Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Revisiting the Refactoring Names

Jonhnanthan Victor Pereira Oliveira

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)

Melina Mongiovi (Orientadora)

Campina Grande, Paraíba, Brasil

©Jonhnanthan Victor Pereira Oliveira, 03/09/2018

O48r Oliveira, Jonhnanthan Victor Pereira.
Revisiting the refactoring names / Jonhnanthan Victor Pereira Oliveira. – Campina Grande, 2018.
65 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia e Elétrica e Informática, 2018.
"Orientação: Prof. Dr. Rohit Gheyi, Profa. Dra. Melina Mongiovi".
Referências.

1. Software. 2. Refactoring Nanes – Survey. 3. Tool Developers. I. Gheyi, Rohit. II. Mongiovi, Melina. II. Título.

CDU 004.4(043)

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

"REVISITING THE REFACTORING NAMES"

JONHNANTHAN VICTOR PEREIRA OLIVEIRA

DISSERTAÇÃO APROVADA EM 03/09/2018

ROHIT GHEYI, Dr., UFCG
Orientador(a)

MELINA MONGIOVI CUNHA LIMA SABINO, Dra., UFCG
Orientador(a)

AUGUSTO CEZAR ALVES SAMPAIO, Dr.
Examinador(a)

SABRINA DE FIGUEIRÊDO SOUTO,
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Refactoring é uma prática chave em metodologias ágeis utilizadas por vários desenvolvedores e disponível em IDEs profissionais. Existem livros e artigos que explicam os refactorings e analisam problemas relacionados aos nomes. Alguns trabalhos identificaram que os nomes de refactorings em ferramentas automatizadas de refactoring podem confundir os desenvolvedores. No entanto, não sabemos até que ponto os nomes dos refactorings são confusos no contexto de transformações de pequena granularidade. Neste trabalho, conduzimos um estudo de método misto a partir de diferentes perspectivas para entender melhor o significado dos nomes dos refactorings para desenvolvedores e desenvolvedores de ferramentas (implementações de refactorings e ferramentas de detecção de refactorings). No primeiro estudo, revisitamos os nomes dos refactorings através de uma pesquisa com 107 desenvolvedores de projetos Java populares no GitHub. Perguntamos a eles sobre o resultado de sete tipos de refatoração aplicados a pequenos programas. Esse estudo identifica que os desenvolvedores não esperam a mesma saída para todas as perguntas, mesmo usando pequenos programas Java como entrada. O significado dos nomes dos refactorings é baseado na experiência dos desenvolvedores para um número deles (71.02%). No segundo estudo, observamos até que ponto as implementações de refatoração têm o mesmo significado dos nomes dos refactorings. Aplicamos 10 tipos de refactorings em 157,339 programas usando 27 implementações de refactorings de três ferramentas, usando a mesma entrada e parâmetros, e comparando as saídas. Categorizamos as diferenças em 17 tipos que ocorrem em 9 de 10 tipos de refactorings implementados por Eclipse, NetBeans e JRRT. No terceiro estudo, comparamos o significado dos nomes dos refactorings usados em uma ferramenta (*RMiner*) que detecta refactorings com implementações de refactorings implementadas por três ferramentas. *RMiner* não produz o mesmo conjunto de refactorings aplicados pelas implementações do Eclipse, NetBeans e JRRT em 48.57%, 35% e 9.22% dos casos, respectivamente. Em geral, desenvolvedores e desenvolvedores de ferramentas usam diferentes significados para os nomes dos refactorings, e isso pode afetar a comunicação entre desenvolvedores e pesquisadores.

Abstract

Refactoring is a key practice in agile methodologies used by a number of developers, and available in professional IDEs. There are some books and papers explaining the refactoring names. Some works identified that the names of some automated refactoring tools are a distraction to developers. However, we do not know to what extent the refactoring names are confusing in the context of small-grained transformations. In this work, we conduct a mixed-method study from different perspectives to better understand the meaning of refactoring names for developers, and tool developers (refactoring implementations, and refactoring detection tools). In the first study, we revisit the refactoring names by conducting a survey with 107 developers of popular Java projects on GitHub. We asked them about the output of seven refactoring types applied to small programs. It finds that developers do not expect the same output to all questions, even using small Java programs as input. The meaning of refactoring names is based on developers' experience for a number of them (71.02%). In the second study, we observe to what extent refactoring implementations have the same meaning of the refactoring names. We apply 10 types of refactorings to 157,339 programs using 27 refactoring implementations from three tools using the same input and parameters, and compare the outputs. We categorize the differences into 17 types that occur in 9 out of 10 refactoring types implemented by Eclipse, NetBeans, and JRRT. In the third study, we compare the meaning of the refactoring names used in a tool (*RMiner*) that detects refactorings to refactoring implementations implemented by three tools. *RMiner* does not yield the same set of refactorings applied by implementations from Eclipse, NetBeans, and JRRT in 48.57%, 35%, and 9.22% of the cases, respectively. Overall, developers and tool developers use different meanings for refactoring names, and this may impact developers' and researchers' communication.

Agradecimentos

Agradeço a Rohit Gheyi e a Melina Mongiovi como meus orientadores nesses dois anos de mestrado na UFCG. Eles sempre buscaram me ensinar tudo que era necessário e que melhoraria meu trabalho. Eles sempre acreditaram no meu potencial e me incentivaram a sempre fazer o melhor que eu conseguisse. Me ensinaram a ser dedicado e acreditar que todas as barreiras podem ser ultrapassadas com calma e perseverança. Além disso, o fato de sempre acreditarem que o trabalho pode ser melhorado me mostrou o quanto ainda tenho o que aprender com eles. Agradeço a UFCG, aos professores e aos funcionários do DSC/COPIN por todo comprometimento e ajudas prestadas ao longo desse trabalho. Agradeço a CAPES pelo apoio ao meu trabalho.

Contents

1 Introduction **1**

1.1 Problem 2

1.2 Solution 3

1.3 Evaluation 4

1.4 Summary of Contributions 5

1.5 Organization 5

2 Background **7**

2.1 Program Refactoring 7

2.1.1 Refactoring Specification 8

2.1.2 Example 8

2.2 Refactoring Implementations 11

2.3 JDOLLY 13

3 Revisiting Refactoring Names **17**

3.1 STUDY I: Developers 17

3.1.1 Definition 17

3.1.2 Planning 18

3.1.3 Results 18

3.1.4 Discussion 19

3.1.5 Threats to Validity 25

3.2 STUDY II: Refactoring Implementations 26

3.2.1 Approach 26

3.2.2 Definition 29

3.2.3 Planning 30
3.2.4 Results 31
3.2.5 Discussion 32
3.2.6 Threats to Validity 37
3.2.7 Answers to the Research Questions 38
3.3 STUDY III: Refactoring Detection Tool 40
3.3.1 Approach 40
3.3.2 Definition 43
3.3.3 Planning 43
3.3.4 Results 44
3.3.5 Discussion 47
3.3.6 Threats to Validity 49
3.3.7 Answers to the Research Questions 50

