Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Dissertação de Mestrado

Comparing the Refactoring Mechanics of Refactoring Detection Tools and IDEs

Osmar Leandro Dantas da Silva

Campina Grande, Paraíba, Brasil

04/2022

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Comparing the Refactoring Mechanics of Refactoring Detection Tools and IDEs

Osmar Leandro Dantas da Silva

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Rohit Gheyi

(Orientador)

Campina Grande, Paraíba, Brasil

S586c Silva, Osmar Leandro Dantas da.
Comparing the refactoring mechanics of refactoring detection tools and IDEs / Osmar Leandro Dantas da Silva. – Campina Grande, 2022.
81 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2022.
"Orientação: Prof. Dr. Rohit Gheyi".
Referências.

1. Software Engineering. 2. IDEs. 3. Refactoring. 4. Detection Tools. 5. Mechanics. I. Gheyi, Rohit. II. Título.

CDU 004.41(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA SEVERINA SUELI DA SILVA OLIVEIRA CRB-15/225

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO CIENCIAS DA COMPUTACAO
Rua Aprigio Veloso, 882, - Bairro Universitario, Campina Grande/PB, CEP 58429-900

FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

OSMAR LEANDRO DANTAS DA SILVA

COMPARING THE REFACTORING MECHANICS OF REFACTORING DETECTION TOOLS AND IDES

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do título de Mestre em Ciência da Computação.

Aprovada em: 28/04/2022

Prof. Dr. ROHIT GHEYI, Orientador, UFCG

Prof. Dr. MÁRCIO DE MEDEIROS RIBEIRO, Examinador Interno, UFAL

Prof. Dr. LEOPOLDO MOTTA TEIXEIRA, Examinador Externo, UFPE

Resumo

Ferramentas de detecção de refatoração, como REFACTORINGMINER e REFDIFF, são úteis para estudar refatorações aplicadas a repositórios de software. Para avaliá-las, os autores das ferramentas estudam repositórios de software e classificam manualmente as transformações como refatorações. Entretanto, esta é uma atividade que consome bastante tempo e propensa a erros. Além disso, não está claro até que ponto a mecânica de refatoração é consistente com as implementações de refatoração disponíveis em IDEs. Neste trabalho, propomos uma técnica para testar ferramentas de detecção de refatoração. Em nossa técnica, aplicamos uma única refatoração usando um IDE popular e, em seguida, executamos a ferramenta de detecção de refatoração para verificar se ela detecta a transformação aplicada pelo IDE. Avaliamos nossa técnica executando automaticamente 9.885 transformações em quatro projetos reais de código aberto usando oito tipos de refatoração do ECLIPSE IDE. Nosso principal objetivo é verificar se existem diferenças na mecânica de refatoração de IDEs e ferramentas de detecção de refatoração e discutir essas diferenças. REFACTORINGMINER e REFDIFF detectam mais refatorações em 20,41% e 14,11% das transformações analisadas, respectivamente. Nos casos restantes, REFACTORINGMINER e REFDIFF não detectaram ou a classificam como outros tipos. Relatamos 34 relatórios de problemas para as ferramentas de detecção de refatoração. Os desenvolvedores corrigiram 16 bugs e 3 relatórios foram estavam duplicados. Em outros casos, 3 relatórios de problemas não foram aceitos.

Abstract

Refactoring detection tools, such as REFACTORINGMINER and REFDIFF, are helpful to study refactorings applied to software repositories. To evaluate them, the tools' authors study software repositories and manually classify transformations as refactorings. However, this is a time-consuming and error-prone activity. It is unclear to what extent the refactoring mechanics is consistent with refactoring implementations available in IDEs. In this work, we propose a technique to test refactoring detection tools. In our technique, we apply a single refactoring using a popular IDE, and then we run the refactoring detection tool to check whether it detects the transformation applied by the IDE. We evaluate our technique by automatically performing 9,885 transformations on four real open-source projects using eight ECLIPSE IDE refactorings. Our main goal is to see whether there are some differences in the refactoring mechanics of IDEs and refactoring detection tools and discuss the differences in the refactoring mechanics. REFACTORINGMINER and REFDIFF detect more refactorings in 20.41% and 14.11% of the analyzed transformations, respectively. In the remaining cases, REFACTORINGMINER and REFDIFF either do not detect the refactoring or classify it with other types of refactorings. We report 34 issues to refactoring detection tools, and developers fixed 16 bugs, and 3 bugs are duplicated. In other cases, 3 issues are not accepted.

Agradecimentos

Agradeço aos meus pais, Osmano e Maria de Lourdes, que mesmo a distância nunca mediram esforços para a minha educação. A minha esposa, Emília, que durante o período de mestrado mostrou incentivo irrestrito e apoio fundamental para que estes resultados fossem por mim alcançados. Agradeço ao professor Rohit Gheyi como meu orientador nesses dois anos de mestrado na UFCG. Partir de um ponto mais simples para um ponto melhor e mais complexo, que é a escrita de um trabalho científico, me fez perceber o quanto ainda preciso melhorar e aprender com meus professores. Agradeço ao Rohit por me mostrar que a ciência é um trabalho de paciência, dedicação e muita técnica. Agradeço aos meus revisores da banca, os professores Márcio Ribeiro (UFAL), Leopoldo Teixeira (UFPE) e Tiago Massoni (UFCG) por promoverem sugestões e ideias bastantes esclarecedoras que contribuíram para melhoria deste trabalho. Agradeço a UFCG, aos professores e aos funcionários do DSC/-COPIN por todo comprometimento e auxílio prestada ao longo deste período. Agradeço a Capes pelo apoio ao meu trabalho.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Refactoring [9, 20, 29] is the process of changing a program to improve its internal structure while preserving its observable behavior. For a given refactoring, we use the term mechanics to denote the informal description of how to carry out such refactoring, as some works describe [9, 20, 29]. Over the years refactoring has become a central part of software development processes, such as eXtreme Programming [1]. In addition, IDEs (ECLIPSE [7], NETBEANS [30], INTELLIJ [16]) have automated a number of refactorings, such as Rename Class, Extract Method, Inline Method and Move Method.

Some works explain the refactoring mechanics [9, 20, 29], which is a concise step-by-step description of how to carry out the refactoring. Tools that detect refactorings have been proposed in the literature. Currently, the best tools available in the state of the art are REFACTORINGMINER [49] and REFDIFF [37].

Recovering refactoring information can provide useful insights to researchers focused on understanding software evolution. Some studies address important aspects of refactorings, such as the motivations behind refactoring [24, 25, 51], improvements of detection algorithms [26, 32, 37, 49], understanding the perspective of developers [17, 24, 25, 27], and detecting behavioral changes introduced by refactorings [41, 42]. Knowing which refactoring operations were applied in the version history of a system may help in several practical tasks, such as proposing better diff visualization tools, helping during code review, and simplifying API library migration [37].

1.1 Problem

To evaluate the accuracy of refactoring detection tools, developers manually mine open-source projects to identify transformations based on their experience. Then, they create a dataset of manually classified transformations to evaluate their tools. For example, Tsantalis et al. [49] manually identified 7,226 refactorings in open-source projects for 40 different refactoring types.

However, this process is time-consuming and error-prone. Since there is no refactoring mechanics specification widely accepted by developers, developers may have different refactoring mechanics [27]. It is unclear to what extent the refactoring mechanics is consistent with refactoring implementations available in IDEs. Moreover, there is no study evaluating whether the manually classified transformations are consistent with the refactoring engines/mechanics implemented by popular IDEs. It is important to mention that developers typically use IDEs to apply refactorings [27].

1.2 Motivating Example

Fowler [9] describes the mechanics of some refactorings. For instance, Rename Method is a refactoring that changes a method name to a new one that better reveals its purpose. There are popular IDEs that implement this refactoring, such as ECLIPSE.

Consider the `RefactoringSet` class declared in a project source code. In this example, we use the source code of REFACTORINGMINER project. We can apply the Rename Method refactoring to the `getRevision` method, changing its name to `getRevisionID`. This process can be automated using the ECLIPSE IDE.

Listing 1.1: Using the ECLIPSE Rename Method refactoring.

```
@@ class RefactoringSet
-   public getRevision() {
+   public getRevisionID() {
      return revision;
    } ...
```

Listing 1.1 presents part of the actual transformation applied using ECLIPSE. We simplify the listing code to allow more readability and focus on main changes. ECLIPSE also performed similar transformations in all places where `getRevision` is used. The lines with − (red) and + (green) indicates the lines of code that are removed and added, respectively.

We can use some refactoring detection tools, such as REFACTORINGMINER and REFDIFF, to detect refactorings applied by developers. If we use them to evaluate the transformation presented in Listing 1.1, REFACTORINGMINER 2.0.3 indicates that the Rename Parameter refactoring was applied, while REFDIFF yields the Rename Method refactoring. We reported the issue #141 of Listing 1.1 and developers fixed it in REFACTORINGMINER 2.1.0.

Refactoring detection tool developers create datasets (oracles) to evaluate their tools. They manually mine open-source projects to identify refactorings based on their experience. For instance, Silva [36] uses a manual-validated dataset proposed by Tsantalis et al. [50], which contains non-refactoring and refactoring changes. However, this process is time-consuming and error-prone. It is important to propose other techniques to test refactoring detection tools.

Since developers may have different refactoring mechanics in mind when applying refactorings, and they use IDEs to apply refactorings [27], it is relevant to evaluate to what extent the refactoring mechanics of refactoring detection tools and IDEs have consistent results. Our research may help the developers' community to improve refactoring detection tools and refactoring automation tools. Likewise, it alerts the research community about the risks of not considering related biases in their work.

1.3 Solution

In this work, we propose a technique to test refactoring detection tools. In short, the technique consists of applying a refactoring using a popular IDE and then running the refactoring detection tool to check whether it detects the transformation by the IDE. For instance, we apply the Move Method refactoring to an open-source project using ECLIPSE. Then, we run REFDIFF to see whether it correctly detects the Move Method refactoring.

To evaluate the technique, we use the ECLIPSE IDE to apply 9,885 transformations of the following refactoring types: Rename Method, Rename Class, Move Method, Pull Up Method, Push Down Method, Extract Interface, Inline Method, and Extract Method. All of these refactorings are supported by both REFACTORINGMINER and REFDIFF. This allows comparing the mechanics of these tools and the ECLIPSE IDE.

REFACTORINGMINER and REFDIFF are aligned with the refactoring mechanics of ECLIPSE in 74.28% and 78.45% of the transformations, respectively. They detect more refactorings in 20.41% and 14.11% of the analyzed transformations, respectively. REFACTORINGMINER and REFDIFF do not detect refactorings or detect other types of refactorings in other cases.

We reported 34 issues to the developers of refactoring detection tools, out of which 16 were fixed. At the moment, 12 bug reports are still open, 3 bugs are duplicated, and 3 issues are not accepted, which might indicate possible problems in the mechanics of refactoring implementations. Developers fixed bugs related to the Move Method (4), Inline Method (3), Rename Method (3), Extract Method (2), Extract Interface (2), Pull Up Method (1), and Push Down Method (1) refactorings.

This technique may help the refactoring detection tool developers to improve the evaluation and the performance of their tools in combination with manually identifying transformations in software repositories.

1.4 Summary of Contributions

In summary, our main contributions are the following:

- A technique to test refactoring detection tools (Chapter 3);
- We evaluate our technique in 9,885 transformations (Section 4.1.3), and;
- We find 34 issues, out of which 16 are fixed (Section 4.1.4).

1.5 Organization

We organize this work as follows. Chapter 2 introduce some relevant concepts about refactorings and refactoring tools. Chapter 3 presents our technique, explains how it works and shows some implementation details. Section 3.1 presents an overview of our technique and its main steps. Next, Section 3.2 details the steps of our technique. Then, Section 3.3 shows some implementation details. Next, in Chapter 4 we evaluate our technique. In Section 4.1, we apply the technique to analyze the refactoring mechanics of refactoring detection tools and ECLIPSE. Section 4.2 studies the composite refactorings in refactoring detection tools. Then, Chapter 5 shows a brief history about refactoring studies and relates our study to others. Finally, Chapter 6 presents the findings of the experiments and Section 6.3 proposes some future works.

Chapter 2

Background

In this chapter we present the background of some concepts needed for understanding this work. We organize this chapter as follows. Section 2.1 explains the code refactoring. Next, Section 2.2 shows an overview about refactoring tools. Finally, we introduce some concepts about the refactoring detection tools in Section 2.3.

2.1 Code Refactoring

The life cycle of software consists of continuous changes, such as fixing bugs, introducing new features, or improving the development process and reusable parts [29]. The software must change to stay relevant. As software is adapted, it becomes more complex. Thus, the effort to carry out new adaptations also grows. Thus, making the software easier to change makes subsequent design iterations easier, and then more reusable.

Coined by Opdyke [29], define a set of program restructuring operations that support the design, evolution and reuse of object-oriented programs. Later, Fowler [9] defines code refactoring as the process of modifying a software system to improve its internal quality while preserving the observable behavior. Furthermore, the essence of code refactoring consists of several small changes that preserve the program's behavior. Mens and Tourwé [20] define some refactoring activities, such as identifying where the program should be refactored, determining which refactoring should be applied, guaranteeing that the applied refactoring preserves behavior, and then applying the refactoring.

Common refactorings have been identified, automated and incorporated into refactoring tools, such as VISUAL STUDIO CODE [21], INTELLIJ [16], ECLIPSE [7], and NETBEANS [30], and detection tools, such as REFACTORINGMINER [49], REFDIFF [37], REF-FINDER [32], and REFACTORINGCRAWLER [6]. Furthermore, some commercial development methodology have refactoring as the major activity of their workflows, such as eXtreme Programming [1].

Empirical studies have investigated the refactoring benefits and its developers' perception. Silva et al. [39] investigated the reasons that drive developers to refactor their code. They asked developers of 748 Java projects why they performed the identified refactorings. Their results indicate that fixing a bug or feature additions mainly drives refactorings. Kim et al. [18] conduct a field study of refactoring benefits and challenges at Microsoft through complementary methods: a survey, semi-structured interviews with professional software engineers, and quantitative analysis of version history data. The survey participants reported the benefits they have observed from refactoring and cite readability and maintainability. The quantitative analysis finds the main refactored modules had a higher reduction in the number of inter-module dependencies.

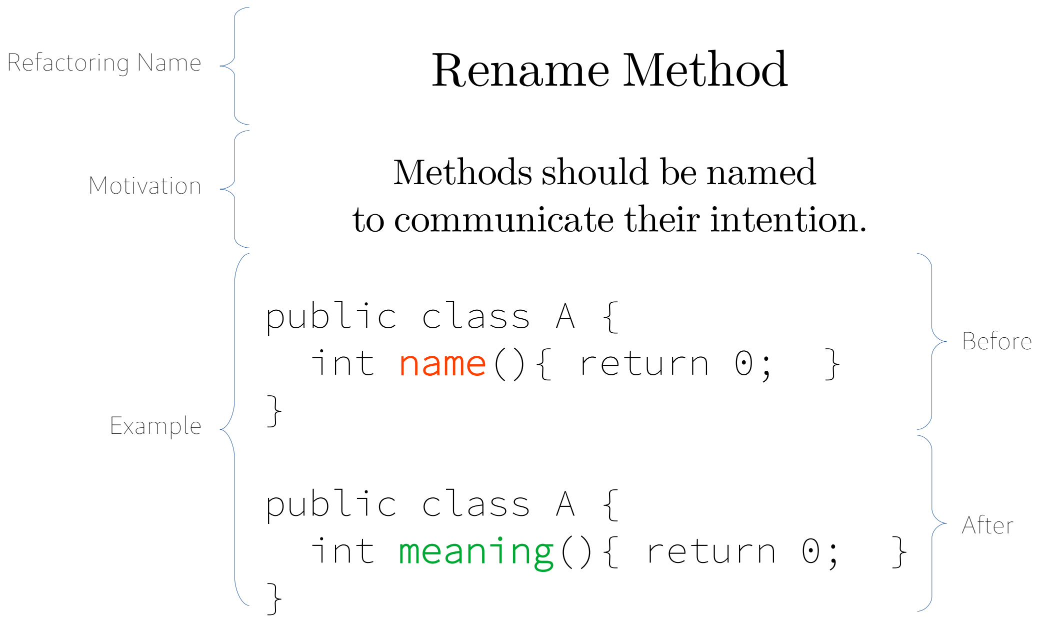
2.2 Refactoring Implementations

Refactorings can occurs in high-level, or major design changes, and low-level, when a scope is minimum such as renaming a variable. The high-level refactorings can be implemented in terms of several low-level refactorings [29].

