smell removal: Heuristics and evaluation". Oizumi, W., Bibiano, A. C., Cedrim, D., Oliveira, A., Sousa, L., Garcia, A., & **Oliveira, D**. (2020). XXXIV Brazilian Symposium on Software Engineering (pp. 72-81).	Published
	"How does incomplete composite refactoring affect internal quality attributes?". Bibiano, A. C., Soares, V., Coutinho, D., Fernandes, E., Correia, J. L., ... & **Oliveira, D**. (2020). 28th International Conference on Program Comprehension (pp. 149-159).	Published
	"Characterizing and identifying composite refactorings: Concepts, heuristics and patterns". Sousa, L., Cedrim, D., Garcia, A., Oizumi, W., Bibiano, A. C., **Oliveira, D**., ... & Oliveira, A. (2020). 17th International Conference on Mining Software Repositories (pp. 186-197).	Published
	"Behind the intents: An in-depth empirical study on software refactoring in modern code review". Paixão, M., Uchôa, A., Bibiano, A. C., **Oliveira, D**., Garcia, A., Krinke, J., & Arvonio, E. (2020). 17th International Conference on Mining Software Repositories (pp. 125-136).	Published

Bibliography

[1] OPDYKE, W. F.. **Refactoring: A Program Restructuring Aid in Designing Object-OrientedApplication Frameworks**. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[2] FOWLER, M.. **Refactoring**. Addison-Wesley Professional, 1 edition, 1999.

[3] DURELLI, R. S.; SANTIBÁÑEZ, D. S.; DELAMARO, M. E. ; DE CAMARGO, V. V.. **Towards a refactoring catalogue for knowledge discovery metamodel**. In: PROCEEDINGS OF THE 2014 IEEE 15TH INTERNATIONAL CONFERENCE ON INFORMATION REUSE AND INTEGRATION (IEEE IRI 2014), p. 569–576. IEEE, 2014.

[4] BRITO, A.; HORA, A. ; TULIO VALENTE, M.. **Towards a catalog of composite refactorings**. Journal of Software: Evolution and Process, 36(4):e2530, 2024.

[5] TSANTALIS, N.; KETKAR, A. ; DIG, D.. **Refactoringminer 2.0**. IEEE Transactions on Software Engineering, 48(3):930–950, 2022.

[6] SILVA, D.; SILVA, J.; SANTOS, G. J. D. S.; TERRA, R. ; VALENTE, M. T. O.. **Refdiff 2.0: A multi-language refactoring detection tool**. IEEE Transactions on Software Engineering, 2020.

[7] SILVA, D.; TSANTALIS, N. ; VALENTE, M. T.. **Why we refactor? Confessions of GitHub contributors**. In: 24TH INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 858–870, 2016.

[8] AGNIHOTRI, M.; CHUG, A.. **Understanding refactoring tactics and their effect on software quality**. In: 2022 12TH INTERNATIONAL CONFERENCE ON CLOUD COMPUTING, DATA SCIENCE & ENGINEERING (CONFLUENCE), p. 41–46. IEEE, 2022.

[9] WEISSGERBER, P.; DIEHL, S.. **Are refactorings less error-prone than other changes?** In: PROCEEDINGS OF THE 2006 INTERNATIONAL WORKSHOP ON MINING SOFTWARE REPOSITORIES, p. 112–118, 2006.

[10] BAVOTA, G.; DE CARLUCCIO, B.; DE LUCIA, A.; DI PENTA, M.; OLIVETO, R. ; STROLLO, O.. **When does a refactoring induce bugs? an empirical study**. In: 2012 IEEE 12TH INTERNATIONAL WORKING CONFERENCE ON SOURCE CODE ANALYSIS AND MANIPULATION, p. 104–113. IEEE, 2012.

[11] RACHATASUMRIT, N.; KIM, M.. **An empirical investigation into the impact of refactoring on regression testing**. In: 2012 28TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), p. 357–366. IEEE, 2012.

[12] BIBIANO, A. C.; ASSUNCAOO, W. K. G.; COUTINHO, D.; SANTOS, K.; SOARES, V.; GHEYI, R.; GARCIA, A.; FONSECA, B.; RIBEIRO, M.; OLIVEIRA, D.; BARBOSA, C.; MARQUES, J. L. ; OLIVEIRA, A.. **Look ahead! revealing complete composite refactorings and their smelliness effects**. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 298–308, 2021.