4 Conclusions **51**
4.1 Related Work 53
4.2 Future Work 57

List of Figures

1.1 Inline Method refactoring. 2

2.1 Inline Method specification. 9

2.2 Available options to apply a refactoring to the selected item on NetBeans. . 12

2.3 Available additional parameters to apply the selected refactoring on NetBeans. 13

2.4 Preview on NetBeans. 13

3.1 A question of our survey. 19

3.2 Preference of developers for each kind of refactoring type. 20

3.3 Distribution of answers. 20

3.4 Pull Up Field refactoring. 21

3.5 Pull Up Method refactoring. 22

3.6 Push Down Field refactoring. 22

3.7 Push Down Method refactoring. 23

3.8 Rename Field refactoring. 24

3.9 Encapsulate Field refactoring. 24

3.10 An approach to detect differences in refactoring implementations. First, JDOLLY automatically generates programs as inputs (Step 1). For each generated program, the refactoring implementation attempts to apply the transformation (Step 2). Finally, it identifies differences by using Differential Testing Oracle (Step 3). 27

3.11 The web-based diff view of *GumTree*. 30

3.12 The Inline Method refactoring implementation of NetBeans 8.2 removes some statements. 37

3.13 An approach to detect differences in refactoring implementations and refactoring detection tools. First, JDOLLY automatically generates programs as inputs (Step 1). For each generated program, the refactoring implementation attempts to apply the transformation (Step 2). Finally, we run *RMiner* in each transformation, and we check whether *RMiner* yields the same refactoring type applied by the refactoring implementation. 41

List of Tables

3.1 Number of differences found by our approach. Programs = number of programs; Differences = number of transformations applied by both implementations that have different output programs; - = we could not evaluate the NetBeans implementations. 31

3.2 Type of differences found by our approach. Difference Type = it specifies the difference found; Comparison = it specifies the comparison that showed the related difference; #Diffs = number of differences. 32

3.3 Summary of *RMiner* detection results. #Pairs = number of transformations; Undetected = *RMiner* does not yield the refactoring type applied by the refactoring implementation; Detected = *RMiner* identifies the refactoring applied by the refactoring implementation; Difference = *RMiner* yields a different refactoring type, or more than one refactoring type for the same pair of programs. 47

List of Source Codes

2.1 Input program to apply the Inline Method refactoring. 9

2.2 Result of the Inline Method refactoring application performed by Eclipse JDT. 10

2.3 Result of the Inline Method refactoring application performed by NetBeans and IntelliJ. 10

2.4 Input program to apply the Pull Up Field refactoring. 11

2.5 Result of the Pull Up Field refactoring application. 11

2.6 An example of a program generated by JDOLLY 14

3.1 Input program to apply the Push Down Method refactoring. 27

3.2 Push Down Method refactoring application of Eclipse JDT. 28

3.3 Push Down Method refactoring application of NetBeans. 28

3.4 Rename Field refactoring application of NetBeans. 33

3.5 Rename Field refactoring application of JRRT. 33

3.6 Move Method refactoring application of Eclipse JDT. 34

3.7 Move Method refactoring application of NetBeans. 35

3.8 Input program to apply the Inline Method refactoring. 37

3.9 Inline Method refactoring application of NetBeans 8.2. 37

3.10 Input program to apply the Push Down Method refactoring. 41

3.11 Result of the Push Down Method refactoring application performed by Eclipse JDT. 42

3.12 Input program to apply the Pull Up Method refactoring. 44

3.13 Result of the Pull Up Method refactoring application performed by Eclipse JDT. 45

3.14 Input program to apply the Move Method refactoring. 46

3.15 Result of the Move Method refactoring application performed by NetBeans. 46

3.16 Input program to apply the Pull Up Field refactoring. 47
3.17 Result of the Pull Up Field refactoring application performed by Eclipse JDT. 48

Chapter 1

Introduction

During the life cycle of a software, it may need to be changed to fix bugs, introduce new features and enhancements, improve its internal structure, or make the processing more efficient. Systems continue to evolve over time and become more complex as they grow. Lehman's Laws describes software evolution as a force that is responsible for both the driving of new and/or revising of developments in a system [1].

Opdyke and Johnson coined the refactoring term in a research that described the process and identify common refactorings [2]. Code refactoring, a kind of perfective maintenance [3], is the process of changing the internal structure of a program to improve its internal quality but preserving its external behavior [2; 4; 5]. Later, Fowler popularized it, and explains the principles and best practices through a catalog of 72 refactorings. Each refactoring type has a name to facilitate the communication among developers [5].

Since then, common refactorings have received names and have been explained [4; 5; 6; 7; 8; 9; 10; 11; 12], have been automated and incorporated into refactoring tools (such as Eclipse [13], NetBeans [14], and IntelliJ [15]), and has become a central part of software development processes, such as eXtreme Programming [16].

1.1 Problem

In practice, tool developers implement a refactoring based on their experience, some previous work or formal specifications [4; 5]. Vakilian et al. [17] find that the names of some automated refactorings are confusing, and developers cannot predict the outcomes of complex tools. Murphy-Hill et al. [18] find that the names of the refactoring in some tools are a distraction to the developer because they can vary from one environment to another. For example, Fowler's Introduce Explaining Variable [5] is called Extract Local Variable in Eclipse.

Consider applying the Inline Method refactoring to the `foo` method in the Java input program presented in Figure 1.1. The Inline Method refactoring puts the method's body into the body of its callers, and removes the method from the program [5]. The code presented in Figure 1.1(A) does not remove the `foo` method. Moreover, in this example, it adds more statements in the body of the `m` method. If we use tools to apply this refactoring, Eclipse JDT 4.5 yields the program described in Figure 1.1(A). This example is a test case from the Eclipse test suite.

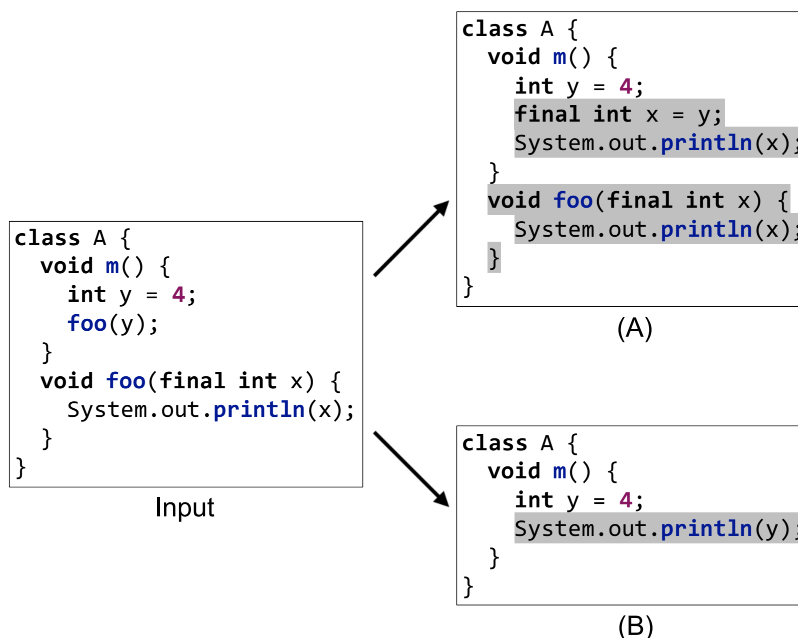


Figure 1.1: Inline Method refactoring.