The availability of catalogs allows widely adopting the refactoring practices and describing both high and low-level refactorings. These catalogs give names and describe common operations to apply the refactorings, named refactoring mechanics. For example, Rename Class is the name of refactoring, and its mechanics consists of changing the class name and updating its references.

Fowler [9] published a catalog that had a high impact on the world of software development. The catalog describes the refactorings through specifications of 72 refactoring types. The standard format for each specification has five parts: a name, a summary, a motivation, the mechanics, and examples. Figure 2.1 shows a brief of refactoring specification from the author's specification. According to Fowler, the name is important to build a refactoring vocabulary. Later, Fowler [10] improved his catalog by adding, renaming, and removing refactorings. For example, the Add Parameter, the Remove Parameter, and the Rename Method refactorings are part of the Change Function Declaration refactoring.

Figure 2.1: A brief illustration of refactoring specification from Fowler's [9] catalog.



```
public class A {
  int name(){ return 0;  }
}

public class A {
  int meaning(){ return 0;  }
}
```

Preconditions are an essential part of refactoring specification. When satisfying its preconditions, the program preserves the behavior. For example, if we run the program before and after refactoring with the same input, the program yields the same output. Some refactorings are simple to implement, but for a few refactorings, their preconditions are undecidable. Fortunately, it can be determined whether these refactorings can be applied safely [29].

2.2.1 Code Examples

To read the following examples, ponder the refactoring specifications of Fowler's [9] catalog. The listings are reduced for didactic purposes and simple reading. Consider the Java programming language to read the listings and discard some implementation details that are not necessary to understand the refactoring mechanics, in other words, the way to apply the refactoring. Furthermore, consider the red lines as the deleted lines of code and green lines as the new lines of code.

The Extract Method refactoring is a popular refactoring, and several instances can be found on public code repositories, such as GitHub[1], GitLab[2], and Bitbucket.[3] This refactoring can be applied when the method is too long or its purpose is hard to understand. Accordingly to Fowler [9], the mechanics of the Extract Method refactoring consists of selecting statements from a long method with a purpose in common and creating a new one, as seen in Listing 2.1. The new method (lines 10 to 14) has a distinct purpose from the original method (lines 1 to 8).

Listing 2.1: Using the ECLIPSE Extract Method refactoring.

```
 1    public void show(Step step) {
 2      ...
 3  -   float value = step.getValue();
 4  -   boolean isCelsius = step.getMetric();
 5  -   float temp = isCelsius ? (1.8*value + 32) : (value-32)*1.8;
 6  +   float temp = convert(step);
 7      ...
 8    }
 9
10  + private float convert(Step step) {
11  +   float value = step.getValue();
12  +   boolean isCelsius = step.getMetric();
13  +   float temp = isCelsius ? (1.8*value + 32) : (value-32)*1.8;
14  +   return temp;
15  + }
```

As another example, the Rename Method refactoring is also popular in public code repositories. It can be applied when the method's name does not reflect its purpose. For example, suppose we manually rename the method named as `getNumber` to `getEven` in Listing 2.2. According to Fowler's [9] mechanics, the Rename Method refactoring aims to replace the old name (line 4) with the new name (line 5). Moreover, the mechanics updates the references from the old method to the new method, as seen in lines 1 and 2.

[1]https://github.com
[2]https://gitlab.com
[3]https://bitbucket.org

Listing 2.2: Using the ECLIPSE Rename Method refactoring.

```
-   String number = getNumber(obj);
+   String number = getEven(obj);
    ...

- private int getNumber(int number) {
+ private int getEven(int number) {
    return number%2 > 0 ? number++ : number;
  }
```

As another example, consider the Inline Method refactoring can be applied to remove a method. Fowler's [9] mechanic of the Inline Method refactoring aims to put the method's body into the body of its callers and then delete the method's declaration. The Inline Method refactoring has several motivations, such as when a group of methods can be composed or when delegations can be simplified. Consider we apply the Extract Method refactoring to the `getName` method in Listing 2.3. First, we copy the body of the `getName` method to the `show` method (lines 3 and 4). Then, we remove references to the `getName` method (line 2) and finally we delete its declaration (lines 8 to 12).

Listing 2.3: Using the ECLIPSE Inline Method refactoring.

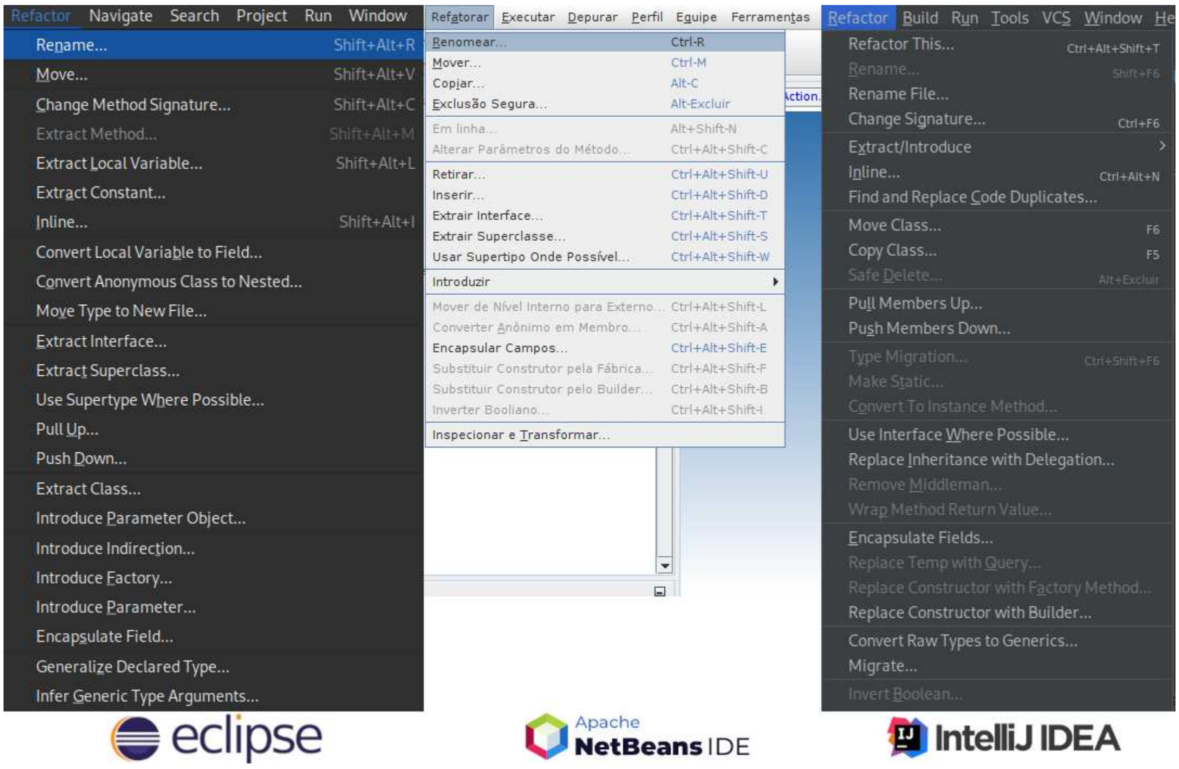
```
 1   public void show(Object obj) {
 2 -   String number = getName(obj);
 3 +   String number = obj.getValue();
 4 +   number += 10;
 5     ...
 6   }
 7
 8 - private String getName(Object obj) {
 9 -   String number = obj.getValue();
10 -   number += 10;
11 -   return number;
12 - }
```

2.2.2 Automating Refactorings

In his dissertation, Roberts [33] examines techniques for using runtime analysis to assist refactoring and identifies the criteria that are necessary for any refactoring tool to succeed. Furthermore, he describes several ways to make a refactoring tool that is both fast and reliable. His approach, the REFACTORING BROWSER, automated several refactorings for the Smalltalk object-oriented programming language.

Since then, researchers have improved the correctness and applicability of refactorings by using formal techniques through refactoring tools, such as JastAdd Refactoring Tools (JRRT). Schafer and Moor [35] proposed specifications to improve refactoring implementations of Eclipse in terms of correctness. They implemented some Java refactorings in JRRT. Moreover, a number of modern development tools have automated refactorings to programming languages, such as VISUAL STUDIO CODE [21], INTELLIJ [16], ECLIPSE [7], and NETBEANS [30]. Figure 2.2 shows side-by-side the refactor menu of three IDEs: ECLIPSE, NETBEANS, and INTELLIJ.

Figure 2.2: Refactor menu of ECLIPSE 4.14, NETBEANS 8.2, and INTELLIJ 11.


Refactor Navigate Search Project Run Window
Rename... Shift+Alt+R
Move... Shift+Alt+V
Change Method Signature... Shift+Alt+C
Extract Method... Shift+Alt+M
Extract Local Variable... Shift+Alt+L
Extract Constant...
Inline... Shift+Alt+I
Convert Local Variable to Field...
Convert Anonymous Class to Nested...
Move Type to New File...
Extract Interface...
Extract Superclass...
Use Supertype Where Possible...
Pull Up...
Push Down...
Extract Class...
Introduce Parameter Object...
Introduce Indirection...
Introduce Factory...
Introduce Parameter...
Encapsulate Field...
Generalize Declared Type...
Infer Generic Type Arguments...
eclipse
Refatorar Executar Depurar Perfil Equipe Ferramentas
Renomear... Ctrl-R
Mover... Ctrl-M
Copiar... Alt-C
Exclusão Segura... Alt-Excluir
Em linha... Alt+Shift-N
Alterar Parâmetros do Método... Ctrl+Alt+Shift-C
Retirar... Ctrl+Alt+Shift-U
Inserir... Ctrl+Alt+Shift-D
Extrair Interface... Ctrl+Alt+Shift-T
Extrair Superclasse... Ctrl+Alt+Shift-S
Usar Supertipo Onde Possível... Ctrl+Alt+Shift-W
Introduzir
Mover de Nível Interno para Externo... Ctrl+Alt+Shift-L
Converter Anônimo em Membro... Ctrl+Alt+Shift-A
Encapsular Campos... Ctrl+Alt+Shift-E
Substituir Construtor pela Fábrica... Ctrl+Alt+Shift-F
Substituir Construtor pelo Builder... Ctrl+Alt+Shift-B
Inverter Booliano... Ctrl+Alt+Shift-I
Inspecionar e Transformar...
Apache NetBeans IDE
Refactor Build Run Tools VCS Window He
Refactor This... Ctrl+Alt+Shift+T
Rename... Shift+F6
Rename File...
Change Signature... Ctrl+F6
Extract/Introduce
Inline... Ctrl+Alt+N
Find and Replace Code Duplicates...
Move Class... F6
Copy Class... F5
Safe Delete... Alt+Excluir
Pull Members Up...
Push Members Down...
Type Migration... Ctrl+Shift+F6
Make Static...
Convert To Instance Method...
Use Interface Where Possible...
Replace Inheritance with Delegation...
Remove Middleman...
Wrap Method Return Value...
Encapsulate Fields...
Replace Temp with Query...
Replace Constructor with Factory Method...
Replace Constructor with Builder...
Convert Raw Types to Generics...
Migrate...
Invert Boolean...
IntelliJ IDEA

We can notice some additional refactorings names, such as Replace Constructor with Builder in INTELLIJ and Introduce Factory in ECLIPSE. Moreover, INTELLIJ and VISUAL STUDIO CODE can enable refactoring functionalities on demand, according to the user's selection. Oliveira et al. [27] surveyed 107 developers and find the developers use IDEs to apply refactorings, and 71.02% expect different programs as output. Besides, about 7% of them consider the meaning of the refactoring names, and it can impact their communication.

2.3 Refactoring Detection Tools

Some works detail the refactoring mechanics [9, 20, 29], which is a concise step-by-step description of how to carry out the refactoring. In addition, IDEs e.g. INTELLIJ, ECLIPSE, and NETBEANS have automated several refactorings. Tools that detect refactorings have been proposed in the literature. Currently, the best tools available in the state of the art are REFACTORINGMINER [49] and REFDIFF [37].

Recovering refactoring information can provide valuable insights to researchers who focus on understanding software evolution. Some studies address relevant aspects of refactorings, such as the motivations behind refactoring [24, 25, 51], improvements in detection algorithms [26, 32, 37, 49], understanding of the developers' perspective [17, 24, 25, 27], and detection of behavioral changes introduced by refactorings [41, 42]. Knowing the performed refactorings in the version history may help in several practical tasks, such as proposing better diff visualization tools, helping during code review, and simplifying API migration [37].

REFACTORINGMINER and REFDIFF are refactoring detection tools that may help researchers empirically investigate several aspects of refactorings, including their benefits. They create an oracle based on refactorings applied to real open-source projects and hosted on GitHub and identified by professional developers to evaluate their tools.

To evaluate the accuracy of refactoring detection tools, developers manually mine open-source projects to identify transformations based on their experience. Then, they create a dataset of manually classified transformations to evaluate their tools. For example, Tsantalis et al. [49] manually identified 7,226 refactorings in open-source projects for 40 different refactoring types. However, this process is time-consuming and error-prone. Since there is no refactoring mechanics specification widely accepted by developers, developers may have different refactoring mechanics [27].

The lack of universally accepted definitions of refactoring mechanics is one of the main challenges for refactoring detection tools. Oliveira et al. [27] show the developers and IDE refactoring tools use different mechanics for most refactoring types. Additionally, it is challenging to account for all Java constructs that may affect refactoring mechanics, such as generics and lambda expressions.

In the following sections, we show the improvement of the mainstream refactoring detection tools. The tools can detect several refactoring types in different programming levels and run through programming languages such as JavaScript and C/C++.

2.3.1 REFACTORINGMINER

REFACTORINGMINER [49] is a refactoring detection tool used as a library and API. The project has been written and specializes in the Java programming language. Moreover, it can automatically detect refactorings applied in the version history of a Java project.

The first version of the tool is called RMINER 1.0 [50]. Its approach relies on an Abstract Syntax Tree (AST) statement matching algorithm that determines refactoring candidates without requiring a well-defined calibration process. The authors present two novel techniques to deal with relationships of refactoring candidates: the concept of abstraction, which deals with AST type changes in statements due to refactoring, and the concept of argumentization, which deals with changes in subexpressions due to parameterization. From these techniques, RMINER 1.0 can match AST statements with detection rules to support 15 refactoring types.

The next version of the tool was renamed REFACTORINGMINER 2.0 [49] and had relevant increments of the detection algorithms. The improvements support the detection of low-level and submethod-level refactorings. In other words, it means the REFACTORINGMINER can detect refactorings in a level of variables, such as Rename Variable and Extract Variable refactorings. Thereby, the authors expand the supported refactorings to 40 types. Some types are present in Fowler's catalog [11], such as Extract Class, automated by IDEs, such as Move Class, and manually applied by developers, such as Change Variable Type. In addition, the authors improved the precision and recall by supporting new heuristics and replacement types to match statements and consequently refactoring candidates.

In the recent version available in the public repository,[4] REFACTORINGMINER 2.2 enhances the fine-grained detections, for example, the Add Parameter Modifier, the Remove Variable Modifier, and the Replace Attribute with Variable refactorings. Besides, it improves the language-specific statements, such as anonymous class and lambdas. Thereby, it supports a total of 85 types of refactorings. Table 2.1 shows the evolution of supported refactoring types along with tool versions.

Table 2.1: Evolution of refactoring types supported by REFACTORINGMINER.