[13] MURPHY-HILL, E.; BLACK, A. P.. **Refactoring tools: Fitness for purpose**. IEEE software, 25(5):38–44, 2008.

[14] NOEI, S.; LI, H.; GEORGIOU, S. ; ZOU, Y.. **An empirical study of refactoring rhythms and tactics in the software development process**. IEEE Transactions on Software Engineering, 49(12):5103–5119, 2023.

[15] OLIVEIRA, D. T. M.. **Towards customizing smell detection and refactorings**. Master thesis, Pontifical Catholic University of the Rio de Janeiro, 2020.

[16] TENORIO, D.; BIBIANO, A. C. ; GARCIA, A.. **On the customization of batch refactoring**. In: 3RD INTERNATIONAL WORKSHOP ON REFACTORING, p. 13–16. IEEE Press, 2019.

[17] MURPHY-HILL, E.; PARNIN, C. ; BLACK, A. P.. **How we refactor, and how we know it**. In: 31ST INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 287–297, 2009.

[18] GE, X.; SARKAR, S.; WITSCHEY, J. ; MURPHY-HILL, E.. **Refactoring-aware code review**. In: 2017 IEEE SYMPOSIUM ON VISUAL LANGUAGES AND HUMAN-CENTRIC COMPUTING (VL/HCC), p. 71–79. IEEE, 2017.

[19] KIM, M.; ZIMMERMANN, T. ; NAGAPPAN, N.. **An empirical study of refactoring challenges and benefits at Microsoft**. IEEE Transactions on Software Engineering (TSE), 40(7):633–649, 2014.

[20] MOREIRA, J. S.; ALVES, E. L. ; ANDRADE, W. L.. **An exploratory study on extract method floss-refactoring**. In: PROCEEDINGS OF THE 35TH ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, p. 1532–1539, 2020.

[21] SOUSA, L.; OIZUMI, W.; GARCIA, A.; OLIVEIRA, A.; CEDRIM, D. ; LUCENA, C.. **When are smells indicators of architectural refactoring opportunities: A study of 50 software projects**. In: PROCEEDINGS OF THE 28TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, p. 354–365, 2020.

[22] XING, Z.; STROULIA, E.. **Refactoring practice: How it is and how it should be supported-an eclipse case study**. In: ICSM'06, p. 458–468. IEEE, 2006.

[23] OLIVEIRA, J.; GHEYI, R.; MONGIOVI, M.; SOARES, G.; RIBEIRO, M. ; GARCIA, A.. **Revisiting the refactoring mechanics**. Information and Software Technology, 110:136–138, 2019.

[25] VAKILIAN, M.; CHEN, N.; NEGARA, S.; RAJKUMAR, B. A.; BAILEY, B. P. ; JOHNSON, R. E.. **Use, disuse, and misuse of automated refactorings**. In: 34TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 233–243. IEEE Press, 2012.

[26] BRITO, R.; VALENTE, M. T.. **Raid: Tool support for refactoring-aware code reviews**. In: 2021 IEEE/ACM 29TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC), p. 265–275. IEEE, 2021.

[27] OLIVEIRA, D.; ASSUNÇÃO, W. K.; GARCIA, A.; BIBIANO, A. C.; RIBEIRO, M.; GHEYI, R. ; FONSECA, B.. **The untold story of code refactoring customizations in practice**. In: 2023 IEEE/ACM 45TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 108–120. IEEE, 2023.

[28] ECLIPSE. **Eclipse ide website**, 2022. Available at: `https://www.eclipse.org`.

[29] NETBEANS. **Netbeans ide website**, 2022. Available at: `https://netbeans.org/`.

[30] JTBRAINS. **Intelij ide website**, 2022. Available at: `https://www.jetbrains.com/`.

[31] ALOMAR, E. A.; MKAOUER, M. W.; OUNI, A. ; KESSENTINI, M.. **On the impact of refactoring on the relationship between quality attributes and design metrics**. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 1–11. IEEE, 2019.

[32] CHÁVEZ, A.; FERREIRA, I.; FERNANDES, E.; CEDRIM, D. ; GARCIA, A.. **How does refactoring affect internal quality attributes? A multi-project study**. In: 31ST BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), p. 74–83, 2017.