If we provide the same input program and parameters to NetBeans 8.2 and IntelliJ IDEA 2017.3.5 to apply this refactoring, the tools yield the output program presented

in Figure 1.1(B). Different outcomes from tools show misunderstandings among tool developers. The example showed a single transformation performed in a small Java program that preserves the external behavior but the scenario may be even worse when considering coarse-grained refactorings applied to larger Java programs, and may impact developers that use those tools to apply refactorings. However, we do not know to what extent these differences occur.

1.2 Solution

In this work, we conduct a mixed-method study from different perspectives to better understand the meaning of refactoring names used by developers and tool developers. For simplicity, we consider only the refactoring name to also indicate that the mechanics may be different. The first study considers the developers' perspective. We revisit the refactoring names by conducting a survey with 107 developers of popular Java projects on GitHub to better understand the meaning of the refactoring names used by them in practice. We asked them about the output of seven refactoring types available in popular IDEs. We asked developers the output of the Rename Field, Inline Method, Encapsulate Field, Pull Up Field/Method, and Push Down Field/Method refactorings applied to small programs.

The second study observe to what extent refactorings implemented by developers have the same meaning as the homonymous refactorings found in the literature [4; 5]. To evaluate it, we use a number of small Java programs, using JDOLLY [19], an automated Java program generator, with at most 10 LOC. We use them as input to Eclipse, NetBeans, and JastAdd Refactoring Tools (JRRT [8]). Eclipse and NetBeans are popular IDEs that allow developers to apply refactorings. JRRT improves the correctness and applicability of refactorings by using formal techniques [8]. These tools contain a number of refactoring implementations, such as Rename Class, Pull Up Method, and Encapsulate Field. We apply the homonymous refactorings to the same input program and parameters to different implementations. We perform a pairwise comparison (Eclipse X NetBeans, Eclipse X JRRT, and NetBeans X JRRT) to identify differences. We consider 27 refactoring implementations of 10 types of refactorings (Pull Up Method/Field, Push Down Method/Field, Rename Class/Method/Field, Add Parameter, Encapsulate Field, and Move Method) of Eclipse, NetBeans, and JRRT in

our evaluation.

The third study compares the meaning of the refactoring names used in *RefactoringMiner* (*RMiner*) [11] to refactoring implementations of Eclipse JDT, NetBeans, and JRRT. *RMiner* is a tool that detects refactorings implemented by researchers. Researchers used refactoring detection to empirically study [20; 21; 22; 23; 24; 25; 26] software evolution, and to support other software engineering tasks, such as library adaptation [27; 28; 29; 30], software merging [31], code completion [32; 33], and code review [34; 35; 36]. *RMiner* is a novel technique that overcomes limitations, such as the requirement to build the project under analysis. *RMiner* has an improved precision and recall over the previous works [37; 38; 30]. We evaluate *RMiner* with a set of transformations applied by 18 refactoring implementations of 7 refactoring types. We provide an input program and a refactored program yielded by Eclipse, NetBeans, and JRRT in our previous study.

1.3 Evaluation

Our survey finds that developers do not expect the same output. In some cases, such as Rename Field (46.73%) and Pull Up Method (28.04%), they do not agree on whether the refactoring should be applied. In other cases, they expect different programs as output. Most developers expect the refactoring output based on their experience (71.02%). A few of them consider the meaning of refactoring names presented in papers, books, and sites (7.48%). This may impact developers' communication. Moreover, most developers (75.70%) do not apply manual refactorings; this finding is different from the results showed by previous studies [17; 39; 40; 41]. They use IDEs to apply refactorings.

Our second study compares the outputs of 157,339 programs. Our approach compares refactoring implementations by pairs (Eclipse x NetBeans, Eclipse x JRRT, and NetBeans x JRRT) to identify differences. Overall, only 6.8% of the refactoring applications do not have differences. In general, 63% of the input programs are not refactored by at least one tool. Overall, Eclipse, NetBeans, and JRRT applied 23.75%, 26.3%, and 32% of the refactorings, respectively. We identified that developers of Eclipse, NetBeans, and JRRT adopt the same meaning to Rename Class refactoring. However, Encapsulate Field refactoring has 94.4% of differences in the comparison between Eclipse x NetBeans, and NetBeans x JRRT.

We evaluate *RMiner* [11] in 76,303 transformations of 7 refactoring types using 18 refactoring implementations of Eclipse JDT, NetBeans, and JRRT applied to small Java programs with at most 10 LOC. We run *RMiner* in each transformation applied by a tool to see whether *RMiner* yields the same refactoring type applied by the tool. *RMiner* does not detect 46% of the refactorings applied by Eclipse JDT, 30% of JRRT, and 6% of NetBeans in 7 refactoring types. *RMiner* does not detect over 96% of Pull Up Method refactoring applied by Eclipse JDT and JRRT. The same occurs in over 24% of Rename Class refactoring application. Moreover, *RMiner* does not detect all applications of the seven refactorings applied by NetBeans. In some cases, *RMiner* detects more refactoring types in a single refactoring applied by the tool. For example, 34% of the detection of Move Method refactoring applied by Eclipse JDT yields other types of refactorings, such as Push Down Method. The same occurs in 22% of detection of Move Method refactoring applied by NetBeans and JRRT. In other cases, *RMiner* has the same meaning of the Pull Up Field, and Push Down Method/Field refactorings implemented by Eclipse JDT, and the Pull Up Field and Push Down Field refactorings implemented by JRRT.

1.4 Summary of Contributions

In summary, the main contributions of this work are the following:

- A study to better understand the meaning of refactoring names used by developers in practice (Section 3.1);
- A comparison of 27 refactoring implementations of Eclipse JDT 4.5, NetBeans 8.2, and JRRT (Section 3.2);
- A comparison of *RMiner* to 18 refactoring implementations of Eclipse JDT, NetBeans, and JRRT (Section 3.3).

1.5 Organization

We organize this work as follows. In Chapter 2, we present the background to understand this work. We explain the concept of refactorings, refactoring tools, and an automated Java

program generator. Section 3.1 shows a survey to better understand the refactoring names used by developers in practice. Section 3.2 compares some refactoring implementations of three tools. Next, Section 3.3 compares the meaning of refactoring names used by refactoring implementations and *RMiner*. Finally, we present concluding remarks, and relate our study to others (Chapter 4).

Chapter 2

Background

In this chapter we present the background of some concepts needed for understanding this work. First, we explain program refactoring in Section 2.1. Next, Section 2.2 presents an overview about refactoring tools. Finally, we describe a JDOLLY overview in Section 2.3.