RMINER 1.0	**REFACTORINGMINER 2.0**	**REFACTORINGMINER 2.1**	**REFACTORINGMINER 2.2**
	Move and Rename Class		
	Extract Class	Add Method Annotation	Change Attribute Access Modifier
	Extract Subclass	Remove Method Annotation	Encapsulate Attribute
	Extract Variable	Modify Method Annotation	Parameterize Attribute
	Inline Variable	Add Attribute Annotation	Replace Attribute with Variable
Extract Method	Parameterize Variable	Remove Attribute Annotation	Add Method Modifier
Inline Method	Rename Variable	Modify Attribute Annotation	Remove Method Modifier
Rename Method	Rename Parameter	Add Class Annotation	Add Attribute Modifier
Move Method	Rename Attribute	Remove Class Annotation	Remove Attribute Modifier
Move Attribute	Move and Rename Attribute	Modify Class Annotation	Add Variable Modifier
Pull Up Method	Replace Variable with Attribute	Add Parameter Annotation	Add Parameter Modifier
Pull Up Attribute	Replace Attribute (with Attribute)	Remove Parameter Annotation	Remove Variable Modifier
Push Down Method	Merge Variable	Modify Parameter Annotation	Remove Parameter Modifier
Push Down Attribute	Merge Parameter	Add Variable Annotation	Change Class Access Modifier
Extract Superclass	Merge Attribute	Remove Variable Annotation	Add Class Modifier
Extract Interface	Split Variable	Modify Variable Annotation	Remove Class Modifier
Move Class	Split Parameter	Add Parameter	Move Package
Rename Class	Split Attribute	Remove Parameter	Split Package
Extract and Move Method	Change Variable Type	Reorder Parameter	Merge Package
Rename Package	Change Parameter Type	Add Thrown Exception Type	Localize Parameter
	Change Return Type	Remove Thrown Exception Type	Change Type Declaration Kind
	Change Attribute Type	Change Thrown Exception Type	Collapse Hierarchy
	Extract Attribute	Change Method Access Modifier	Replace Loop with Pipeline
	Move and Rename Method		Replace Anonymous with Lambda
	Move and Inline Method		

REFACTORINGMINER can be imported as a library into Java projects or used as a command-line tool by generating a bundle from the source code. Since version 2.0, REFACTORINGMINER is available on Maven projects by adding as a dependency. The tool works with remote or local Git-based repositories and can detect refactorings between two versions

[4]https://github.com/tsantalis/RefactoringMiner

of source code, such as commits or tags, at a specific version or the entire project. Next, we show some detection examples by running the library of REFACTORINGMINER 2.2 in the REFACTORING-TOY-EXAMPLE[5] project. We publish the following examples in a Maven project on Github.[6]

Listing 2.4 presents the `detectBetweenCommits` method, which accepts four arguments: first, a repository object, the start commit (line 1), the end commit (line 2), and a callback. Then, the tool yields a list of detected refactorings for each commit present between these commits (line 7). In this example, we have a unique commit between them, and the tool finds a single instance of Rename Method refactoring.

Listing 2.4: Detection at between commits of REFACTORING-TOY-EXAMPLE project.

```
1 String s = "9921a4c"; // start commit
2 String e = "124ce1c"; // end commit
3 miner.detectBetweenCommits(repo, s, e, new RefactoringHandler() {
4   @Override
5   public void handle(String c, List<Refactoring> refs) {
6     for (Refactoring r : refs)
7       System.out.println(r.toString());
8   }
9 });
```

As another example, REFACTORINGMINER allows detecting refactorings at a specific commit. Listing 2.5 shows the `detectAtCommit` method accepts three arguments: an object, which references the Git repository, a commit hash (line 1), which contains refactorings, and a callback, which is called after processing. In this case, the tool verifies the modified statements between the commit and its parent from the history. Then, the tool yields a unique instance of the Extract Method refactoring (line 6).

The next example shows the detection of all refactoring in the Git project. Listing 2.6 shows the `detectAll` method accepts three arguments: an object that references the repository, the branch name (line 1), and a callback. REFACTORINGMINER starts from the most recent commit and scans its successive parents from the same branch. Then, it yields a list of

[5]https://github.com/osmarleandro/refactoring-toy-example
[6]https://github.com/osmarleandro/mining-refactorings-example

detected refactorings for each commit (line 5). The results can take a long time of processing, depending on repository history size.

Listing 2.5: Detection at single commit of REFACTORING-TOY-EXAMPLE project.

```java
String commit = "124ce1c";
miner.detectAtCommit(repo, commit, new RefactoringHandler() {
  @Override
  public void handle(String c, List<Refactoring> refs) {
    for (Refactoring r : refs)
      System.out.println(r.toString());
  }
});
```

Listing 2.6: Detection at all commits in repository.

```java
String branch = "master";
miner.detectAll(repo, branch, new RefactoringHandler() {
  @Override
  public void handle(String c, List<Refactoring> refs) {
    for (Refactoring r : refs)
      System.out.println(r.toString());
  }
});
```

2.3.2 RefDiff

RefDiff [40] is a detection tool to find relationships between code elements in the commit history of git repositories. The relationships are refactoring types and denote is the elements are the same or there is a refactoring operation. The tool has a language-agnostic design, which allows supporting programming languages by the implementation of plugins.

The first versions of RefDiff employ a combination of heuristics based on static analysis and code similarity to detect 13 well-known refactoring types. The tool's approach uses the classical TF-IDF similarity measure from information retrieval to compute code similarity.

It relies on similarity thresholds to find relationships between the entities and needs to be calibrated by applying a well-defined calibration process.

For example, the RefDiff 1.0 calibration process used ten random sets of commits that contain refactorings from a public dataset. Then, the authors run the tool using different thresholds values, ranging from 0.1 to 0.9 by 0.1 increments. Later, the output of RefDiff was compared to the known refactorings from the dataset to verify precision and recall. The goal is to optimize the precision and recall by adjusting thresholds.

The strategy of RefDiff 2.0 [37] is to design a new core that supports language-independent implementations. Beyond Java, the authors created plugins to support the popular general-purpose programming languages JavaScript and C/C++. Besides, some relationships have no support in the second version, for example, the Move Field, the Pull Up Field, and the Push Down Field refactorings. Table 2.2 shows the refactoring types and relationships supported by each tool version.

Table 2.2: Evolution of refactoring types supported by RefDiff.

RefDiff 1.0	**RefDiff 2.0**
Java	Java, JavaScript, C/C++
Rename Type, Rename Method Move Type, Move Method, Move and Rename Type, Extract Supertype Change Method Signature, Pull Up Method, Push Down Method Extract Method, Inline Method	Rename, Move, Move and Rename, Extract Supertype Change Signature Pull Up, Push Down, Extract, Inline
-	Extract and Move
Pull Up Field, Push Down Field, Move Field	-
-	Convert Type

The RefDiff's approach has two main steps: first, it performs a Source Code Analysis, and then a Relationship Analysis. In the Source Code Analysis, the authors build tree models that represent the source code, called Code Source Tree (CST). They parse the source code and generate an Abstract Syntax Tree (AST) that is used to build the CST for a specific language. Each language plugin defines the code elements to be extracted.

For example, suppose we rename the `getNumber` method to `getEven` as seen in Listing 2.2. To build the CST nodes from Java REFDIFF extracts methods, classes, enums, and interfaces. In this example, it extracts the methods from the AST of before and after renaming versions.

In the next step, Relationship Analysis, the tool mine the refactorings relationships from CSTs. REFDIFF finds the set of relationships R between the CST sets. Each relationship $r \in R$ is a combination of $v1 \in V_1$, $v2 \in V_2$, $t = RelationshipType$. REFDIFF's authors define several relationship types and conditions to find them, such as Pull Up, Rename, Move, Inline, and Extract. For example, consider the same example of Listing 2.2. The CSTs contains the following relationship: $v_1 = getNumber$, $v_2 = getEvent$, $t = Rename$.

REFDIFF's authors evaluated the precision and recall by a dataset of 3,248 real refactoring instances from Java repositories on GitHub. They achieved an overall precision and recall of 96% and 80%, respectively. To evaluate JavaScript and C/C++ languages, they selected the 20 most popular GitHub projects of each programming language. Then, they randomly selected ten instances of each refactoring type and manually count the true and false positives. Following this process, they calculate an overall precision of 91% and 88% and recall of 88% and 91% for JavaScript and C/C++, respectively.

Next, we present the using of the REFDIFF library. We create a Maven project with its dependency and run the Java program. This source-code is available in the online repository.[7] The following example of Listing 2.7 shows the detection of refactoring from a specific commit at the REFACTORING-TOY-EXAMPLE project. Listing 2.7 shows the `computeDiffForCommit` method from the REFDIFF's API. It accepts two arguments: a repository object, which references the Git repository (lines 2 and 6), and the commit hash (line 3).

REFDIFF internally computes the differences between the commit and its parent (line 7), selects candidates from modified code, and yields an object that contains the relationships between refactoring candidates (line 9). Finally, our program prints a description of each refactoring relationship (line 10). Listing 2.7 prints the Extract Method refactoring.

[7] https://github.com/osmarleandro/mining-refactorings-example

Listing 2.7: REFDIFF detection at specific commit.

```
public static void main(String[] args) {
  String url = ".../refactoring-toy-example.git",
  String commit = "0d3a06c";

  RefDiff refDiffJava = new RefDiff(new JavaPlugin(...));
  File repo = refDiffJava.cloneGitRepository(... url);
  CstDiff diff = refDiffJava.computeDiffForCommit(repo, commit));

  for (Relationship rel : diff.getRefactoringRelationships())
    System.out.println(rel.getStandardDescription());
}
```

2.3.3 Refactoring Names

The names of refactorings can differ in each automation tool, such as IDEs, as well in detection tools, such as REFACTORINGMINER and REFDIFF. For example, REFDIFF's Change Signature Method refactoring refers to low-level refactorings applied by REFACTORINGMINER, such as the Change Parameter Type refactoring. Table 2.3 presents a comparison between the names of some refactorings in both tools.

Table 2.3: Matching the refactoring names in REFACTORINGMINER to relationship types applied to code elements in REFDIFF.

REFACTORINGMINER 2.2	REFDIFF 2.0
Rename Method, Rename Class	Rename Method, Class
-	Convert Type
Pull Up Method, Push Down Method	Pull Up Method, Push Down Method
Move Method, Move Class	Move Method, Class
Extract and Move Method	Extract and Move Method
Inline Method	Inline Method
Extract Method, Extract Superclass, Extract Interface	Extract Method, Interface, Superclass
Change Parameter Type	Change Signature Method
Move and Rename Method, Move and Rename Class	Move and Rename Class

Chapter 3

Technique to Compare Refactoring Mechanics

In this chapter, we propose a technique to compare refactoring detection tools. Section 3.1 presents an overview of our technique and its main steps. Next, Section 3.2 details the steps of our technique. Then, Section 3.3 explains how we implement our technique.

3.1 Overview

Figure 3.1 shows the main steps of our technique. It receives as input a program and a particular refactoring type, hereafter called `X`, to be evaluated. First, we search for all possible locations where `X` can be applied in the input program. Then, for each location, we apply a *single* refactoring to the input program using an implementation of `X`, producing as output a new version of the input program. Notice that this step yields a set of output programs containing the application of a single refactoring for each possible location identified in Step 1. Finally, we run a refactoring detection tool (`Y`) to check whether `Y` detects the application of refactoring `X` in each pair (Input and Output Programs). The technique then produces a report indicating whether the refactoring detection tool was able to detect each refactoring.

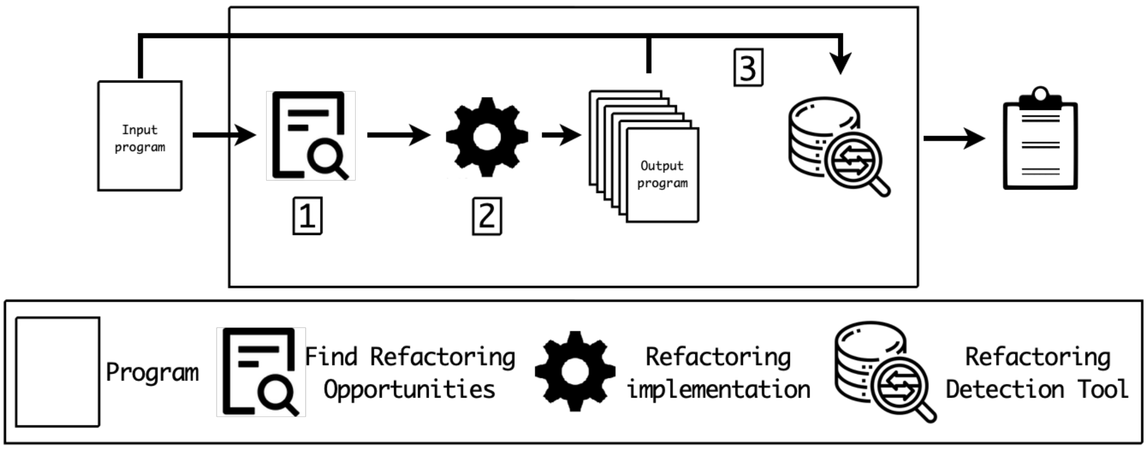

Input program
1
2
Output program
3
Program
Find Refactoring Opportunities
Refactoring implementation
Refactoring Detection Tool

Figure 3.1: A technique to compare refactoring mechanics of refactoring detection tools and refactoring implementation.

3.2 Steps

Step 1 consists of identifying all *locations* where we can apply a refactoring (`X`). In this step, we search for all possible refactoring targets in the input source code. We followed a similar approach proposed by Gligoric et al. [13].

For example, suppose we would like to evaluate the Rename Class refactoring. Our technique finds all classes (locations) in the program received as input. As another example, consider we would like to evaluate the Rename Method refactoring. Our technique searches for all methods (locations) declared in the input program.

The result of Step 1 is a set of locations (`L`). This way, locations are dependent on the particular refactoring type. For instance, it might be a method for the Inline Method refactoring, statements for the Extract Method refactoring, or a method declaration for the Move Method refactoring.

In Step 2, we apply the refactoring implementation of `X` to all possible locations `L`. For example, suppose we would like to apply the Move Method refactoring. In Step 1, we find all methods in the input program. In Step 2, we apply the Move Method refactoring to each location, in this case a method, such as the `doHealthCheck` method presented in Listing 3.1. Then, we move it from the `ElasticsearchRestHealthIndicator` class to the `Health` class. In real scenarios, the user must choose a target class for moving. For this example, consider the target class that was selected previously.

Listing 3.1: Using the ECLIPSE Move Method refactoring.

```
@@ class ElasticsearchRestHealthIndicator
-   doHealthCheck(builder, ...);
+   builder.doHealthCheck(this, ...);

- private void doHealthCheck(Health.Builder b, String json) { ...
-   builder.withDetails(response);
- }

@@ class Health
+ public void doHealthCheck(ElasticsearchRestHealthIndicator e,
+   String json) { ...
+   withDetails(response);
+ }
```

For simplicity, we use a default parameter for other options available in the refactoring implementation. Listing 3.1 presents one pair of input and output program. We repeat this process to all locations identified in Step 1. The result of this step is a set of pairs (`P`) containing the input program and the output program. It is important to mention that each pair consists of an output program yielded by a single application of `X` to the input program. We only apply a single refactoring using the IDE to make it simpler to compare with the refactoring detection tool output. It is also easier for us to report an issue to developers. If the refactoring implementation does not apply a transformation, or the resulting code does not compile, we ignore it.

In Step 3, we run the refactoring detection tool for each pair in `P`. For example, consider Listing 3.1 in which we apply the Move Method refactoring to the `getAnnotation` method using the ECLIPSE implementation. REFACTORINGMINER yields a single instance of the Move Method refactoring. For this pair, we then report that the refactoring mechanics of REFACTORINGMINER **is consistent with** the refactoring implementation of ECLIPSE.

Now, consider the example presented in Listing 3.2. Suppose we would like to apply the Rename Class refactoring to the `UMLJavadoc` class using the ECLIPSE implementation. For this pair, REFACTORINGMINER yields seven refactorings: Rename Class (1), Change

Return Type (2), Change Attribute Type (2), and Change Parameter Type (2). For this pair, since the output of REFACTORINGMINER contains the refactoring applied by ECLIPSE and other refactorings, we report that the refactoring mechanics of REFACTORINGMINER **is partially consistent with** the refactoring implementation of ECLIPSE.

Listing 3.2: Using the ECLIPSE Rename Class refactoring.

```
@@ class UMLJavadoc
- public class UMLJavadoc {
+ public class UMLDocJava {

@@ class UMLClass
- public void setJavadoc(UMLJavadoc javadoc) {
+ public void setJavadoc(UMLDocJava javadoc) {
```

As another example, suppose we would like to apply the Rename Method refactoring using ECLIPSE to rename the `getRevision` method to `getRevisionID`. Part of this refactoring in presented in Listing 1.1. Running REFACTORINGMINER on this pair yields the Rename Parameter refactoring. For this pair, since the output of REFACTORINGMINER is not the refactoring applied by ECLIPSE and the output is not empty, we report that the refactoring mechanics of REFACTORINGMINER **is different** from the refactoring applied by ECLIPSE.

In some cases, REFACTORINGMINER yields an empty output. Suppose we would like to apply the Extract Method refactoring illustrated in Listing 3.3. REFACTORINGMINER 2.0.3 yields an empty set of refactorings, and does not detect the refactoring applied by ECLIPSE. The issue #159 of Listing 3.3 was fixed, and REFACTORINGMINER 2.1.0 correctly detects the applied refactoring.