[33] TSANTALIS, N.; MANSOURI, M.; ESHKEVARI, L. M.; MAZINANIAN, D. ; DIG, D.. **Accurate and efficient refactoring detection in commit history**. In: 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 483–494. ACM, 2018.

[34] CEDRIM, D.; SOUSA, L.; GARCIA, A. ; GHEYI, R.. **Does refactoring improve software structural quality? a longitudinal study of 25 projects**. In: 30TH BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, p. 73–82, 2016.

[35] NETTY. **Removing a seekaheadnobackarrayexception to avoid exception handling**, 2017. Available at: `https://github.com/netty/netty/commit/b03b0f22d1e`.

[36] TOMCAT, A.. **Apply patch 12 from jboynes to improve cookie handling.**, 2014. Available at: `https://github.com/apache/tomcat/commit/0cdfed561d`.

[37] SZŐKE, G.; NAGY, C.; FÜLÖP, L. J.; FERENC, R. ; GYIMÓTHY, T.. **Faultbuster: An automatic code smell refactoring toolset**. In: IEEE 15TH INTERNATIONAL WORKING CONFERENCE ON SOURCE CODE ANALYSIS AND MANIPULATION (SCAM), p. 253–258. IEEE, 2015.

[38] MURPHY-HILL, E.; PARNIN, C. ; BLACK, A.. **How we refactor, and how we know it**. IEEE Transactions on Software Engineering (TSE), 38(1):5–18, 2012.

[39] CEDRIM, D.; GARCIA, A.; MONGIOVI, M.; GHEYI, R.; SOUSA, L.; DE MELLO, R.; FONSECA, B.; RIBEIRO, M. ; CHÁVEZ, A.. **Under-**

standing the impact of refactoring on smells. In: FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 465–475, 2017.

[40] BIBIANO, A. C.; FERNANDES, E.; OLIVEIRA, D.; GARCIA, A.; KALINOWSKI, M.; FONSECA, B.; OLIVEIRA, R.; OLIVEIRA, A. ; CEDRIM, D.. **A Quantitative Study on Characteristics and Effect of Batch Refactoring on Code Smells**. In: 13TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), p. 1–11, 2019.

[41] ROBERTS, D.; BRANT, J. ; JOHNSON, R.. **A refactoring tool for smalltalk**. Theory and Practice of Object systems, 3(4):253–263, 1997.

[42] KIM, M.; GEE, M.; LOH, A. ; RACHATASUMRIT, N.. **Ref-finder: a refactoring reconstruction tool based on logic query templates**. In: 18TH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, p. 371–372, 2010.

[43] MENS, T.; TOURWÉ, T.. **A survey of software refactoring**. IEEE Transactions on software engineering, 30(2):126–139, 2004.

[44] MEANANEATRA, P.. **Identifying refactoring sequences for improving software maintainability**. In: 27TH INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), p. 406–409, 2012.

[45] TSANTALIS, N.; CHAIKALIS, T. ; CHATZIGEORGIOU, A.. **Ten years of JDeodorant: Lessons learned from the hunt for smells**. In: 25TH INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING (SANER), p. 4–14, 2018.

[46] LIN, Y.; PENG, X.; CAI, Y.; DIG, D.; ZHENG, D. ; ZHAO, W.. **Interactive and guided architectural refactoring with search-based recommendation**. In: 24TH INTERNATIONAL SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 535–546, 2016.

[47] SILVA, D.; TERRA, R. ; VALENTE, M. T.. **Recommending automated extract method refactorings**. In: 22ND INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, p. 146–156, 2014.

[48] XU, S.; SIVARAMAN, A.; KHOO, S.-C. ; XU, J.. **Gems: An extract method refactoring recommender**. In: IEEE 28TH INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING (ISSRE), p. 24–34. IEEE, 2017.

[49] PAIXÃO, M.; UCHÔA, A.; BIBIANO, A. C.; OLIVEIRA, D.; GARCIA, A.; KRINKE, J. ; ARVONIO, E.. **Behind the intents: An in-depth empirical study on software refactoring in modern code review**. In: 17TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES, p. 125–136, 2020.