2.1 Program Refactoring

Opdyke originally coined the term refactoring in his PhD thesis [4]. Later, Fowler [5] popularized it. They define code refactoring as the process of modifying a software system in order to improve its internal quality while preserving the observable behavior. The essence of code refactoring consists in a number of small changes that preserve the program behavior. A sequence of small changes (known as refactorings) produces a substantial restructuring. According to Mens and Tourwé [6], the process of code refactoring consists of the following activities:

1. Identify where the software should be refactored;
2. Determine which refactoring(s) should be applied to the identified places;
3. Guarantee that the applied refactoring preserves behavior;
4. Apply the refactoring;
5. Assess the effect of the refactoring on quality characteristics of the software or the process;

6. Maintain the consistency between the refactored program code and other software artifacts.

Over the years, refactoring has become a central part of the software development process, and developers intend to improve their code. Silva et al. [42] investigated the reasons that drive developers to refactor their code. They identified refactorings on 748 Java projects in the GitHub repository. Then, they asked developers why they performed the identified refactorings. Their results indicate that fixing a bug or changing the requirements, such as feature additions, mainly drives refactorings.

In addition, Kim et al. [39] perform a field study of refactoring benefits and challenges at Microsoft through three complementary study methods: a survey, semi-structured interviews with professional software engineers, and quantitative analysis of version history data. The survey participants also reported the benefits they have observed from refactoring. The two most cited benefits were improved readability and maintainability.

2.1.1 Refactoring Specification

Fowler [5] describes the refactorings through a catalog with specifications for 72 refactoring types. He uses a standard format for each specification with five parts: a name, a summary, a motivation, the mechanics, and examples. According to Fowler, the name is important to build a refactoring vocabulary.

The name is followed by a summary. The summary includes a short statement of the problem that the refactoring helps with, a short description of what should be done, and a sketch that shows a simple before and after example. The motivation describes why the refactoring should be done and describes circumstances in which it should not be done. The mechanics are a step-by-step description of how to carry out the refactoring, and the examples show a use of the refactoring to illustrate how it works.

2.1.2 Example

The following example considers the refactoring specification of the Inline Method refactoring. The specification of the Inline Method refactoring starts with the scenario where

a method's body is just as clear as its name. And, the application of this refactoring intends to put the method's body into the body of its callers and remove the method (Figure 2.1).

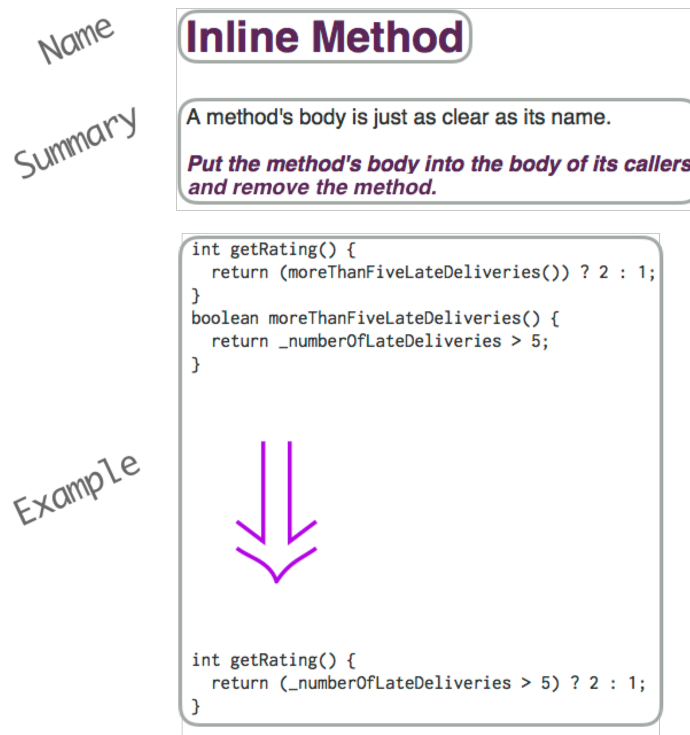


Figure 2.1: Inline Method specification.

Reading the Inline Method refactoring specification as described before, the refactoring is simple. In general, it is not. Problems on how to handle recursion, and multiple return points can appear and makes the refactoring difficult to apply.

The tool developers' perspective may generate some problems. For example, the input code in Listing 2.1 is small and simple. Consider applying the Inline Method refactoring to the method `foo`.

Listing 2.1: Input program to apply the Inline Method refactoring.

```
1 class A {
2   void m() {
3     int y = 4;
4     foo(y);
5   }
6
7   void foo(final int x) {
8     System.out.println(x);
```

```
9     }
10  }
```

Considering the mechanics described by Fowler [5] to perform this refactoring, the refactoring implementation must verify whether the method `foo` is polymorphic. Next, find all calls to method `foo` and replace each call with the method body. Compile and test the refactored program. Finally, remove the method `foo`.

The application presented in Listing 2.3 seems to follow those steps and produced an output program. But, Listing 2.2 does not remove the method and adds a new statement in the code. Furthermore, the code in Listing 2.2 is a test case of Eclipse JDT.

Listing 2.2: Result of the Inline Method refactoring application performed by Eclipse JDT.

```
1  class A {
2    void m() {
3      int y = 4;
4      final int x = y;
5      System.out.println(x);
6    }
7
8    void foo(final int x) {
9      System.out.println(x);
10   }
11 }
```

Listing 2.3: Result of the Inline Method refactoring application performed by NetBeans and IntelliJ.

```
1  class A {
2    void m() {
3      int y = 4;
4      System.out.println(y);
5    }
6  }
```

Another problem is the developers' perspective on the refactoring. For example, the summary of the Pull Up Field refactoring specifies that the scenario to apply this refactoring happens when two subclasses have the same field. For example, consider applying the

Pull Up Field refactoring in the program presented in Listing 2.4. Developers may prefer not applying the refactoring because the scenario contains only one subclass. In contrast, some developers may agree on applying the refactoring and also want to make the field more accessible such as presented in Listing 2.5. This may cause misunderstandings among developers when the scenario has only one subclass.

Listing 2.4: Input program to apply the Pull Up Field refactoring.

```
1 public class A {}
2
3 public class B extends A {
4   int f = 11;
5   public long m() {
6     return f;
7   }
8 }
```

Listing 2.5: Result of the Pull Up Field refactoring application.

```
1 public class A {
2   protected int f = 11;
3 }
4
5 public class B extends A {
6   public long m() {
7     return f;
8   }
9 }
```

2.2 Refactoring Implementations

Roberts proposes the first refactoring tool called Refactoring Browser [43]. It implements a number of refactorings for the Smalltalk [44] language. Refactoring has become more popular, and researchers improves the correctness and applicability of refactorings by using formal techniques through refactoring tools, such as JRRT [8]. In addition, most of the current IDEs have implemented refactorings, such as Eclipse [13], NetBeans [14],

IntelliJ [15].