Listing 3.3: Using the ECLIPSE Extract Method refactoring.

```
@@ class PrometheusPushGatewayManager
catch (UnknownHostException ex) {
- String host = ex.getMessage();
- String message = ...
+ String message = extracted(ex);

```

```
@@ class PrometheusPushGatewayManager
+ private String extracted(UnknownHostException ex) {
+   String host = ex.getMessage();
+   String message = ...
+   return message;
+ }
```

Our technique classifies each pair into one of the four categories previously mentioned with the examples, and reports the results to the user. We summarize our categorization in Table 3.1. Our main goal is to see whether there are differences in the refactoring mechanics of IDEs and refactoring detection tools. When such a difference is found, it does not necessarily mean that the issue is on the refactoring detection tool side. Moreover, when a refactoring detection tool reports more information, it does not always mean that they are wrong. Sometimes, the refactoring mechanics implemented by the IDE may add some additional optional changes, such as introducing temporary variables when inlining a method. As future work, we aim to study the mechanics implemented by IDEs and see how they can be modified to be consistent with refactoring detection tools.

Table 3.1: Classification of transformations: x = refactoring type applied by the refactoring implementation A; Y = the list of refactorings detected by the refactoring detection tool B.

Category	**Definition**
A and B are different	$x \notin Y \land Y \neq \emptyset$
A is consistent with B	$x \in Y \land \#Y=1$
A is partially consistent with B	$x \in Y \land \#Y>1$
B yields an empty set	$Y = \emptyset$

Automating the technique significantly increases the number of differences. Furthermore, we have two main advantages of our technique: first, we can raise discussions about the aspects that make the mechanics of IDEs and detection tools have differences, and second, it allows us to find bugs in both mechanics. In short, Step 1 provides a set of locations. Locations are methods, classes or statements read from the Abstract Syntax Tree (AST). The detection of refactoring opportunities consists of iterating on each location and verify-ing preconditions to apply the refactoring. After Step 2, and before Step 3, our technique filters uncompilable refactorings. Then, we store the input and output versions. Thus, we can isolate the resulting code between refactorings, and we compare them more easily.

3.3 Tool Support

In Step 1, we use ECLIPSE's AST to perform code analysis and identify the possible locations where we might apply refactorings: classes, methods, interfaces, and so on. We use the refactoring implementations from ECLIPSE [7] in Step 2. So far, we have tool support for the following refactoring implementations: Rename Method, Rename Class, Move Method, Push Down Method, Pull Up Method, Extract Interface, Inline Method, and Extract Method. Finally, in Step 3 we consider two refactoring detection tools currently, namely REFACTORINGMINER [49] and REFDIFF [37].

Our technique works from any IDE, and we select the popular IDE ECLIPSE [7] to perform transformations. Moreover, we select the state-of-art refactoring detection tools REFACTORINGMINER and REFDIFF to mine refactorings from the source code transformed by ECLIPSE. Moreover, we use other tools to build and run tests and manage code versions.

Gradle [15] is an open-source build automation system that is based on the concepts of Apache Ant[1] and Apache Maven.[2] The initial plugins are primarily focused around Java development and deployment, Groovy and Scala. We use Gradle to check compilation errors and run testing tasks when available.

Git [12] is a distributed version control system initially designed and developed by Linus Torvalds for Linux kernel development but has been adopted by many other software projects. In this work, we use some concepts of Git to arrange the massive refactoring instances, such as branches and commits. A commit object, or only commit, is a checkpoint of the actual file state and is identified by a hash. A branch is simply a lightweight movable pointer to one commit.

Once we select a GitHub project, Step 2 processes all possible code elements to apply refactorings, such as classes, interfaces, and methods. The ECLIPSE IDE evaluates precon-

[1]https://ant.apache.org/
[2]https://maven.apache.org/

ditions to apply the refactoring. If some precondition fails, our technique implementation selects the next code element. For each code element, we apply a unique refactoring type and perform all steps of the technique. For example, suppose we want to apply Rename Class to all classes in a Java project called SPRING-BOOT. First, the technique implementation reads a class, performs the Rename Class refactoring, and checks if the project still compiles. If the project does not compile, we discard all modified files and select the next class.

Each Eclipse's refactoring implementation has specific parameters and mechanics. We studied these refactoring implementations and defined the default parameters according to Table 4.2. Suppose we would like to apply the Rename Method refactoring. The first step is to select a method to rename. Next, we give a new name to the method as a parameter. Then, the IDE evaluates some preconditions and replaces the old method name with a new method name in all locations that call the old method. In this case, the user needs to give one parameter: the new method name.

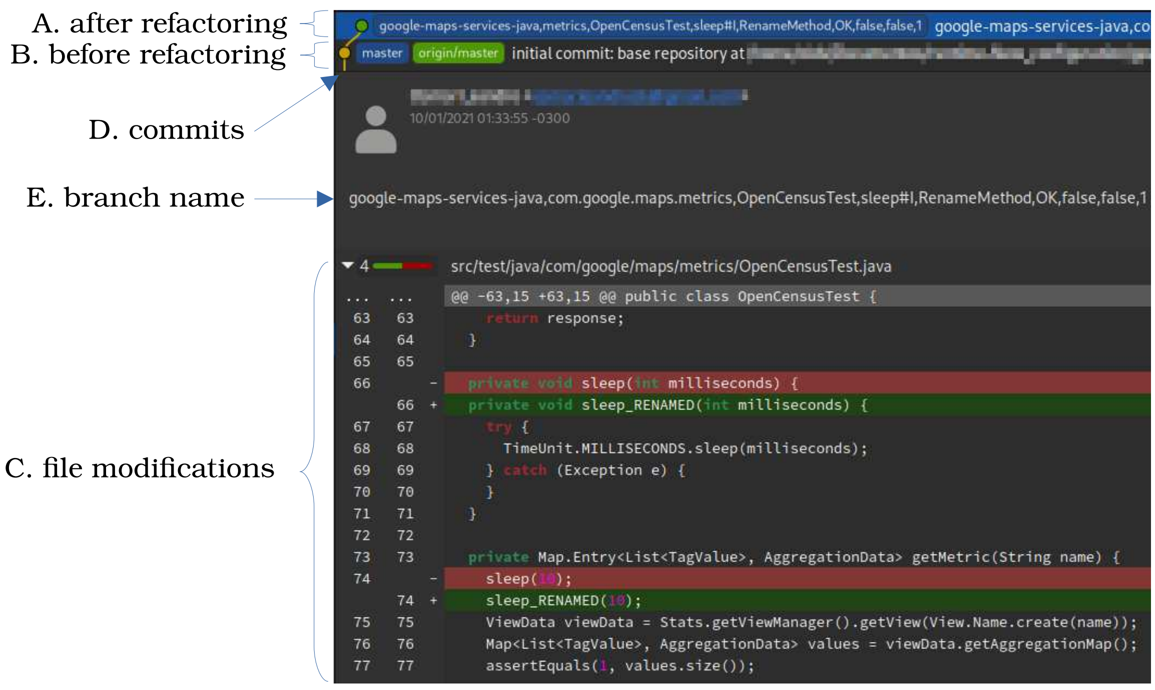
After applying the refactoring in Step 2, we use the Gradle tool to build and run testing tasks when available. If the output code does not pass in some build or testing task, we use the Git tool to discard all files modifications and return to the original code. Otherwise, we save the resulting code as a new refactoring instance.

For example, Figure 3.2 shows the GITG[3], an open-source graphical user interface for Git. In this example, we apply the Rename Method refactoring to the `sleep` method of the GOOGLE-MAPS-SERVICE-JAVA project. Notice the file modifications (C) have the resulting code (green) and original code (red). The green code is stored in a new commit (B) and a new branch (E) points to it. Thus, this commit can be compared with its parent (A) to show file modifications (C) after applying the Rename Method refactoring.

Once a new Git commit has the refactored code, we run the refactoring detection tools REFACTORINGMINER and REFDIFF. The refactoring detection tools read the refactored code, present in the commit, and yield a list of detected refactorings or an empty list. Then, we verify whether their results contain the applied refactoring, according to categories of Table 3.1. These categories help to analyze the differences between the mechanics of the refactoring detection tools and IDEs.

[3]`https://wiki.gnome.org/Apps/Gitg`

Figure 3.2: A diff tool shows the differences between two versions of code: deleted lines (red) and added lines (green). The original code (B) is the parent of refactored code (A), and the tool shows the differences between them (C). Each new commit (D) is based on the original code and stored in a new branch (E).



```
google-maps-services-java,metrics,OpenCensusTest,sleep#I,RenameMethod,OK,false,false,1  google-maps-services-java,co
master  origin/master  initial commit: base repository at [redacted]

[redacted]
10/01/2021 01:33:55 -0300

google-maps-services-java,com.google.maps.metrics,OpenCensusTest,sleep#I,RenameMethod,OK,false,false,1

4  src/test/java/com/google/maps/metrics/OpenCensusTest.java
@@ -63,15 +63,15 @@ public class OpenCensusTest {
     return response;
   }

-  private void sleep(int milliseconds) {
+  private void sleep_RENAMED(int milliseconds) {
     try {
       TimeUnit.MILLISECONDS.sleep(milliseconds);
     } catch (Exception e) {
     }
   }

   private Map.Entry<List<TagValue>, AggregationData> getMetric(String name) {
-    sleep(10);
+    sleep_RENAMED(10);
     ViewData viewData = Stats.getViewManager().getView(View.Name.create(name));
     Map<List<TagValue>, AggregationData> values = viewData.getAggregationMap();
     assertEquals(1, values.size());
```

For example, Table 3.2 shows a technique report produced after Step 3. The first column (*Project*) shows the project name, and the second column (*Method*) shows the code element to apply the refactoring, in this case, a method. Then, the third column (*Refactoring*) shows the refactoring implementation of ECLIPSE. The fourth and fifth columns (*RefactoringMiner*, *Count*) show the results of REFACTORINGMINER and its count, respectively. Finally, the last two columns (*RefDiff*, *Count*) show the results of REFDIFF and its count, respectively.

The *Count* columns represent each category. Consider the definition of Table 3.1 of a refactoring implementation `A` and a list of refactoring types detected `B`. When `A` is consistent with `B`, $count = 1$. When `A` is partially consistent with `B`, $count > 1$. When `A` and `B` are different, $count < 0$. When `B` is empty, $count = 0$.

We manually analyze a number of refactoring instances from the technique's report to find the categories between the refactoring detection tools and ECLIPSE IDE. For example,

Table 3.2: An example of the technique's report [19]. We simplify it by changing the project names and removing some columns, such as the packages and classes columns.

Project	Method	Refactoring	REFACTORINGMINER	Count	REFDIFF	Count
A	renamedMethodA#	Rename Method	Rename Method, Change Variable Type, Change Variable Type	3	Rename Method	1
B	extractedMethodB#	Extract Method	Extract Method	1	NA	0
C	movedMethodC#	Move Method	NA	0	Inline Method, Inline Method, Inline Method	-3

consider the negative value of Table 3.2. We intend to understand the output of the refactoring detection tool by analyzing the resulting code. So, we use a diff tool to compare the code before and after refactoring, as seen in Figure 3.2. We discuss our findings in Section 4.1.4.

Chapter 4

Comparing Refactoring Mechanics

We compare the refactoring mechanics of refactoring detection tools and ECLIPSE by conducting studies from different perspectives. First, we apply our technique to detect differences between the mechanics (Section 4.1). Then, we verify whether composite refactorings impact the results of refactoring detection tools (Section 4.2). We organize this chapter as follows. Section 4.1 presents the evaluation of the technique and discusses the results. Next, Section 4.2 shows the evaluation of the composite refactorings.

4.1 Refactoring Detection Tools and IDE

In this study, we use our technique to evaluate eight refactoring implementations of ECLIPSE using two refactoring detection tools in four open-source projects. We run two instances of the technique for each detection tool, first using ECLIPSE and REFACTORINGMINER, and next using ECLIPSE and REFDIFF. Section 4.1.1 presents our study definition. Section 4.1.2 describes the experimental setup. Sections 4.1.3 and 4.1.4 present results and discuss them, respectively. Section 4.1.5 describes some threats to validity, and Section 4.1.6 answers our research questions.

4.1.1 Study Definition

Our goal is to apply our technique to check the consistency of the refactoring mechanics used by refactoring detection tools and refactoring implementations of ECLIPSE. We analyze

REFACTORINGMINER, REFDIFF, and the refactoring implementations of ECLIPSE. We address the following research questions:

RQ$_1$ To what extent the refactorings applied by ECLIPSE are detected by REFACTORINGMINER or REFDIFF?

We count the number of refactorings detected by REFACTORINGMINER (RQ$_{1.1}$) or REFDIFF (RQ$_{1.2}$) that are consistent with ECLIPSE, as well as the number of refactorings that are partially consistent and different. Finally, we also count the number of times that the detection tool yields an empty set.

RQ$_2$ How many bugs can our technique detect in REFACTORINGMINER and REFDIFF?

We submit issues to the developers of refactoring detection tools and count the number of accepted and fixed bugs in each tool.

4.1.2 Experimental Setup

We ran the experiment on a laptop computer with Core i7 3.1 GHz and 8 GB RAM running Fedora 33 and Oracle JDK 1.8. Table 4.1 shows the four open-source projects used as inputs: APACHE GOBBLIN, GOOGLE MAPS SERVICES JAVA, REFACTORINGMINER, and SPRING BOOT. We deliberately select Gradle[1] projects to allow integration with build automation. In addition, we verify compilation of the stable branch and prefer projects with at least 30 KLOC. The selected projects allow to generate 9,885 transformations.

Table 4.1: Projects used in our evaluation.

Project	**Domain**	**KLOC**	**Stars**	**Contributors**
APACHE GOBBLIN	A distributed data integration framework	454	1.8K	78
GOOGLE MAPS SERVICES JAVA	A Java client for Google Maps Services	38	1.4K	90
SPRING BOOT	A framework to create Spring-based applications	674	9.6K	320
REFACTORINGMINER	A refactoring detection tool	127	173	12

[1]https://gradle.org/

We use eight refactoring implementations of ECLIPSE JDT 4.16 in Step 2. ECLIPSE JDT is a widely used IDE and has several refactoring implementations. In this work, we consider the following refactoring implementations: Rename Method, Rename Class, Move Method, Push Down Method, Pull Up Method, Extract Interface, Inline Method, and Extract Method refactorings. We select refactoring types that both refactoring detection tools support in common. It allows comparing the mechanics of refactoring detection tools and IDE. In addition, the total of transformations is superior to previous oracles [37, 49, 50] and was enough to provide pertinent discussions.

In Step 3, we use two refactoring detection tools: REFACTORINGMINER 2.0.3[2] (to answer $RQ_{1.1}$), and REFDIFF 2.0[3] (to answer $RQ_{1.2}$). Although there are other refactoring detection tools, such as REF-FINDER [32] and REFACTORINGCRAWLER [6], we are restricted to the most recent detection tools, which have the higher precision and recall and are compatible with the Java language. To answer RQ_2, we verify several transformations to see whether it is possible to arrive at the refactored version of the code by applying the detected refactorings. Then, we create an issue to discuss that behavior with the refactoring detection tool developers. Finally, the experimental data are available online [19].

Table 4.2: ECLIPSE modified parameters used in the experiments.

Refactoring	**Modified Parameters**
Extract Interface	Members to declare in the interface: select all; Interface name: class name with "I" prefix.
Extract Method	Selection: see Algorithm 1.
Inline Method	Inline a random method.
Move Method	New target for the method: the first type from the list.
Pull Up Method	Specify actions for members: check a random method.
Push Down Method	Specify actions for members: check a random method.
Rename Class	New name: old class name with a suffix.
Rename Method	New name: old method name with a suffix.

Table 4.2 presents the modified parameters for each refactoring implementation. For example, when applying the Rename Method refactoring using the graphical user interface

[2]`https://github.com/tsantalis/RefactoringMiner/commit/fee2968`

[3]`https://github.com/aserg-ufmg/RefDiff/commit/2a06cfd`

of ECLIPSE, the user must provide a new method name. We define this parameter as the method's name with a suffix. As another example, applying the Move Method refactoring, the user provides a target class to move. We define this user parameter as the first element from a target class list provided by the IDE. In addition, Table 4.3 shows the parameters that we did not set in this experiment, and they come as default in ECLIPSE JDT 4.16.

Table 4.3: ECLIPSE default parameters for each refactoring implementation.

Refactoring	**Default Parameters**
Extract Interface	Use the extracted interface type where possible: checked;

Extract Method	Method name: extracted;
	Access modifiers: private;
	Declare thrown runtime exceptions: unchecked;
	Generate method comment: unchecked;

Inline Method	Delete method declaration: checked.
Move Method	New method name: unchanged; Keep original method as delegate to moved method: unchecked;
Pull Up Method	Select destination type: unchanged; Use the destination type where possible: checked; Use the destination type in 'instanceof' expressions: unchecked;
Push Down Method	-
Rename Class	Update references: checked; Update similarly named variables and methods: unchecked; Update textual occurrences in comments and strings: unchecked; Update fully qualified names in non-Java text files: unchecked.
Rename Method	Update references: checked; Keep original method as delegate to renamed method: unchecked.