[50] PANTIUCHINA, J.; ZAMPETTI, F.; SCALABRINO, S.; PIANTADOSI, V.; OLIVETO, R.; BAVOTA, G. ; PENTA, M. D.. **Why developers refactor source code: A mining-based study**. ACM Transactions on Software Engineering and Methodology (TOSEM), 29(4):1–30, 2020.

[51] BRANT, J.; STEIMANN, F.. **Refactoring tools are trustworthy enough and trust must be earned**. IEEE Software, 32(6):80–83, 2015.

[52] TOMCAT. **Re-fixed bug #49711: Httpservletrequest#getparts() does not work**, 2011. Available at: `https://github.com/apache/tomcat/commit/f69c17895`.

[53] **The untold story of code refactoring customizations in practice**, 2022. Complementary materials.

[54] BORGES, H.; VALENTE, M. T.. **What's in a GitHub star? Understanding repository starring practices in a social coding platform**. J. Syst. Softw. (JSS), 146:112–129, 2018.

[55] **Elasticsearch-hadoop**, 2022. Available at: `https://github.com/elastic/elasticsearch-hadoop`.

[56] **Hystrix**, 2022. Available at: `https://github.com/Netflix/Hystrix`.

[57] **Fresco**, 2022. Available at: `https://github.com/facebook/fresco`.

[58] **Achilles**, 2022. Available at: `https://github.com/doanduyhai/Achilles`.

[59] **Ikasan**, 2022. Available at: `https://github.com/ikasanEIP/ikasan`.

[60] **Exoplayer**, 2022. Available at: `https://github.com/google/ExoPlayer`.

[61] **Signal-android**, 2022. Available at: `https://github.com/signalapp/Signal-Android`.

[62] **Netty**, 2022. Available at: `https://github.com/netty/netty`.

[63] **Materialdrawer**, 2022. Available at: `https://github.com/mikepenz/MaterialDrawer`.

[64] **Derby**, 2022. Available at: `https://github.com/apache/derby`.

[65] **Tomcat**, 2022. Available at: `https://github.com/apache/tomcat`.

[66] **Hikaricp**, 2022. Available at: `https://github.com/brettwooldridge/HikariCP`.

[67] **Material dialogs**, 2022. Available at: `https://github.com/afollestad/material-dialogs`.

[68] TSANTALIS, N.; GUANA, V.; STROULIA, E. ; HINDLE, A.. **A multidimensional empirical study on refactoring activity**. In: 23RD ANNUAL INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE AND SOFTWARE ENGINEERING (CASCON), p. 132–146, 2013.

[69] ECLIPSE. **Using the help system**, 2022. Available at: `https://help.eclipse.org/mars/index.jsp`.

[70] BIEGEL, B.; SOETENS, Q. D.; HORNIG, W.; DIEHL, S. ; DEMEYER, S.. **Comparison of similarity metrics for refactoring detection**. In: 8TH WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, p. 53–62. ACM, 2011.

[71] ANDFLORÉAL MORANDAT ANDXAVIER BLANC ANDMATIAS MARTINEZ ANDMARTIN MONPERRUS, J. F.. **Fine-grained and accurate source code differencing**. In: ACM/IEEE INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, p. 313–324, 2014.

[72] XUAN, J.; CORNU, B.; MARTINEZ, M.; BAUDRY, B.; SEINTURIER, L. ; MONPERRUS, M.. **B-refactoring: Automatic test code refactoring to improve dynamic analysis**. Information and Software Technology, 76:65–80, 2016.

[73] LIU, K.; KIM, D.; KOYUNCU, A.; LI, L.; BISSYANDÉ, T. F. ; LE TRAON, Y.. **A closer look at real-world patches**. In: 2018 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 275–286. IEEE, 2018.

[74] FRESCO, F.. **Added dropcache to animationbackend**, 2016. Available at: `https://github.com/facebook/fresco/commit/2d82c6c185`.

[75] WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B. ; WESSLÉN, A.. **Experimentation in software engineering**. Springer Science & Business Media, 2012.

[76] TOMCAT, A.. **Implment a standard isblocking() method for output.**, 2013. Available at: `https://github.com/apache/tomcat/commit/53617a2011`.

[77] NETTY. **Move generic code to httporspdychooser to simplify implementations**, 2016. Available at: `https://github.com/netty/netty/commit/33a810a513`.