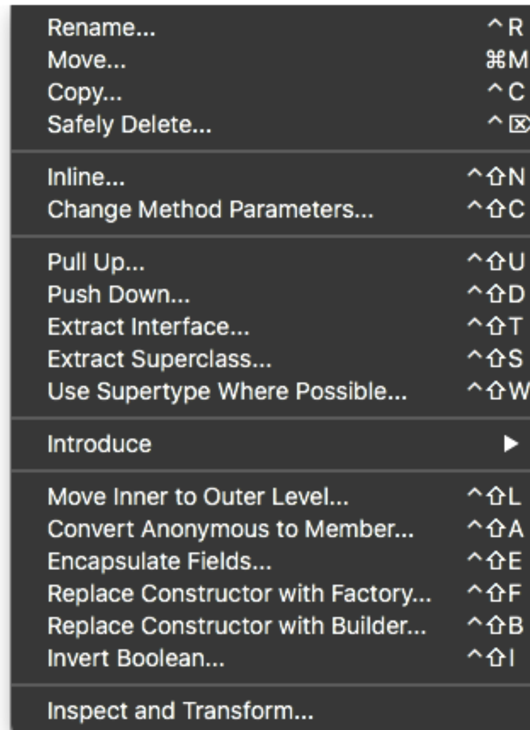


Figure 2.2: Available options to apply a refactoring to the selected item on NetBeans.

For example, Figure 2.2 shows available refactorings when the developer selects an item to refactor using NetBeans. A refactoring tool allows developers to select the refactoring to be applied and the parameters for configuration. For example, consider applying the Encapsulate Field refactoring in a field using the NetBeans IDE. Figure 2.3 shows options that the developer can choose to apply the refactoring. In addition, NetBeans allows the developer to see the preview of the transformation by pressing the *Preview* button (Figure 2.4), which allows the developer to manually inspect the correctness of the transformation.

The first version of Eclipse, the first IDE to implement refactorings, released in the end of 2001, included the following refactorings: Rename, Move, and Extract Method [45]. Murphy et al. [46] conduct a survey on Java software development by using Eclipse. They analyze the use of the Eclipse refactorings by 41 developers. The five most used refactorings were: Rename, Move, Extract Method, Pull Up Method, and Add Parameter.

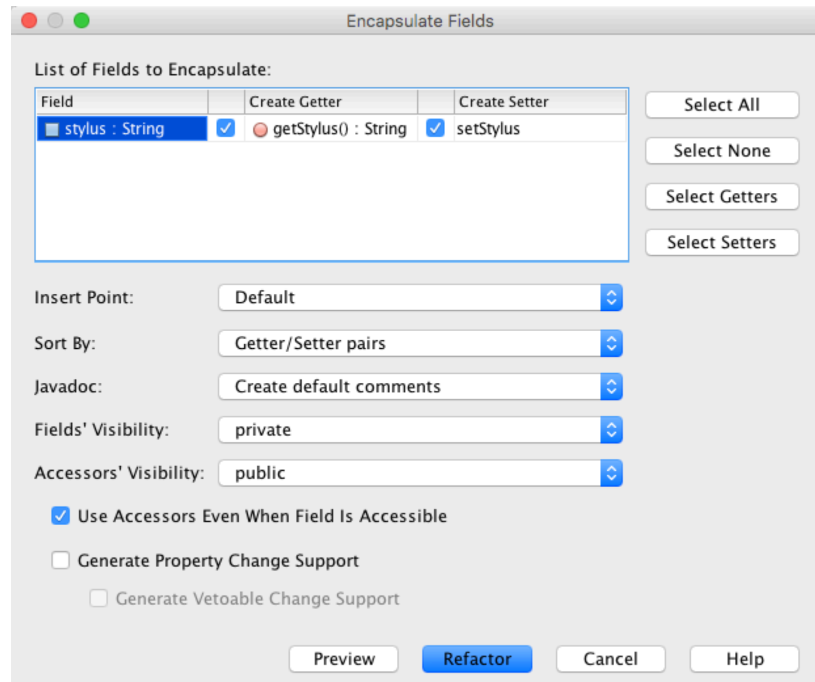


Figure 2.3: Available additional parameters to apply the selected refactoring on NetBeans.

Figure 2.4: Preview on NetBeans.

2.3 JDOLLY

JDOLLY [19] is an automated and bounded-exhaustive Java program generator [19; 47; 48] based on Alloy, a formal specification language [49]. JDOLLY receives as input an Alloy specification with the scope, which is the maximum number of elements (classes, methods, fields, and packages) that the generated programs may declare, and additional constraints for guiding the program generation. It uses the Alloy Analyzer tool [50], which takes an Alloy specification and finds a finite set of all possible instances that satisfy the constraints

within a given scope. JDOLLY translates each instance found by the Alloy Analyzer to a Java program. It reuses the syntax tree available in Eclipse JDT for generating programs from those instances. Listing 2.6 illustrates an example of a program generated by JDOLLY.

Listing 2.6: An example of a program generated by JDOLLY

```
1  package p1;
2  public class A {
3    public int m() {
4      return 1 ;
5    }
6  }
7
8  package p2 ;
9  import p1.*;
10 public class B extends A {}
11
12 package p1 ;
13 public class B extends A {}
```

An Alloy specification is a sequence of signatures and constraints paragraphs declarations. A signature introduces a type and can declare a set of relations. Alloy relations specify multiplicity using qualifiers, such as *one* (exactly one), *lone* (zero or one), and *set* (zero or more). In Alloy, a signature can extend another, establishing that the extended signature is a subset of the parent signature. For example, the following Alloy fragment specifies part of the Java meta-model of JDOLLY encoded in Alloy. A Java class is a type, and may extend another class. Additionally, it may declare fields and methods.

```
sig Type {}
sig Class extends Type {
  extend: lone Class,
  methods: set Method,
  fields: set Field
}
sig Method {}
sig Field {}
```

A number of well-formedness constraints can be specified for Java. For instance, a class

cannot extend itself. In Alloy, we can declare facts, which encapsulate formulas that always hold. The *ClassCannotExtendItself* fact specifies this constraint. The *all*, *some*, and *no* keywords denote the universal, existential, and non-existential quantifiers respectively. The ˆ and ! operators represent the transitive closure and negation operators respectively. The dot operator (.) is a generalized definition of the relational join operator.

```
fact ClassCannotExtendItself {
 all c: Class | c ! in c.^extend
}
```

The Alloy model is used to generate Java programs using the *run* command, which is applied to a predicate, specifying a scope for all declared signatures in the context of a specific Alloy model. Predicates (*pred*) are used to encapsulate reusable formulas and specify operations. For example, the following Alloy fragment specifies that we should run the *generate* predicate using a scope of 3. The user can also specify different scopes for each signature.

```
pred generate[] {...}
run generate for 3
```

The user can guide JDOLLY to generate more specific programs. For example, to generate programs to test the Pull Up Method refactoring, JDOLLY uses the following additional constraints. It specifies that a program must have at least one class (*C2*) extending another class (*C1*), and that *C2* declares at least one method (*M1*).

```
one sig C1, C2 extends Class {}
one sig M1 extends Method {}
pred generate[] {
  C1 = C2.extend
  M1 in C2.methods
}
```

Furthermore, developers can specify a skip number to jump some of the Alloy instances. For a skip of size *n* such that *n*>1, JDOLLY generates one program from an Alloy instance, and jumps the following *n*-1 Alloy instances. Consecutive programs generated by JDOLLY tend to be very similar, potentially detecting the same kind of bug [47; 49]. Thus, developers can set a parameter to skip some of the generated programs to reduce the time needed to

test the refactoring implementations. It avoids generating an impracticable number of Alloy instances by the Alloy Analyzer.