The refactoring implementation of IDEs also contain several preconditions that must be satisfied before applying a refactoring. For instance, when applying the Rename Method refactoring, ECLIPSE verifies some preconditions to avoid naming conflicts. It also checks a valid range of statements or expressions when applying the Extract Method refactoring. Furthermore, it verifies whether the class has a superclass before applying the Pull Up Method refactoring. Similarly, it checks whether a class has a subclass before applying the Push Down Method refactoring.

Algorithm 1 describes how we apply the Extract Method refactoring. It applies the Extract Method refactoring to methods containing at least three statements. First, it tries to extract the second statement. If it cannot apply a refactoring, it tries to extract the second and third statements. We repeat this process by adding more statements until we successfully apply a refactoring using the IDE, or we reach the last statement.

Algorithm 1 Applying the Extract Method refactoring.

```
1: function $applyExtractMethod(method, tool)$
2:     $stmts \leftarrow method.getStatements()$
3:     if $stmts.size() < 3$ then
4:         return $false$    ▷ It is not possible to apply a refactoring.
5:     end if
6:     $stmts = stmts - first(stmts) - last(stmts)$    ▷ Remove first and last statements.
7:     $extract \leftarrow \emptyset$
8:     for $stmt \in stmts$ do    ▷ Iterate all statements in order of declaration
9:         $extract = extract.concat(stmt)$    ▷ Add $stmt$ in the end of extract
10:        if $tool.applyExtractMethod(extract)$ then    ▷ Can extract the statements?
11:            return $true$    ▷ It applies the refactoring, and concludes the process.
12:        end if
13:    end for
14:    return $false$    ▷ It is not possible to apply a refactoring.
15: end function
```

4.1.3 Results

Our technique analyzed a total of 9,885 transformations applied by ECLIPSE using eight refactoring types implemented by ECLIPSE, which were evaluated using REFACTORING-MINER and REFDIFF. Table 4.4 summarizes our results.

We apply 2,740 transformations using the Rename Method refactoring implementation. REFACTORINGMINER and REFDIFF are aligned in 2,696 (98.39%) and 2,380 (86.86%) transformations with ECLIPSE's refactoring mechanics, respectively. In five of the transformations, REFACTORINGMINER detects more refactorings. For instance, it detects the Change Variable Type and the Rename Parameter refactorings. In seven transformations,

Table 4.4: Summary of the results of our technique. The second from the last column indicates the total number of refactorings applied by ECLIPSE JDT 4.16. The last column indicates the total number of reported issues. RM = REFACTORINGMINER; RD = REFDIFF.

Refactoring	Tool	Consistent	Partially Consistent	Different	Empty	Total	Issues
Rename Method	RM	2,696	5	0	39	**2,740**	3
	RD	2,380	0	7	353		3
Rename Class	RM	399	234	7	3	**643**	1
	RD	523	118	2	0		2
Move Method	RM	720	669	15	154	**1,558**	4
	RD	1,432	0	17	109		4
Push Down Method	RM	10	130	0	5	**145**	2
	RD	10	134	1	0		2
Pull Up Method	RM	23	0	0	2	**25**	1
	RD	25	0	0	0		0
Extract Interface	RM	1,000	373	1	2	**1,376**	2
	RD	740	636	0	0		1
Inline Method	RM	263	396	13	229	**901**	4
	RD	503	346	0	52		1
Extract Method	RM	2,212	205	1	52	**2,470**	3
	RD	2,121	157	0	192		1

REFDIFF detects other refactoring types. For example, REFDIFF detects a combination of the Extract and Move Method refactorings instead of the Rename Method refactoring. Finally, REFACTORINGMINER and REFDIFF do not detect any refactoring in 39 (1.42%) and 353 (12.88%) transformations, respectively.

We apply 643 transformations using the Rename Class refactoring implementation of ECLIPSE. REFACTORINGMINER and REFDIFF are aligned in 399 (62.05%) and 523 (81.34%) transformations with ECLIPSE's refactoring mechanics, respectively. In 234 (36.39%) and 118 (18.35%) transformations, REFACTORINGMINER and REFDIFF detect more refactorings. For instance, REFACTORINGMINER detects the Change Attribute Type, the Change Parameter Type, the Change Return Type, and the Change Variable Type refactorings. REFACTORINGMINER did not detect any refactoring in seven transformations, while REFDIFF failed to detect refactorings in two transformations. Finally, REFACTORINGMINER detects other refactorings in three transformations. For example, it reports the Change Attribute Type, the Change Parameter Type, the Change Return Type, and the Change Variable Type refactorings instead of the Rename Class refactoring.

We apply 1,558 transformations using the Move Method refactoring implementation. REFACTORINGMINER and REFDIFF are aligned in 720 (46.21%) and 1,432 (91.91%) transformations with ECLIPSE's refactoring mechanics, respectively. In 669 (42.94%) transformations, REFACTORINGMINER yields more refactorings. For example, it yields up to 11 different refactoring types, such as the Add Parameter, the Change Parameter Type, the Inline Method and the Pull Up Method refactorings. REFACTORINGMINER and REFDIFF detect other refactorings in 15 and 17 transformations, respectively. Finally, REFACTORINGMINER and REFDIFF do not detect any refactoring in 154 (9.88%) and 109 (7%) transformations, respectively. For example, REFDIFF yields the Change Signature Method, the Extract Method, the Inline Method, the Pull Up Method refactorings instead of the Move Method refactoring.

Using the Push Down Method refactoring implementation, we apply 145 transformations. Both tools are aligned with the ECLIPSE refactoring mechanics in 10 transformations. In 130 (89.66%) and 134 (92.41%) transformations, REFACTORINGMINER and REFDIFF identify more refactorings. For example, they consider an instance of the Push Down Method refactoring for each subclass we push down instead of a single transformation. If we push down a method to five subclasses using ECLIPSE, the refactoring detection tools yield five instances of the Push Down Method refactoring. In 5 transformations, REFACTORINGMINER does not detect any refactoring. REFDIFF detects the Move Method refactoring when we apply Push Down Method to a method that has a parameterized type replaced by a concrete type. We discuss this behavior in issue #16.

We use the Pull Up Method refactoring implementation to apply 25 transformations. REFACTORINGMINER and REFDIFF are aligned in 23 (92%) and 25 (100%) transformations with ECLIPSE's refactoring mechanics, respectively. In two (8%) transformations, REFACTORINGMINER does not yield any transformation.

Using the Extract Interface refactoring implementation, we apply 1,376 transformations.

REFACTORINGMINER and REFDIFF are aligned with ECLIPSE's refactoring mechanics in 1,000 (72.67%) and 740 (53.78%) transformations, respectively. REFACTORINGMINER and REFDIFF detect more refactorings in 373 (27.11%) and 636 (46.22%) transformations, respectively. For instance, REFACTORINGMINER detects the Change Variable Type and the Change Parameter Type refactorings. In addition, REFACTORINGMINER does not report any refactoring only in a single transformation. For example, it yields the Change Return Type refactoring, but it does not detect the Extract Interface refactoring. Finally, REFACTORINGMINER detects other refactorings in two transformations.

We apply 901 transformations using the Inline Method refactoring implementation. REFACTORINGMINER and REFDIFF are aligned in 263 (29.19%) and 503 (55.83%) transformations with ECLIPSE's refactoring mechanics, respectively. REFACTORINGMINER and REFDIFF detect more refactorings in 396 (43.95%) and 346 (38.40%) transformations, respectively. For instance, REFACTORINGMINER detects the Extract Variable and the Rename Parameter refactorings. Besides, REFACTORINGMINER and REFDIFF do not detect any refactoring in 229 (25.42%) and 52 (5.77%) transformations, respectively. Finally, REFACTORINGMINER detects other refactorings in 13 transformations. For example, it yields the Change Variable Type and Extract Variable refactorings instead of the Inline Method refactoring.

Finally, we apply 2,470 transformations using the Extract Method refactoring implementation. REFACTORINGMINER and REFDIFF are aligned in 2,212 (89.55%) and 2,121 (85.87%) transformations with ECLIPSE's refactoring mechanics, respectively. REFACTORINGMINER and REFDIFF detect more refactorings in 205 (8.30%) and 157 (6.36%), respectively. For instance, REFACTORINGMINER detects the Parameterize Variable, Rename Parameter and Rename Variable refactorings. Moreover, REFACTORINGMINER and REFDIFF do not yield any transformation in 52 (2.11%) and 192 (7.77%) transformations, respectively. Finally, REFACTORINGMINER detects another refactoring in a single transformation only. For this case, it yields the Change Parameter Type instead of the Extract Method refactoring.

4.1.4 Discussion

We organize this section as follows. Next, we present our process to inspect the results and report the issues. Then, we discuss the cases when the refactoring detection tools yield an empty set of refactorings. Following, we discuss the transformations in which the refactoring detection tools do not detect the applied refactoring. Finally, we discuss the cases that the refactoring detection tools detect the applied transformation and more refactoring types.

Bug Reports

After the last step of the technique (Figure 3.1), we manually classify failures into distinct issues, and then we report to developers. We analyze all transformations in the Partially Consistent, Different, and Empty categories (Table 3.1). We analyze the results of each refactoring detection tool separately and discuss the main issues in the following sections.

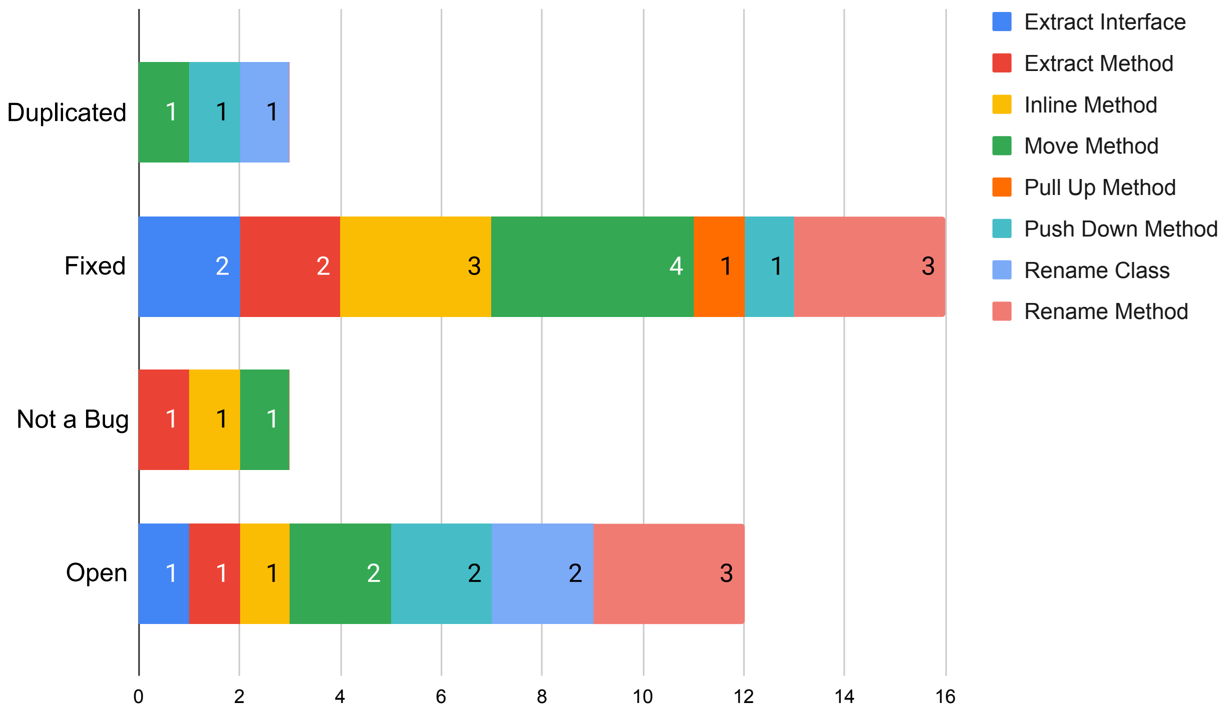
For the Different and Empty categories, we deliberately select one transformation to manually analyze. For the transformations in the Partially Consistent category, we cluster the outputs based on the types of refactorings yielded by each refactoring detection tool. Then, we select one instance of each cluster to manually analyze. To make it simpler to explain to refactoring detection tool developers, we modify the program by removing the parts that are unrelated to the bug, inspired by delta debugging [31,53]. Next, we manually analyze each candidate and discard the ones that are not bugs. For example, if a tool reports refactoring `A` and it is not possible to transform, or partially transform, the code before into the code after applying `A`, we consider the candidate as an issue. For the remaining ones, we report each pair of small input and output programs to refactoring detection tool developers stating that we expected the application of a single refactoring type.

By following this process, we find 34 issues. Figure 4.1 presents a summary of the submitted issues per refactoring type. Developers fixed bugs in all refactoring types, except for the Rename Class refactoring. In this case, we classify it as Duplicate, because developers fixed the issue before submission. In addition, three issues are rejected (Not a Bug). Until the writing of this work, the open issues have no answer.

During our analysis, we notice refactoring detection tools have similar output in transformations of the same refactoring type. Then, we cluster these transformations as the same issue. We analyze the results of each refactoring detection tool separately and discuss the main issues in next sections. Our technique helps to find 34 issues[4] related to comparing the consistency with ECLIPSE.

[4] `https://github.com/osmarleandro/refs/blob/master/issues.csv`

Figure 4.1: Summary of issues by the status and refactoring type.

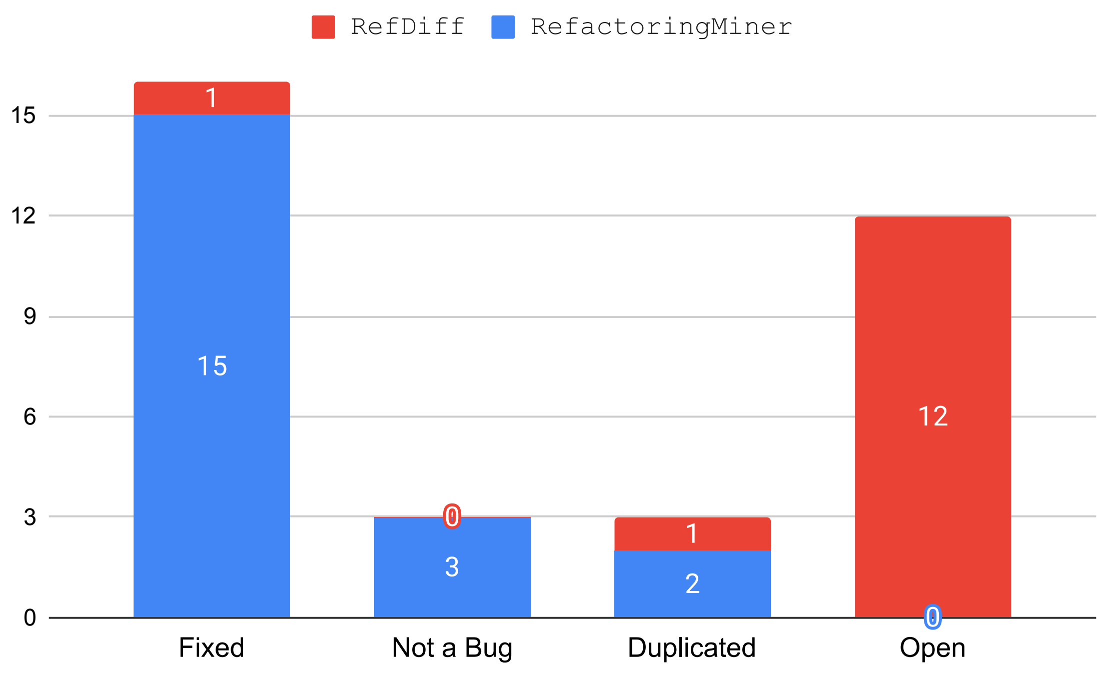


From the refactoring detection tools' perspective, we present the bug reports by status. Figure 4.2 shows the quantitative summary of issues per refactoring detection tool. Developers fixed 16 bugs, where REFACTORINGMINER fixed 15 bugs and REFDIFF fixed 1 bug. Moreover, 3 issues are duplicated, 3 issues are not bugs, and 12 issues are still open. We run the technique in REFACTORINGMINER 2.1.0[5] and REFDIFF 2.0[6] after bugs fixing, and our technique does not detect new issues.