[78] NETTY. **Throw exception if keymanagerfactory is used with opensslclientcontext**, 2016. Available at: `https://github.com/netty/netty/commit/ebfb2832b2`.

[79] NETTY. **Add nonstickyeventexecutorgroup**, 2016. Available at: `https://github.com/netty/netty/commit/fc14ca31cb`.

[80] EILERTSEN, A. M.; MURPHY, G. C.. **Stepwise refactoring tools**. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 629–633. IEEE, 2021.

[81] EILERTSEN, A. M.; MURPHY, G. C.. **The usability (or not) of refactoring tools**. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING (SANER), p. 237–248. IEEE, 2021.

[82] IVERS, J.; NORD, R. L.; OZKAYA, I.; SEIFRIED, C.; TIMPERLEY, C. S. ; KESSENTINI, M.. **Industry experiences with large-scale refactoring**. In: 30TH ACM JOINT EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, p. 1544–1554, 2022.

[83] OMORI, T.; MARUYAMA, K.. **Comparative study between two approaches using edit operations and code differences to detect past refactorings**. IEICE TRANSACTIONS on Information and Systems, 101(3):644–658, 2018.

[84] FLURI, B.; WURSCH, M.; PINZGER, M. ; GALL, H.. **Change distilling: Tree differencing for fine-grained source code change extraction**. IEEE Transactions on software engineering, 33(11):725–743, 2007.

[85] TOMCAT, A.. **Fix `https://bz.apache.org/bugzilla/show_bug.cgi?id=34319`**, 2015. Available at: `https://github.com/apache/tomcat/commit/27ca28f275e457e520842e32c0d70894a89a96a4`.

[86] TOMCAT, A.. **Tomcat**, 2024.

[87] SOARES, V.; OLIVEIRA, A.; PEREIRA, J. A.; BIBANO, A. C.; GARCIA, A.; FARAH, P. R.; VERGILIO, S. R.; SCHOTS, M.; SILVA, C.; COUTINHO, D. ; OTHERS. **On the relation between complexity, explicitness, effectiveness of refactorings and non-functional concerns**. In: PROCEEDINGS OF THE XXXIV BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, p. 788–797, 2020.

[88] IVERS, J.; NORD, R. L.; OZKAYA, I.; SEIFRIED, C.; TIMPERLEY, C. S. ; KESSENTINI, M.. **Industry's cry for tools that support large-scale refactoring**. In: PROCEEDINGS OF THE 44TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING: SOFTWARE ENGINEERING IN PRACTICE, p. 163–164, 2022.

[89] OLIVEIRA, D.. **Github: Tool for identifying refactoring-related modification and data**, 2024. Available at: `https://github.com/refstudy/RefactoringRelatedDetector`.

[90] FOWLER, M.. **Refactoring: improving the design of existing code**. Addison-Wesley Professional, 2018.

[91] TOMCAT, A.. **Fix `https://bz.apache.org/bugzilla/show_bug.cgi?id=34319`**, 2015. Available at: `https://github.com/apache/tomcat/commit/b72c9ec9e39bf8a36c5a23e21181ccde9594cf70`.

[92] MIDDLETON, J.; ORE, J.-P. ; STOLEE, K. T.. **Barriers for students during code change comprehension**. In: IEEE/ACM 46TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE). ACM, 2024.

[93] TSANTALIS, N.; KETKAR, A. ; DIG, D.. **Refactoringminer 2.0**. IEEE Transactions on Software Engineering (TSE), 2020.

[94] KURBATOVA, Z.; KOVALENKO, V.; SAVU, I.; BROCKBERND, B.; ANDREESCU, D.; ANTON, M.; VENEDIKTOV, R.; TIKHOMIROVA, E. ; BRYKSIN, T.. **Refactorinsight: Enhancing ide representation of changes in git with refactorings information**. In: 2021 36TH IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), p. 1276–1280. IEEE, 2021.

[95] KALLIAMVAKOU, E.; GOUSIOS, G.; BLINCOE, K.; SINGER, L.; GERMAN, D. M. ; DAMIAN, D.. **The promises and perils of mining github**. In: PROCEEDINGS OF THE 11TH WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, p. 92–101, 2014.

[96] **Refviewer: A tool to identify and review refactoring-related modifications**, 2024. Available at: `https://github.com/RefViewTool/RefViewer`.