Chapter 3

Revisiting Refactoring Names

We revisit the refactoring names by conducting a mixed-study from three different perspectives. We organize this chapter as follows. Section 3.1 describes the developers' perspective and discusses the results. Section 3.2 presents the evaluation of the tool developers' perspective, and discusses the results. Next, Section 3.3 presents the evaluation of the researchers' perspective.

3.1 STUDY I: Developers

In this section, we survey developers that contribute to popular Java projects. First, we explain the study definition (Section 3.1.1), and planning (Section 3.1.2). Sections 3.1.3 and 3.1.4 present and discuss the results, respectively. Finally, Section 3.1.5 describes some threats to validity.

3.1.1 Definition

The goal of this study consists on analyzing the differences from developers' perspective for the purpose of evaluating it with respect to identify misunderstandings among developers concerning refactoring implementations.

3.1.2 Planning

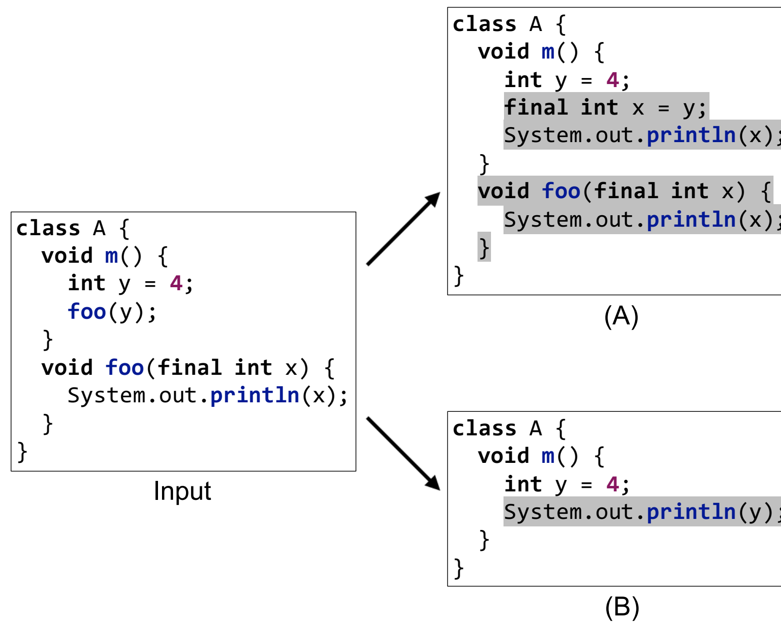
To recruit participants, we sent e-mails to 7,500 developers randomly selected from 124 popular GitHub Java projects, including projects from Google, Facebook, and the Apache Foundation. We divided the survey into three main sections. The first section asks developers about the output of seven refactoring types (Rename Field, Inline Method, Pull Up Field/Method, Encapsulate Field, and Push Down Field/Method) that are available in the mainstream IDEs. We present an input program, a refactoring name, and a parameter, and ask what is the expected program after applying a single refactoring. Each question shows a Java program with at most 10 LOC on the left-hand side (input), and asks to apply a refactoring to it. We present four options: it yields program (A), it yields program (B), the refactoring cannot be applied, and another for the developer to mark in case of an expected different output. The two Java programs on the right-hand side representing options (A) and (B) for each question are yielded by applying a refactoring using Eclipse, NetBeans, IntelliJ, or JRRT. We do not mention in our survey that options (A) and (B) are derived from them. In the last option, there is an open text box in case developers think about a different output. We presented all questions in the same order to all developers. Figure 3.1 shows one question of our survey.

The second section asks more information about refactoring activities: (i) Which tool do you use when applying refactorings? and (ii) How do you know the expected output of a refactoring application? We used open text boxes after each question to allow developers to explain their choices. The third section asks developers for additional comments.

3.1.3 Results

Overall, 107 developers completed the survey. In all questions, we do not have a consensus (Figure 3.2). In some cases, developers prefer to apply a refactoring, while others prefer to not apply it. Different from previous works [39; 40], most developers use IDEs to apply refactorings (75.70%), such as IntelliJ and Eclipse. Only 3.74% manually apply refactorings. The other ones use both strategies (Figure 3.3a). Most developers expect the refactoring output based on their experience (71.02%). A few of them consider the meaning of refactoring names presented in papers, books, and sites (7.48%). The other ones use both

Consider applying the Inline Method refactoring to A.foo() in the following input program in Java. Which is the expected program after applying it?



- ○ It yields program (A)
- ○ It yields program (B)
- ○ The refactoring cannot be applied
- ○ Other: ____________

Figure 3.1: A question of our survey.

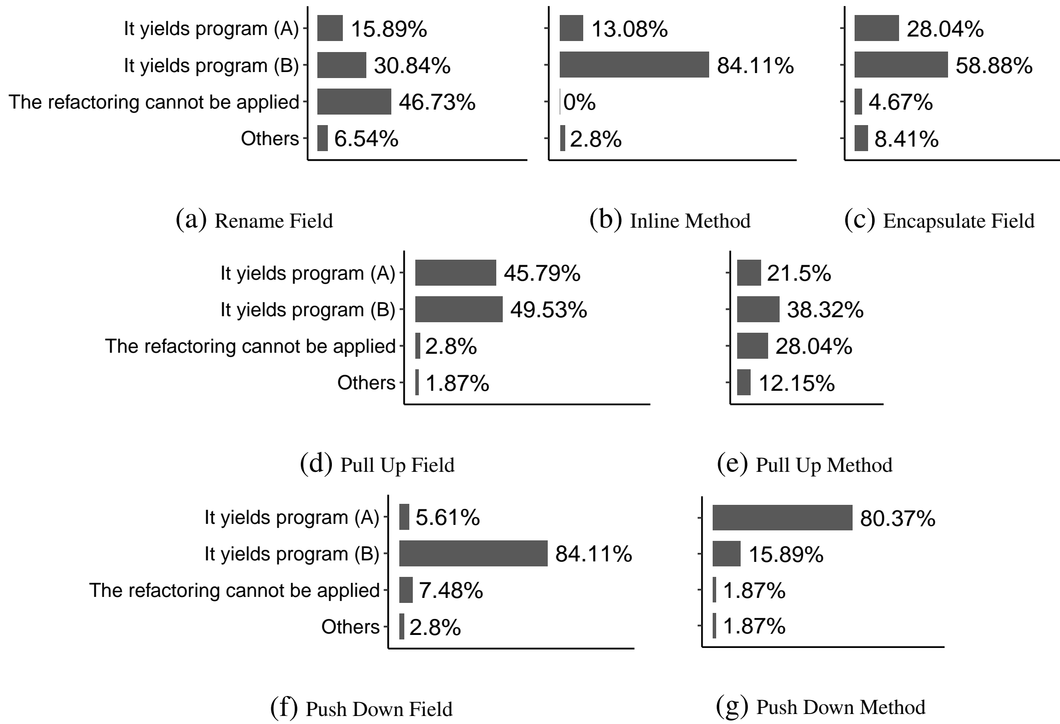
(Figure 3.3b).