As mentioned before, we aim to see whether there are differences between the refactoring mechanics of IDEs and refactoring detection tools. Therefore, when inconsistencies arise, it does not mean that the issue is on the refactoring detection tool side. Furthermore, when a refactoring detection tool reports multiple potential results, and we classify it as Partially Consistent, it does not always mean that they are wrong. Sometimes the refactoring mechanics implemented by the IDE may add some additional optional changes. For example, we apply the Inline Method refactoring[7] to argumentIntersectionSize method. REFACTOR-

[5] https://github.com/tsantalis/RefactoringMiner/commit/149468e
[6] https://github.com/aserg-ufmg/RefDiff/commit/3dabc79
[7] https://github.com/osmarleandro/RefactoringMiner/commit/58213a7

Figure 4.2: Summary of issues per refactoring detection tool. We submit a total of 14 and 20 issues to RefDiff and RefactoringMiner, respectively.



ingMiner detects the Inline Method and Rename Variable refactoring. We consider the Rename Variable refactoring a true instance, but Eclipse could avoid it.

To determine the causes for all bugs [19], we need to study the source code of each tool and the proposed correction by the author, which can also be mixed with other bug fixes. Understanding the root cause of a bug is not an easy task, given that we are not the tool developers. In what follows, we discuss our results in light of some of the comments we received while submitting bugs to the developers [19] in RefactoringMiner and RefDiff.

First example, we analyze the source code of Listing 4.3. Its cause may be related to the support to Generic Types in the Push Down Method refactoring added by RefactoringMiner's developers. When moving a method that returns Generic Types, it returns the actual type corresponding to the type in each subclass.

A similar issue of that in Listing 4.3 was reported to RefDiff's developers. It detects the Move Method refactoring when we apply the Push Down Method refactoring. Developers explain that the tool enforces the same signature when searching Push Down Method candidates. Moreover, they do not deal with situations when generic types are replaced by concrete types. REFDIFF may have classified it as the Move Method refactoring because this type allows changes to the signature.

As another example, Listing 4.2 may be caused because fan-in relationships (methods which call the refactored method) are ignored in the replacement function implemented by the matching algorithm in REFACTORINGMINER.

On the other hand, in Listing 4.4 REFDIFF detects the Extract and Move Method refactoring, but we apply the Rename Method refactoring. Some code fragments are updated to a new method name. The renamed method is a single-line method, and this can increase the similarity score when comparing with updated code fragments.

Concerning the Extract Method refactoring in Listing 4.12, some of the additional refactoring types reported by REFACTORINGMINER are due to the IDE refactoring mechanics, and REFACTORINGMINER reports correct changes. For example, the Parameterize Variable refactoring is reported when local variables declared in the original method become parameters of the extracted method. As another example, REFACTORINGMINER reports the Rename Parameter refactoring when a parameter of the original method is passed with a different name in the extracted method.

We tried to contact REFDIFF developers a few times, but we did not receive an answer for 12 out of 14 issues reported. So, we compare the output of REFACTORINGMINER to the same code from the issues of REFDIFF. In 6 out of 12 unanswered issues of REFDIFF, REFACTORINGMINER is consistent with ECLIPSE. In 3 out of 12 unanswered issues, REFDIFF and ECLIPSE are different, while REFACTORINGMINER is partially consistent with ECLIPSE.

Finally, we base our conclusions on accepted and fixed bugs in both tools, largely from REFACTORINGMINER. However, we understand the interpretation of mechanics can be different, and we report some issues to both tools: issues #20, #21, and #24 (REFDIFF). From these, we address discussions that were made in both tools, such as repeated transformations in issue #122 (REFACTORINGMINER) and issue #24 (REFDIFF), about the Push Down Method in issue #153 (REFACTORINGMINER) and issue #21 (REFDIFF) and the Partially Consistent category in issue #20 (REFDIFF).

Refactoring Detection Tools yield an empty set.

In some cases, the refactoring detection tools do not detect the applied refactoring using ECLIPSE. For example, Listing 4.1 shows the application of the Move Method refactoring to the `doHealthCheck` method (Step 1). ECLIPSE moves the method to the `Health` class, changes its signature, and removes the original `doHealthCheck` method. REFACTORINGMINER does not detect this transformation.

The Move Method refactoring mechanics described by Fowler [9] allows changing the signature of the method. In our study, we identify 8 issues related to this type of refactoring. This example may help developers discuss the correct refactoring mechanics for the Move Method refactoring. One may argue that correct refactoring mechanics is to enclose operations that change the signature of the moved method, such as Add parameter, Remove Parameter, Change Parameter Type involving the Source or Target class types. We reported this problem on issue #133 and developers fixed it.

Listing 4.1: Using the ECLIPSE Move Method refactoring.

```
@@ class ElasticsearchRestHealthIndicator
-   doHealthCheck(builder, ...));
+   builder.doHealthCheck(this, ...));

@@ class ElasticsearchRestHealthIndicator
- private void doHealthCheck(
-   Health.Builder b, ...) {

@@ class Health
+ public void doHealthCheck(
+   ElasticsearchRestHealthIndicator e, ...) {
```

As another example, Listing 4.2 shows the result of ECLIPSE applying the Rename Method refactoring to the `setAttribute` method. REFACTORINGMINER does not detect this refactoring. We analyzed the resulting code and identified that the transformation applied by ECLIPSE happens in a method containing a single line of code. The developers fixed issue #140 of Listing 4.2 in REFACTORINGMINER 2.1.0. Furthermore, they mention that single-line methods are tricky to detect. In general, it is harder to match statements near to similar single-line methods.

REFACTORINGMINER [49] uses two pre-processing techniques called *abstraction* and *argumentization* to deal with changes taking place in code statements whem applying the Extract, Inline and Move Method refactorings. It matches two versions of the same method if they have an identical signature, that is same name, parameters, return type, parent class, and body. Their algorithm matches the added and deleted code elements to find code elements with signature changes, but similar methods in same class can confuse the detection of Listing 4.2.

Listing 4.2: Using the ECLIPSE Rename Method refactoring.

```
@@ class UMLModelASTReader
-  variableDeclaration.setAttribute(true);
+  variableDeclaration.setAttr(true);

@@ class VariableDeclaration
- public void setAttribute(boolean isAttribute) {
+ public void setAttr(boolean isAttribute) {
    this.isAttribute = isAttribute;
  }
```

In Listing 4.3, ECLIPSE applies the Push Down Method refactoring to the `awaitIgnoreError` method that returns a generic type. The actual return type is replaced in each subclass. REFACTORINGMINER 2.0.3 does not yield any refactoring. We submitted the issue #137 to REFACTORINGMINER 2.1.0, and it is fixed. REFDIFF developers replied to the submitted issue #16 and explained that this behavior is a limitation of their implementation. Some language constructs, such as generic types and lambdas, are challenging for refactoring detection tools in Java programs. Our technique can help to improve them by showing some examples that may expose new rules to be considered in refactoring detection tools.

Listing 4.3: Using the ECLIPSE Push Down Method refactoring.

```
@@ abstract class PendingResultBase
- @Override
- public final T awaitIgnoreError() { ... }
```

```

@@ class DistanceMatrixApiRequest
+ @Override
+ public final DistanceMatrix awaitIgnoreError() { ... }
```

Mechanics Are Different

In Listing 4.4, ECLIPSE applies the Rename Method refactoring to the `isConstructor` method. REFDIFF yields an Extract and Move Method refactoring for each method call, in this case. This result shows that REFDIFF does not yield the applied refactoring, but several others that were not applied. We report the issue #19 to REFDIFF's developers, but they do not answer yet.

Listing 4.4: Using the ECLIPSE Rename Method refactoring.

```
@@ class UMLOperation
-  public boolean isConstructor() {
+  public boolean isConst() {
      return isConstructor;
   }

@@ class UMLModelDiff
private void checkForOperationMoves()
  ...
- else if(r.isConstructor() == a.isConstructor() ...) {
+ else if(r.isConst() == a.isConst() ...) {
```

As another example, ECLIPSE applies the Extract Interface refactoring to the `StaticMapsRequest` class. REFACTORINGMINER 2.0.3 does not detect the Extract Interface refactoring. Moreover, it yields the Change Return Type (12) refactoring (see Listing 4.5). We verify the resulting code and the default parameter of ECLIPSE (Table 4.3) is to replace types where possible to the extracted interface type. This parameter is optional for the mechanics of Extract Interface refactoring, but in ECLIPSE IDE it is a default parameter. In this way, it produces additional transformations, such as Change Return Type refactoring, that can directly impact the number of results in refactoring detection tools. Issue #146 of Listing 4.5 is fixed, and REFACTORINGMINER 2.1.0 correctly detects the applied refactoring.

Listing 4.5: Using the ECLIPSE Extract Interface refactoring.

```
@@ class StaticMapsRequest
- public StaticMapsRequest center(LatLng location) {
+ public IStaticMapsRequest center(LatLng location) {

@@ interface IStaticMapsRequest
+ public interface IStaticMapsRequest {
+   ...
+   IStaticMapsRequest center(LatLng location);
```

ECLIPSE applies the Inline Method refactoring to the `locationBias` method (see Listing 4.6). REFACTORINGMINER detects the Extract Variable (4) refactoring. The method is called in four different locations. This example may help developers to discuss more the temporary variables introduced by the Inline Method refactoring mechanics. ECLIPSE's engine introduces additional statements, causing variable renames or the extraction of temporary variables. Thus, REFACTORINGMINER correctly reports the Extract Variable refactoring. We report issue #121, and it is fixed in REFACTORINGMINER 2.1.0. On the other hand, REFDIFF yields the Inline Method (4) refactoring.

Listing 4.6: Using the ECLIPSE Inline Method refactoring.

```
@@ class FindPlaceFromTextRequest
-  public ... locationBias(LocationBias lb) {
-    return param("locationbias", lb);
-  }

@@ class PlacesApiTest
+   LocationBias lb = new LocationBiasIP();
    ...
    .fields(...)
-     .locationBias(new LocationBiasIP())
+     .param("locationbias", lb)
```

In another case, Listing 4.7 shows the Push Down Method applied to a method that has a generic type as the parameter. A generic type is a generic class or interface that is parameterized over types. The type parameter section is delimited by angle brackets (<>), and it specifies the type parameters, called type variables, `T1`, `T2`, and `Tn`.[8] ECLIPSE replaces the parameterized type with the concrete types. Then, REFDIFF yields the Move Method refactoring. We submit issue #16 to REFDIFF's developers. They explain that the algorithm looks for Push Down candidates, and it enforces the signature of the method is the same. However, it does not deal with the case of replacing a parameterized type with a concrete type. Thus, the algorithm ends up and incorrectly classifies it as Move Method refactoring. The developers know this limitation and can fix it in future versions of the tool.

Listing 4.7: Using the ECLIPSE Push Down Method refactoring.

```
@@ abstract class ManagementWeb<T>
- @Override
- public final void customize(T factory) {
-    ...
- }

@@ class ServletManagement extends ManagementWeb<ConfigurableF>
+ @Override
+ public final void customize(ConfigurableF factory) {
+    ...
+ }
```

Mechanics Are Partially Consistent

In Listing 4.8, ECLIPSE renames the `getNonMappedLeavesT1` method. REFACTORINGMINER 2.0.3 yields that the Change Variable Type refactoring is applied to the `statement` variable. We have checked the generated code after perform the Rename Method refacotring and noticed no changes in the type of this variable. We report issue #139, and developers fixed it in REFACTORINGMINER 2.1.0.

[8]https://docs.oracle.com/javase/tutorial/java/generics/types.html

Listing 4.8: Using the ECLIPSE Rename Method refactoring.

```
@@ class UMLOperationBodyMapper
-  public ... getNonMappedLeavesT1(){
+  public ... getNonMLeavesT1(){
       return nonMappedLeavesT1;

@@ class InlineOperationRefactoring
- for(StatementObject s : b.getNonMappedLeavesT1()) {
+ for(StatementObject s : b.getNonMLeavesT1()) {
  ...
  for(CompositeStatementObject s : b.getNonMappedInnerNodesT1()) {
```

Listing 4.9 shows that ECLIPSE applies the Rename Class refactoring to the `Replacement` class. REFACTORINGMINER detects the Rename Class refactoring along with other 245 refactoring instances, such as the Change Attribute Type, the Change Parameter Type, the Change Return Type, and the Change Variable Type refactorings, since the renamed class are used in several parts of the program. REFDIFF has a similar behavior, and yields 18 instances of the Change Signature Method refactoring. The other refactoring types reported, such as the Change Variable/Attribute/Return/Parameter Type refactorings, are updates to the places where the renamed type is referenced in variable types, return types, parameter types, and field types.

Fowler [9] states that changing each use to the new class name is a step in the refactoring mechanics of the Rename Class refactoring. REFACTORINGMINER yields a coarse-grained refactoring, such as the Rename Class refactoring, and a number of fine-grained transformations used to derive the coarse-grained refactoring, such as, the Change Attribute Type, the Change Parameter Type, the Change Return Type, and the Change Variable Type refactorings. REFACTORINGMINER developers reply a related issue #120 explaining that they prefer to show multiple instances. The refactoring community should discuss about the granularity of each refactoring, and how they relate to coarse-grained transformations.

This example may help developers to discuss more about the correct refactoring mechanics for the Rename Class refactoring. One may argue that the correct refactoring mechanics is to exclude the instances of Change Variable/Attribute/Return/Parameter Type refactoring for which the type change corresponds to the Renamed Class.

Listing 4.9: Using the ECLIPSE Rename Class refactoring.

```
@@ class Replacement;
- public class Replacement {
+ public class Replace {

@@ class AbstractCodeMapping
  private boolean contains(String v) {
-   for(Replacement r : getReplace()) {
+   for(Replace r : getReplace()) {

@@ class TernaryOperatorExpression
- public Replacement m(String s) {
+ public Replace m(String s) {
```

In Listing 4.10, ECLIPSE applies the Move Method refactoring to the `consistency-Check` method. REFACTORINGMINER yields the Inline Method and the Extract and Move Method refactorings. However, ECLIPSE does not apply the Inline Method instances. We report the issue #143, and developers fixed it.

Listing 4.10: Using the ECLIPSE Move Method refactoring.

```
@@ class VariableDeclaration
+ boolean consistencyCheck(...) {

@@ class VariableReplacementAnalysis
- consistencyCheck(v1, v2, set);
+ v1.consistencyCheck(this, v2, set);
  ...
- private boolean consistencyCheck(...) {
```

Suppose a developer pushed down a method to `N` subclasses. REFACTORINGMINER and REFDIFF yield `N` instances of the Push Down Method refactoring. For example, Listing 4.11 shows that ECLIPSE applies the Push Down Method refactoring to the `normalizedEd-itDistance` method, moving it to 11 subclasses using ECLIPSE. REFACTORINGMINER yields 11 instances of the Push Down Method refactoring. REFDIFF also yields the same output. According to Fowler's Push Down Method mechanics [9], this should be considered a single instance of the Push Down Method refactoring.

This example may help developers to discuss more about the correct refactoring mechanics for the Push Down Method, the Extract Method and the Inline Method refactoring. ECLIPSE's default refactoring parameter is to copy/extract the same code fragments. This parameter can fix design flaws, such as the Duplicated Code [9]. However, the end user can change this behavior in the graphical user interface and the refactoring mechanics may be different when a user selects different parameters. One may argue that the correct refactoring mechanics is to exclude the extra refactoring instances reported due to the mechanics. We reported issues #120 and #122 to REFACTORINGMINER's developers. However, they did not accept the issues.

Listing 4.11: Multiple instances of the Push Down Method refactoring in REFACTORINGMINER and REFDIFF.

```
@@ class Replacement
-  public double normalizedEditDistance() { ... }

@@ class CompositeReplacement extends Replacement
+  public double normalizedEditDistance() { ... }
// 10 other subclasses changed
```

Listing 4.12 shows that ECLIPSE applies the Extract Method refactoring, and REFACTORINGMINER yields Extract Method (3) and Rename Variable (2). We submit issue #120 to discuss the additional changes. The developers reject it and explain the extra reported types reported by REFACTORINGMINER are due to the IDE refactoring mechanics. The Rename Variable refactoring is detected in two extracted code fragments, which have a different variable name: `errorResponse` renamed to `response`.

Some of the additional refactoring types reported by REFACTORINGMINER in the Extract Method refactoring are due to the IDE refactoring mechanics, and REFACTORINGMINER reports correct changes. For example, the Parameterize Variable refactoring is reported when local variables declared in the original method become parameters of the extracted method. Likewise, the Rename Parameter refactoring is detected when a parameter of the original method is passed with a different name to the extracted method.

Listing 4.12: The Rename Variable and Extract Method refactoring in REFACTORING-MINER.

```
@@ public class GeoApiContextTest {
-   MockResponse response = new MockResponse();
-   response.setStatus(...);
+   MockResponse response = extracted();
  ...
+ private MockResponse extracted() {
+   MockResponse response = new MockResponse();
+   response.setStatus(...);
+   return response;
+ }
  ...
-   MockResponse errorResponse = new MockResponse();
-   errorResponse.setStatus(...);
+   MockResponse errorResponse = extracted();
```

4.1.5 Threats to Validity

In this section, we discuss some threats to validity. We do not evaluate real transformations applied by developers. However, we apply a number of transformations to four real open-source projects using ECLIPSE. In addition, we manually analyze the candidates yielded after Step 3 of our technique. We submit 34 issues to the developers of the refactoring detection tools. They fixed 16 bugs, 3 were not accepted, and 12 issues are still open. Since this manual classification is a time consuming and error-prone activity, we may miss some bugs.

In our study, we are restricted to a project source code hosted in GitHub. However, the evaluated projects have been actively developed for more than six years. We analyze eight types of refactoring, such as the Rename refactoring, which is frequently applied by developers [25]. Figure 4.1 presents the number of issues reported to the Rename Method and Rename Class refactorings. Moreover, refactoring implementations may introduce behavioral changes when performing a refactoring [41]. As future work, we intend to improve our technique by using SAFEREFACTOR [43] after Step 2 to discard transformations that introduce behavioral changes.

The default parameters used in our study are subject to human errors. However, we addressed that point by inspecting the source code after refactoring when we intend to submit the issues. Table 4.2 shows some default parameters that we define in this study. For example, we specify the parameter of Rename Method refactoring as the method's same with a suffix. In future work, we intend to verify new values for these parameters, which can help to find more issues scenarios.

We only evaluated the refactorings implementations of one IDE, for a particular programming language (Java). ECLIPSE is a popular IDE and is used by some developers to apply refactorings, as discussed by Oliveira et al. [27]. Java is a popular programming language. We also evaluate the best refactoring detection tools available [37,49].

4.1.6 Answers to Research Questions

Next we answer our research questions.

RQ_1 $RQ_{1.1}$: REFACTORINGMINER 2.1.0 is consistent with the refactoring mechanics of ECLIPSE in 74.28% of the transformations. REFACTORINGMINER does not detect refactorings in 4.93% of the cases. REFACTORINGMINER is partially consistent with ECLIPSE in 20.41% of the analyzed transformations. REFACTORINGMINER yields a different result in 0.38% of the analyzed transformations.

$RQ_{1.2}$: REFDIFF 2.0 is consistent with the refactoring mechanics of ECLIPSE in 78.45% of the transformations. REFDIFF does not detect refactorings in 7.16% of the cases. REFDIFF is partially consistent with ECLIPSE in 14.11% of the analyzed transformations. REFDIFF yields a different result in 0.27% of them.

RQ_2 We report 20 issues to REFACTORINGMINER, developers fixed 15 bugs in the Extract Method, Inline Method, Move Method, Push Down Method, Rename Method, and Extract Interface refactorings. Moreover, 3 issues were not accepted, and 2 bug is duplicated. We report 14 issues to REFDIFF, developers fixed 1 bug in the Move Method refactoring, 12 bugs reports are still open, and 1 bug is duplicated.

4.2 Composite Refactorings

In the previous study, we apply a single refactoring per input and output pair. In this study, we use our technique to apply 3 composite refactorings [2] and evaluate 7 refactoring implementations of ECLIPSE using 2 refactoring detection tools in 2 open-source projects. As the first study, we run two instances of the technique for each detection tool, first using ECLIPSE and REFACTORINGMINER, and next using ECLIPSE and REFDIFF.

Section 4.1.1 presents our study definition. Section 4.1.2 describes the experimental setup. Sections 4.1.3 and 4.1.4 present results and discuss them, respectively. Section 4.1.5 describes some threats to validity, and Section 4.1.6 answers our research questions.

4.2.1 Study Definition

Composite refactoring is a set of two or more interrelated single refactorings [2]. This concept is different of nested refactoring, which is a refactoring transformations that takes place in code resulting from the application of another refactoring operation [49].

The scenario of nested refactorings is more challenging for refactoring detection tools, and we do not evaluate it in this study. Nested transformations are successive refactorings applied to the same code fragment. For example, suppose we extract some statements to a method A and then select new statements from the body of method A to create a new method B. In this case, we apply nested Extract Method refactorings.

Our goal is to apply our technique to analyze the impact of composite refactorings. To apply composite refactorings, we have to adapt Step 2 of Figure 3.1 and manually perform the steps of our technique. We follow the same guidelines presented in Section 3.2. In Step 2, we return to Step 1 and apply the three composite refactorings. The output program has subsequent refactorings from different types.

We address the following research questions:

RQ$_{3.1}$ To what extent composite refactorings applied by ECLIPSE are detected by REFACTORINGMINER?

We count the number of refactorings detected by REFACTORINGMINER that are consistent with ECLIPSE. We also count the number of refactorings that are partially consistent, different or empty.

RQ$_{3.2}$ To what extent composite refactorings applied by ECLIPSE are detected by REFDIFF? We count the number of refactorings detected by REFDIFF that are consistent with ECLIPSE. We also count the number of refactorings that are partially consistent, different or empty.

We deliberately search for opportunities in locations not modified by another transformation. The aim is to apply refactorings that do not yield nested refactorings. For example, when we apply the Rename Class refactoring, we do not select methods of this class to apply the Rename Method refactoring. Moreover, we manually apply 30 transformations grouped in 10 pairs of inputs and outputs. For example, Table 4.6 shows subject ID number one contains the Extract Method, the Inline Method, and the Rename Method refactorings.

4.2.2 Experimental Setup

We ran the experiment with the same settings as the previous study described in Section 4.1.2: Core i7 3.1 GHz, 8GB RAM, Fedora 33, and Oracle JDK 1.8. We use two open-source projects integrated with GRADLE as inputs to our technique: GITLINK and TESTNG (see Table 4.5). Thus, we use seven refactoring implementations of ECLIPSE JDT 4.16 in Step 2: the Extract Method, the Rename Method, the Inline Method, the Move Method, the Extract Interface, the Rename Class, and the Push Down Method refactorings. Table 4.2 details the parameters used for each implementation. In Step 3, we use two refactoring detection tools: REFACTORINGMINER 2.1.0[9] (to answer RQ$_{3.1}$), and REFDIFF 2.0[10] (to answer RQ$_{3.2}$).

Table 4.5: Projects used in our evaluation.

Project	Domain	KLOC	Stars	Contributors
GITLINK	A plugin for Git	6.3	108	4
TESTNG	Java testing framework	130	1.6K	168

[9]https://github.com/tsantalis/RefactoringMiner/commit/149468e
[10]https://github.com/aserg-ufmg/RefDiff/commit/889b0bf

4.2.3 Results

In this experiment, we analyze 30 transformations applied by ECLIPSE using seven types of refactorings implemented by ECLIPSE, which were evaluated using REFACTORINGMINER and REFDIFF. Table 4.6 summarizes our results.

Table 4.6: Results of the composite refactorings in feasibility study. RM = REFACTORINGMINER; RD = REFDIFF.

Subject ID	Project	Refactoring	RM	RD
1	GitLink	Extract Method, Inline Method, Rename Method	Consistent	Consistent
2	GitLink	Move Method, Extract Interface, Rename Class	Partially Consistent	Partially Consistent
3	GitLink	Push Down Method, Extract Interface, Rename Method	Partially Consistent	Partially Consistent
4	GitLink	Extract Method, Move Method, Rename Method	Partially Consistent	Partially Consistent
5	GitLink	Push Down Method, Rename Method, Extract Interface	Partially Consistent	Partially Consistent
6	TestNG	Rename Class, Extract Method, Move Method	Partially Consistent	Partially Consistent
7	TestNG	Move Method, Rename Class, Extract Method	Partially Consistent	Partially Consistent
8	TestNG	Extract Method, Rename Class, Inline Method	Partially Consistent	Partially Consistent
9	TestNG	Extract Method, Move Method, Extract Interface	Partially Consistent	Partially Consistent
10	TestNG	Move Method, Push Down Method, Extract Method	Consistent	Consistent

We apply three transformations in 10 subjects of Table 4.6 total of 30 transformations, using randomly the Extract Method, the Rename Method, the Inline Method, the Move Method, the Extract Interface, the Rename Class, and the Push Down Method refactoring implementations.

REFACTORINGMINER and REFDIFF are consistent in 20% (2) transformations with ECLIPSE's refactoring mechanics, respectively. In 80% (8) transformations, REFACTORINGMINER and REFDIFF detect more refactorings. For instance, REFACTORINGMINER detects the Change Variable Type refactoring when applying the Rename Class refactoring. Furthermore, REFDIFF detects other types of refactorings. For example, REFDIFF detects the Change Signature Method refactoring when applying the Rename Class refactoring.

4.2.4 Discussion

This section discusses when the refactoring detection tools yield the applied transformation and more types of refactorings. In Listing 4.13, ECLIPSE applies the Move Method, the Extract Interface, and the Rename Class refactorings. REFACTORINGMINER detects the applied refactorings along to the Change Parameter Type (25), the Change Variable Type (5), the Change Return Type (2), the Change Attribute Type (1) refactorings.

As discussed in Listing 4.9, REFACTORINGMINER detects derived change type refactorings from the Rename Class refactoring when the renamed class is a type of parameters, variables, returns, and attributes. Furthermore, REFDIFF yields the Change Signature Method refactoring when the renamed class is part of the method signature.

Listing 4.13: Using the ECLIPSE Rename Class refactoring.

```
@@ class LineSelection
- public class LineSelection
+ public class SelectionLine
```

In Listing 4.14, ECLIPSE applies the Extract Interface, the Push Down Method, and the Rename Method refactorings. REFDIFF yields the Push Down Method (4), the Pull Up Signature Method (3), the Rename Method (1), the Extract Interface (1) refactorings.

As seen in Listing 4.11, REFACTORINGMINER and REFDIFF yield N instances of the Push Down Method refactoring for each moved method to other N subclasses. ECLIPSE's default refactoring implementation is to extract the same code fragments to preserve behavior. However, the graphical interface can disable this parameter. The multiple instances reported are not wrong, but some studies that depend on the refactoring detection report cannot ensure the refactorings are unique.

Listing 4.14: Using the ECLIPSE Extract Interface, Push Down Method, and Rename Method.

```
@@ IAction / Extract Interface
+ public interface IAction { ... }

@@ class AnnotationAction / Push Down Method
- public void consume( ... ) { ... }
```

```

@@ class BrowserCommitAnnotationAction / Push Down Method
+ public void consume( ... ) { ... }

@@ class URLTemplateProcessor / Rename Method
- private String processFile( ... ) { ... }
+ private String fileProcess( ... ) { ... }
```

Finally, we do not identify any issues after this experiment, but we found several derived refactorings in the Partially Consistent category. However, this behavior follows a similar discussion to the previous study in Section 4.1.4.

4.2.5 Threats to Validity

This section discusses some threats to validity. We also consider threats to validity presented previously in Section 4.1.5, such as behavior preservation, real transformations, default parameters, and the number of open-source projects.

In our study, we do not assess the number of each type of refactoring, but we group three different types in every 10 pairs of input and output. We set some parameters, as seen in Table 4.2, but we describe the default parameters of ECLIPSE in Table 4.3. In addition, we perform our study manually. However, we do not apply refactorings in the same source code file, thus we avoid nested transformations. Finally, we analyze seven types of refactorings, such as the rename-based refactorings, which is frequently applied by developers [25].

Chapter 5

Related Works

Silva et al. [37] propose a language-agnostic refactoring detection tool REFDIFF 2.0. It presents a new refactoring detection algorithm that abstracts the specificity of an programming language by Code Structure Tree. This abstraction allows support some programming languages, such as Java, C/C++, and JavaScript. Tsantalis et al. [49] propose REFACTORINGMINER 2.0, a refactoring detection tool for Java. It relies on an AST-based statement matching algorithm that determines refactoring candidates without requiring user-defined thresholds and covers 40 refactoring types, 25 more refactorings than previous version.

Tsantalis et al. [49] execute REFACTORINGMINER 2.0, GUMTREEDIFF and two versions of REFDIFF on all 536 commits from 185 open-source GitHub-hosted projects monitored over a period of two months of the dataset proposed before [50] and considered the union of all true positives as the ground truth. Two authors validated the refactoring instances [49]. It includes 7,226 true positives in total, for 40 different refactoring types detected by one (minimum) up to six (maximum) different tools. REFDIFF initially used the dataset proposed before [50] to evaluate precision and recall. They also manually included other instances. In our work, we propose a technique to automatically evaluate refactoring detection tools. We evaluate eight refactoring types using 9,885 transformations applied by ECLIPSE to evaluate two refactoring detection tools. Our work can help refactoring detection tool developers to improve their dataset, and find some transformations that may help them improving the refactoring detection rules.

Oliveira et al. [28] propose a technique to identify differences in refactoring mechanics used by tool developers of refactoring implementations. They perform a pairwise comparison of 10 types of refactorings to 157,339 programs using 27 refactoring implementations from ECLIPSE, JRRT, and NETBEANS. Oliveira et al. [27] conduct a survey with 107 developers of popular Java projects to better understand the refactoring mechanics used by them in practice. They found the most developers expect the refactoring output based on their experience and there is no consensus in five out of seven questions in their survey. However, over 50% of the time, the IDEs used by developers yield an output that is different than if they manually apply the same refactoring. They found some differences. In our work, we apply eight types of refactorings to four real open-source projects, and compare the differences between mechanics of two refactoring detection tools and ECLIPSE IDE. We found some differences. These results motivate the importance of discussing more the refactoring mechanics by our community.

Prete et al. [32] develop REF-FINDER, which detects refactorings between two programs versions using a template-based refactoring reconstruction approach. Their tool can identify 63 of 72 refactoring types from Fowler's catalog [9]. To evaluate their tool, they performed two study cases: they create code samples from Fowler's catalog, and they select version pairs from open-source projects. REF-FINDER achieved an overall precision and recall of 79% and 95%, respectively. Dig et al. [6] present an algorithm that detects refactorings performed during component evolution. Their algorithm was implemented as an ECLIPSE plugin called REFACTORINGCRAWLER. They evaluate their tool in three components ranging with 17 KLOC up to 352 KLOC, and its accuracy was over 85% for seven types of refactorings. In this work, we propose a technique that may be used to evaluate their refactoring detection tools.

Soares et al. [42] compare three different approaches based on manual analysis, commit message and dynamic analysis using SAFEREFACTOR [43] to detect refactorings considering behavioral preservation and found the REF-FINDER presented a low precision and recall. Mongiovi et al. [22, 23] improve SAFEREFACTOR by including change impact analysis and skips. We intend to use SAFEREFACTOR after Step 2 to only consider behavior preserving transformations.

There are several works that find the sets of statements to be extracted. Tsantalis and Chatzigeorgiou [48] propose an approach to select related statements that can be extracted. They consider two aspects to identify related statements. First, it selects all statements that computes a given variable. Second, it extracts the statements affecting the state of a given object. Their approach allows producing meaningful and behavior preserving refactoring opportunities.

Silva et al. [38] propose a rank function to classify initial candidates according to their potential to improve program comprehension. Their approach tends to encapsulate well-defined computation with its own set of dependencies and that is also independent of remaining statements of the original method.

Charalampidou et al. [4] suggest resolving the Long Method smell by using the Single Responsibility Principle to identify opportunities of Extract Method refactoring. The approach calculates cohesion between pairs of statements to determine code fragments that collaborate for functionality. Moreover, the approach identifies statements that perform the same functionality.

Xu et al. [52] propose a machine-learning-based approach to recommend Extract Method refactorings based on complexity, cohesion and coupling. They use samples, which were obtained from real-world Extract Method refactorings, to train the probabilistic model. Their tool was evaluated on five open-source repositories and compared against state-of-art approaches: SEMI [4], JEXTRACT [38] and JDEODORANT [8].

In our work, first we try to extract the second statement. If we cannot apply a refactoring, we try to extract the second and third statements. We repeat this process by adding more statements until we successfully apply a refactoring using the IDE or we reach the last statement (see Algorithm 1).

Previous studies discuss refactorings as laws of object-oriented programming. Borba et al. [3] show some laws were formalized and encoded in tools, but not proved sound or complete. Their contributions present algebraic laws for a language similar to a subset of sequential Java. Opdyke [29] shows several preconditions to low-level refactorings, such as checking naming conflicts, the collision of new methods, access control of methods and delete non-referenced methods. In our technique implementation, we use preconditions implemented in ECLIPSE to apply refactorings and study whether composite refactorings can affect themselves.