3.1.4 Discussion

Next, we explain the results of all questions in our survey.[1] The Inline Method refactoring puts the method's body into the body of its callers, and remove the refactored method [5]. Consider the application of the Inline Method refactoring to the input program (Figure 1.1). We include two output options in Figures 1.1(A) and 1.1(B). A number of developers (84.11%) prefer to remove `foo` (Figure 1.1(B)). However, 13.08% of developers prefer maintaining `foo` (Figure 1.1(A)), and also adding a declaration of a final field. Some developers (2.8%) expect both outputs.

The Pull Up Field refactoring moves two fields in subclasses to its direct superclass [5].

[1] Survey: `https://goo.gl/XtCZph`

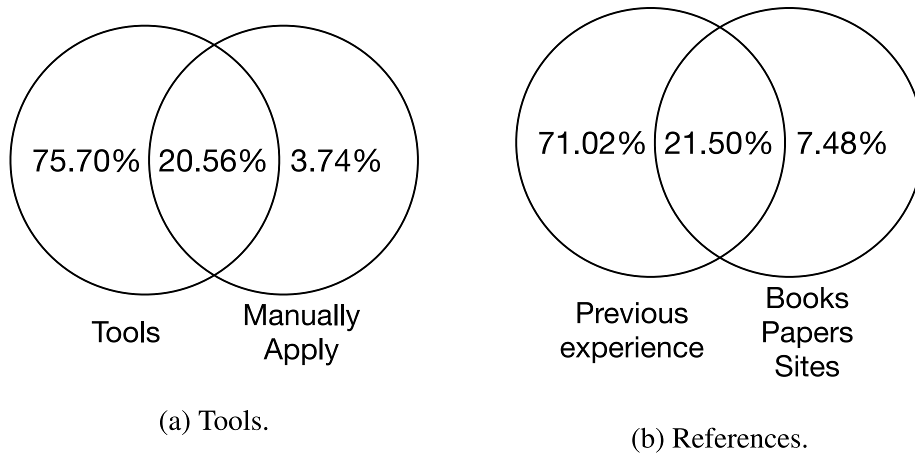


(a) Rename Field (b) Inline Method (c) Encapsulate Field

(d) Pull Up Field (e) Pull Up Method

(f) Push Down Field (g) Push Down Method

Figure 3.2: Preference of developers for each kind of refactoring type.



(a) Tools. (b) References.

Figure 3.3: Distribution of answers.

We asked developers to pull up a field in one subclass to its direct superclass (Figure 3.4). Only 2.8% of developers prefer not to apply the refactoring. But, 49.53% of them expect to pull up the field to the superclass, and preserve the field accessibility (Figure 3.4(B)). Moreover, 45.79% of them expect to pull up the field, but they also want to make the field more accessible (Figure 3.4(A)). Others (1.87%) expect both outputs.

The Pull Up Method refactoring removes methods with identical results on subclasses

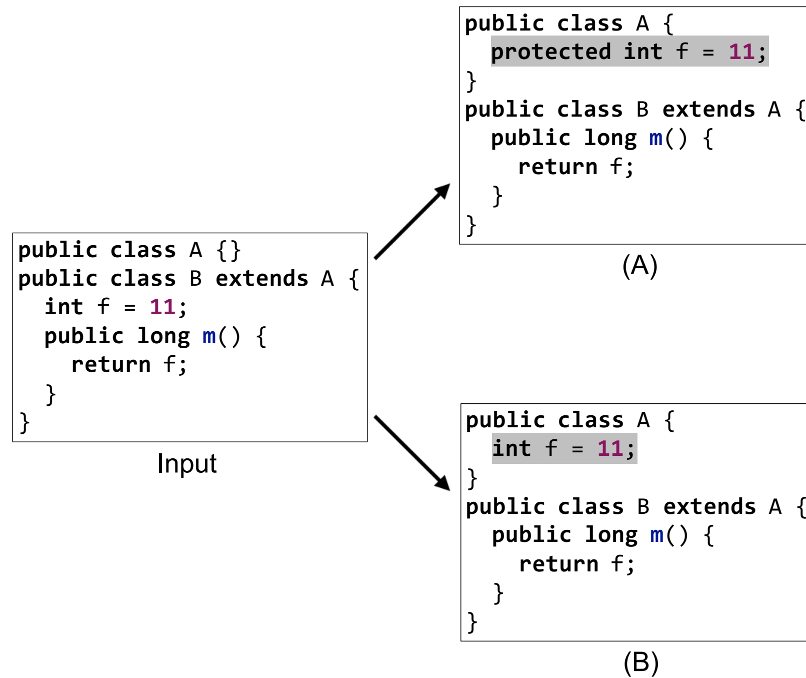


Figure 3.4: Pull Up Field refactoring.

and introduces to a superclass [5]. We asked developers to pull up an abstract method `m` in a subclass `B` to its abstract superclass `A`. There is also a concrete class `C` that extends `A` (Figure 3.5). Some developers (28.04%) prefer not to apply the refactoring. Others (21.50%) expect to apply the refactoring, and also introduce a concrete method `m` with a default body in `C` (Figure 3.5(A)). Some developers (38.32%) prefer to apply it, and also make `C` an abstract class (Figure 3.5(B)). Others (5.06%) expect that class `C` remains concrete with a compilation error.

The Push Down Field refactoring moves a field from a superclass to some subclasses [5]. We asked developers to push down a field in a superclass to its subclass (Figure 3.6). A few developers (7.48%) prefer not to apply the refactoring. Besides that, 5.61% of them expect to apply the refactoring without updating some field calls (Figure 3.6(A)). It introduces a behavioral change. A number of developers (84.11%) prefer to apply the refactoring updating some field calls to preserve behavior (Figure 3.6(B)).