Opdyke and Johnson [29] define a collection of program restructuring operations that preserves behavior, named refactorings. Later, Roberts [33] automates several of the previous set of refactorings [29], but Tokuda and Batory [46] demonstrate that the preconditions proposed by Opdyke are not sufficient to guarantee behavior preservation after. Furthermore, the findings of refactoring implementations suggest that it is a challenge to provide refactorings with respect to the formal semantics of all language constructs [35]. In our evaluation, we identify differences in the refactoring mechanics of ECLIPSE and two refactoring detection tools and intend to use SAFEREFACTOR after Step 2 to only consider behavior preserving transformations.

Murphy-Hill et al. [24] find the developers do not use refactoring tools to apply refactorings. They argue that the names of refactorings assigned by the refactoring tools are unnecessarily hard to identify between environments. For example, Introduce Explaining Variable from Fowler's catalog [9] is called Extract Local Variable in ECLIPSE. Besides, refactoring names differ between refactorings tools. For example, Figure 2.2 shows the refactor menus of ECLIPSE and INTELLIJ. Generify in INTELLIJ is equivalent to Infer Generic Type Arguments in ECLIPSE. Another example, Introduce Field in INTELLIJ can be used as Convert Local Variable to Field in ECLIPSE. In our evaluation, we identify differences in the names of refactoring of each refactoring detection tool, such as Change Parameter Type in REFACTORINGMINER and Change Signature Method in REFDIFF.

The study of Vakilian et al. [51] shows that the major barrier to the adoption of refactoring tools are not their bugs, but their usability. An example, automated refactorings, such as embedded tools in IDE, were not used because a programmer performs refactoring manually. In addition, more than half of the developers sometimes performed the refactoring manually. Some coarse-grained refactorings are ambiguous, and developers cannot predict the outcome of the refactoring implementation. The interviewees did not know the goals of more than eight automated refactorings on average. Moreover, more than half of the interviewees could not describe the transformation automated by some refactoring and did not use some automated refactorings because of their unpredictability. We evaluate automated and manual refactorings performed by IDE to real projects.

Tempero et al. [45] surveyed 3,785 developers to investigate the barriers to applying refactorings. Their findings show that the decision to apply refactorings or not was due to non-design considerations. They mentioned inadequate tool support as a reason for not refactoring. Kim et al. [18] conduct a field study of refactoring benefits and challenges at

Microsoft. They perform three complementary study methods: a survey, semi-structured interviews with professional software engineers, and quantitative analysis of version history data. Their findings show that approximately half of the participants apply refactorings manually, with an exception of the Rename refactoring. Murphy-Hill et al. [25] provide new insights of analyzing four sets of data about how developers apply refactorings. They find that refactoring implementations are underused specially when we consider refactorings that have a method-level granularity or above. We find refactorings applied to real projects using IDEs may have different refactoring mechanics.

Schäfer et al. [35] present several Java refactoring implementations by translating Java programs to an enriched language. The language allows to specify and check preconditions to apply the transformations more easily. They aim to improve the correctness and applicability of the Eclipse refactoring implementations. Steimann and Thies [44] show that mainstreams IDEs, such as ECLIPSE, NETBEANS, and INTELLIJ, are flawed when it comes to maintaining accessibility. They find scenarios where applying refactoring causes unexpected changes to program behavior, such as Pull Up Members. In our work, we use ECLIPSE API to verify preconditions and apply refactorings to open-source programs.

Daniel et al. [5] propose a technique based on ASTGen, a Java program generator, for automated testing refactoring engines. To evaluate the refactoring correctness, they implemented six oracles that evaluate the output of each transformation. They use the oracles DT, Inverse Transformations, and Custom Oracles to identify incorrect transformations. For example, the Inverse oracle checks whether applying a refactoring to a program, its inverse refactoring to the target program yields the same initial program. If they are syntactically different, the refactoring engine developer has to manually check whether they have the same behavior. They evaluate the technique by testing 42 refactoring implementations, and found three transformation issues using Differential Testing and Inverse oracles in 2 refactoring implementations of ECLIPSE and NETBEANS of the Encapsulate Field refactoring, and only one bug using the Custom oracle.

Gligoric et al. [13] use real systems to reduce the effort for writing test generators using the same oracles [14]. They found 141 bugs related to compilation errors in refactoring implementations for Java and C in 285 hours. Our goal is to evaluate whether refactoring tools have differences in the refactoring mechanics. We applied our technique to real projects and found differences in refactoring mechanics. Moreover, we use the same approach of Gligoric et al. [13] to apply refactorings to all possible locations.

A single refactoring rarely suffices to completely remove code smells, such as God Class [9] and Feature Envy [9, 18, 39, 47]. Bibiano et al. [2] present a quantitative study that addresses the incompleteness nature of composite refactorings that can affect internal quality attributes. They define composite refactoring as two or more interrelated single refactorings applied by the same developer to one or more code elements. Their study consists of computing the frequency of incomplete composites according to the refactoring types constituting each composite. They analyzed 353 incomplete composite refactorings in five software projects and targeted 34 popular refactoring types. They evaluate 11 code metrics that are used to capture four internal quality attributes. Their findings reveal that 71% of incomplete composite refactorings, with at least one Extract Method refactoring, are applied without Move Methods on smelly classes. Besides, they found that 58% of incomplete composite refactorings tended to at least maintain the internal structural quality of smelly classes, thus not generating more harm. We verify whether the refactoring detection tools can detect the composite refactorings.

Tsantalis and Chatzigeorgiou [47] propose a methodology for the identification of Move Method refactoring opportunities by employing the notion of distance between an entity, such as attribute or method, and a class. In this way, they automated the identification of many common Feature Envy bad smells. Likewise, their approach measures the effect of all refactoring suggestions based on a novel Entity Placement metric that quantifies the improvement of system classes after the suggestion is performed. Their evaluation consists of quantitative analysis of the refactoring suggestions using open-source projects, a study about the evolution of coupling and cohesion metrics when performing the refactoring suggestions, an assessment by an independent designer about conceptual integrity of suggestions, and an efficiency based on computation time for extraction of refactoring suggestions using several open-source projects. They find the approach suggests useful refactoring to assist the designer and to improve design quality, along with is capable of extracting conceptually sound suggestions. Furthermore, the efficiency of their approach depends on the number of extracted refactoring suggestions and the size of the system. In this work, we report 8 issues about Move Method refactoring.

Chapter 6

Conclusions

Section 4.1 proposes and evaluates a technique to test refactoring detection tools, such as REFACTORINGMINER and REFDIFF. Then in Section 4.2, we investigate the impact of software evolution concerning composite refactorings detection in refactoring detection tools, such as REFACTORINGMINER and REFDIFF.

6.1 Refactoring Detection Tools and IDE

We evaluate 9,885 transformations applied to four real open-source projects using eight refactoring types of ECLIPSE. REFACTORINGMINER and REFDIFF are consistent with the refactoring mechanics of ECLIPSE in 74.28% and 78.45% of the transformations, respectively. We report 34 issues to REFACTORINGMINER's and REFDIFF's developers. They fixed 16 bugs, 12 bug reports are still open, 3 bugs are duplicated, and 3 issues are not accepted.

In this study, we aim to see whether there are differences between the refactoring mechanics and IDEs and refactoring detection tools. It does not mean that the issue is always on the refactoring detection tool side. Likewise, when a refactoring detection tool reports refactorings not triggered by IDE, it does not mean that detection is wrong. In other words, the refactoring mechanics implemented by the IDEs may add some optional changes that not always clear to user community, as seen in Listing 4.6, Listing 4.9, and Listing 4.12.

Our results may be useful for developers of refactoring detection tools and refactoring implementations to discuss about what should be considered in the refactoring mechanics of each refactoring type. This process may help to improve both tools.

Moreover, our technique may be useful to improve the process of creating a better dataset to be used to evaluate refactoring detection tools, since it can automatically yield several of transformations. Furthermore, it avoids the bias of the authors of the refactoring detection tool to manually classify transformations [49]. Our findings may help the developers to improve refactoring detection tools and refactoring automation tools. Likewise, it alerts the research community about the risks of not considering related biases in their work.

To minimize the problem of depending on the refactoring mechanics implemented by IDEs, developers may use advanced refactoring tools, such as REFAZER [34]. The tool is able to generate a transformation based on few examples of transformations given by the user. In this way, developers do not need to follow a fixed refactoring mechanics implemented by each IDE.

6.2 Composite Refactorings

In this experiment, we evaluate 30 transformations grouped in 10 pairs of inputs and outputs applied to two real open-source projects using seven refactoring types of ECLIPSE. REFACTORINGMINER and REFDIFF are consistent with the refactoring mechanics of ECLIPSE in 20% of the transformations, and tools detect more refactorings in 80% of the transformations. We do not identify any additional issue after this experiment, and the results has a similar discussion of the previous study in Section 4.1.4.

These results show evidence that composite refactorings do not impact each other. Consequently, we can skip the additional effort of producing composite refactorings in new experiments and use unique refactorings such as in the first experiment.

6.3 Future Work

As future work, we intend to evaluate more types of refactorings and increase the number of evaluated projects. We mean at evaluating refactorings implemented by other popular IDEs, such as INTELLIJ and NETBEANS. Moreover, we aim at performing a similar study to evaluate refactorings overlapped with other changes instead of evaluating refactorings in isolation.

Furthermore, we intend to improve our technique to focus on applying refactorings only on realistic opportunities and consider using the previous approaches [4, 38, 48, 52] to find the sets of statements to be extracted in the Extract Method refactoring. We aim to study the mechanics implemented by IDEs and see how they can be modified to be consistent with refactoring detection tools.

Additionally, we intend to evaluate more types of refactorings and increase the number of evaluated projects. Finally, we aim at evaluating the refactorings mechanics of other popular IDEs, such as INTELLIJ and NETBEANS, and more programming languages supported by REFDIFF, such as JavaScript and C/C++.

Bibliography

[1] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[2] Ana Carla Bibiano, Vinicius Soares, Daniel Coutinho, Eduardo Fernandes, João Lucas Correia, Kleber Santos, Anderson Oliveira, Alessandro Garcia, Rohit Gheyi, Baldoino Fonseca, Márcio Ribeiro, Caio Barbosa, and Daniel Oliveira. How does incomplete composite refactoring affect internal quality attributes? In *Proceedings of the International Conference on Program Comprehension*, ICPC, page 149–159, 2020.

[3] Paulo Borba, Augusto Sampaio, Ana Cavalcanti, and Márcio Cornélio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, 52(1):53–100, 2004.

[4] Sofia Charalampidou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Antonios Gkortzis, and Paris Avgeriou. Identifying extract method refactoring opportunities based on functional relevance. *IEEE Transactions on Software Engineering*, 43(10):954–974, 2017.

[5] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the Foundations of Software Engineering*, pages 185–194. ACM, 2007.

[6] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated Detection of Refactorings in Evolving Components. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP, pages 404–428, 2006.

[7] Eclipse.org. Eclipse Project. `http://www.eclipse.org`, 2020.

[8] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. JDeodorant: Identification and application of extract class refactorings. In *Proceedings of the International Conference on Software Engineering*, ICSE, page 1037–1039, 2011.

[9] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[10] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[11] Martin Fowler. Catalog of refactorings. https://refactoring.com/catalog/, 2020.

[12] Git community. Git. https://git-scm.com, 2020.

[13] Milos Gligoric, Farnaz Behrang, Yilong Li, Jeffrey Overbey, Munawar Hafiz, and Darko Marinov. Systematic testing of refactoring engines on real software projects. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP, pages 629–653, 2013.

[14] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 225–234. ACM, 2010.

[15] Gradle.org. Gradle Build Tool. https://gradle.org/, 2020.

[16] JetBrains. IntelliJ IDEA. https://www.jetbrains.com/idea/, 2020.

[17] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A Field Study of Refactoring Challenges and Benefits. In *Proceedings of the Foundations of Software Engineering*, FSE, pages 50:1–50:11, 2012.

[18] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. An empirical study of refactoring challenges and benefits at microsoft. *IEEE TSE*, 40(7):633–649, 2014.

[19] Osmar Leandro, Rohit Gheyi, Leopoldo Teixeira, Márcio Ribeiro, and Alessandro Garcia. Comparing the refactoring mechanics of refactoring detection tools and IDEs (artifacts). https://github.com/osmarleandro/comparing-mechanics, 2021.

[20] Tom Mens and Tom Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[21] Microsoft. Code - OSS (Visual Studio Code). https://github.com/Microsoft/vscode/, 2020.

[22] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira, and Paulo Borba. Making refactoring safer through impact analysis. *Science of Computer Programming*, 93:39–64, 2014.

[23] Melina Mongiovi, Gustavo Mendes, Rohit Gheyi, Gustavo Soares, and Márcio Ribeiro. Scaling testing of refactoring engines. In *Proceedings of the International Conference on Software Maintenance and Evolution*, ICSME, pages 371–380, 2014.

[24] Emerson Murphy-Hill, Moin Ayazifar, and Andrew P. Black. Restructuring software with gestures. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*, VL/HCC, pages 165–172, 2011.

[25] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[26] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. A Comparative Study of Manual and Automated Refactorings. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP, pages 552–576, 2013.

[27] Jonhnanthan Oliveira, Rohit Gheyi, Melina Mongiovi, Gustavo Soares, Márcio Ribeiro, and Alessandro Garcia. Revisiting the refactoring mechanics. *Information and Software Technology*, 110:136–138, 2019.

[28] Jonhnanthan Oliveira, Rohit Gheyi, Felipe Pontes, Melina Mongiovi, Márcio Ribeiro, and Alessandro Garcia. Revisiting refactoring mechanics from tool developers' per-

spective. In *Proceedings of the Brazilian Symposium on Formal Methods*, SBMF, pages 25–42, 2020.

[29] William Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[30] Oracle. Netbeans IDE. `http://www.netbeans.org`, 2020.

[31] Felipe Pontes, Rohit Gheyi, Sabrina Souto, Alessandro Garcia, and Márcio Ribeiro. Java reflection API: revealing the dark side of the mirror. In *Proceedings of the Foundations of Software Engineering*, FSE, pages 636–646, 2019.

[32] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. Template-based Reconstruction of Complex Refactorings. In *Proceedings of the International Conference on Software Maintenance*, ICSM, pages 1–10, 2010.

[33] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[34] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 404–415, 2017.

[35] Max Schäfer and Oege de Moor. Specifying and implementing refactorings. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'10, pages 286–301, 2010.

[36] Danilo Silva. *Mining Refactorings from version histories: studies, tools, and applications*. PhD thesis, Federal University of Minas Gerais, 2020.

[37] Danilo Silva, Joao Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. RefDiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, 47(12):2786–2802, 2020.

[38] Danilo Silva, Ricardo Terra, and Marco Tulio Valente. Recommending automated extract method refactorings. In *Proceedings of the International Conference on Program Comprehension*, ICPC, page 146–156, 2014.

[39] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the Foundations of Software Engineering*, FSE, pages 858–870, 2016.

[40] Danilo Silva and Marco Tulio Valente. Refdiff: Detecting refactorings in version histories. In *International Conference on Mining Software Repositories*, MSR, pages 269–279, 2017.

[41] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. Automated Behavioral Testing of Refactoring Engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, 2013.

[42] Gustavo Soares, Rohit Gheyi, Emerson Murphy-Hill, and Brittany Johnson. Comparing approaches to analyze refactoring activity on software repositories. *Journal of Systems and Software*, 86(4):1006–1022, April 2013.

[43] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making Program Refactoring Safer. *IEEE Software*, 27(4):52–57, 2010.

[44] Friedrich Steimann and Andreas Thies. From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP, pages 419–443, 2009.

[45] Ewan Tempero, Tony Gorschek, and Lefteris Angelis. Barriers to Refactoring. *CACM*, 60(10):54–61, 2017.

[46] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. In *Proceedings of the International Conference on Automated Software Engineering*, ASE, pages 174–181, 1999.

[47] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *Transactions on Software Engineering*, 35(3):347–367, 2009.

[48] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, 2011.

[49] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering*, 48(3):930–950, 2020.

[50] Nikolaos Tsantalis, Matin Mansouri, Laleh Mousavi Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 483–494, 2018.

[51] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. Use, Disuse, and Misuse of Automated Refactorings. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 233–243, 2012.

[52] Sihan Xu, Aishwarya Sivaraman, Siau-Cheng Khoo, and Jing Xu. Gems: An extract method refactoring recommender. In *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE, pages 24–34, 2017.

[53] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers, 2nd edition, 2009.