The Push Down Method refactoring moves a method from a superclass to some subclasses [5]. We asked developers to push down a method (Figure 3.7). Only 1.87% of developers prefer not to apply the refactoring. Besides that, 80.37% of them expect to move the method to its direct subclass (Figure 3.7(A)). However, others (15.89%) expect to

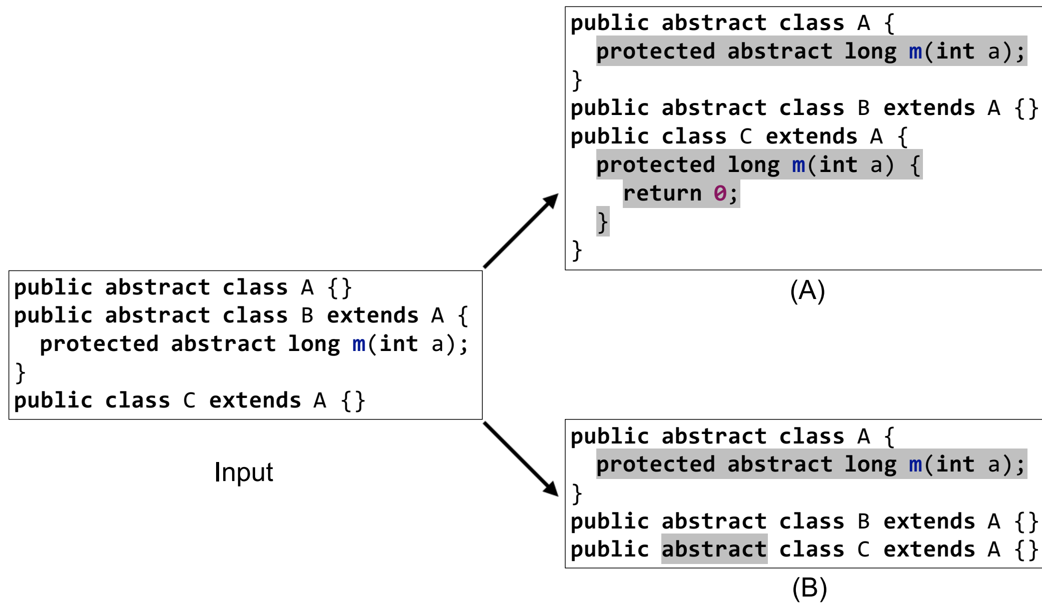


Figure 3.5: Pull Up Method refactoring.

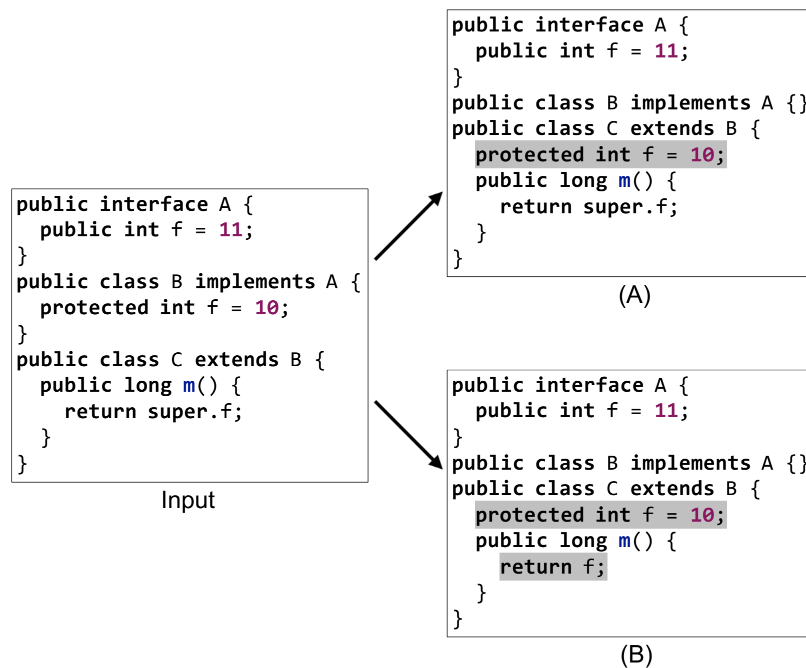


Figure 3.6: Push Down Field refactoring.

move the method not only to its direct subclass, but also to a subclass of its direct subclass (Figure 3.7(B)). Others (1.87%) expect the tool to yield a warning in both options.

The Rename Field refactoring renames a field name. Although Fowler [5] does not describe it, it is available in mainstream IDEs. We asked developers to rename a field

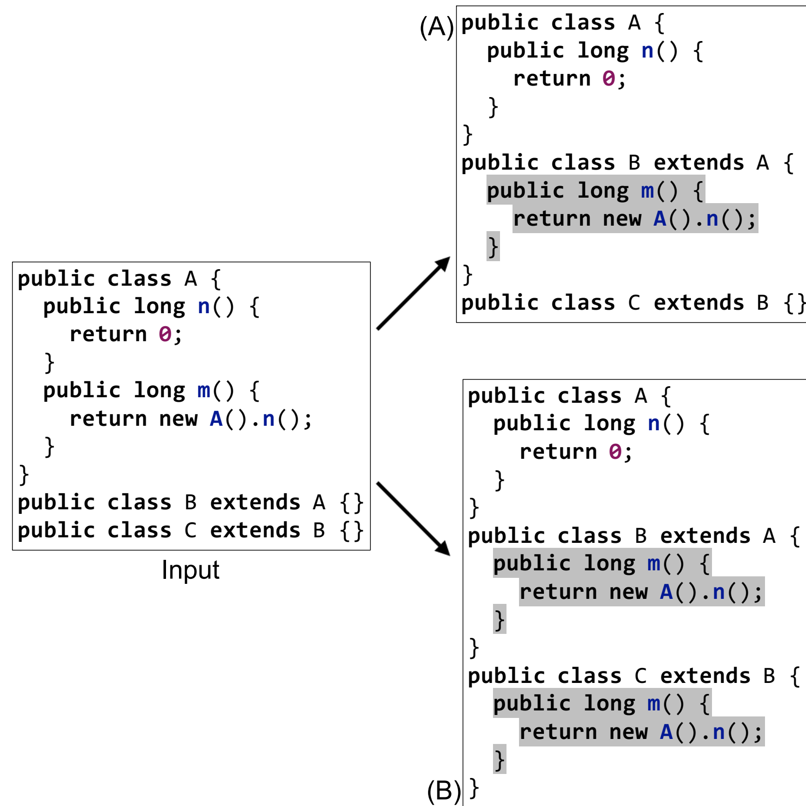


Figure 3.7: Push Down Method refactoring.

(Figure 3.8). Some developers (46.73%) prefer not to apply the refactoring. Moreover, 30.84% prefer to apply the refactoring considering an output that introduces a behavioral change. It only changes the field name (Figure 3.8(B)). Some of them (2.80%) prefer the same thing but they expect the tool to yield a warning. Others (15.89%) prefer to apply the refactoring, and update some calls to the field to avoid behavioral changes (Figure 3.8(A)).

The Encapsulate Field refactoring encapsulates a public field and provide accessors methods [5]. We asked developers to encapsulate a private field (Figure 3.9). Only 4.67% of developers prefer not to apply the refactoring. In addition, 28.04% of developers expect to apply the refactoring and generate private accessors methods (Figure 3.9(A)), and 58.88% of them expect to apply the refactoring and generate public accessors methods (Figure 3.9(B)).

In all questions in our survey, the options (A) and (B) are yielded by Eclipse, NetBeans, IntelliJ, or JRRT when receiving the input programs. We use all refactoring tools to apply the transformations of all questions. Some of it did not apply the refactoring or showed error messages during the application of the refactoring. This gives more evidence that there is

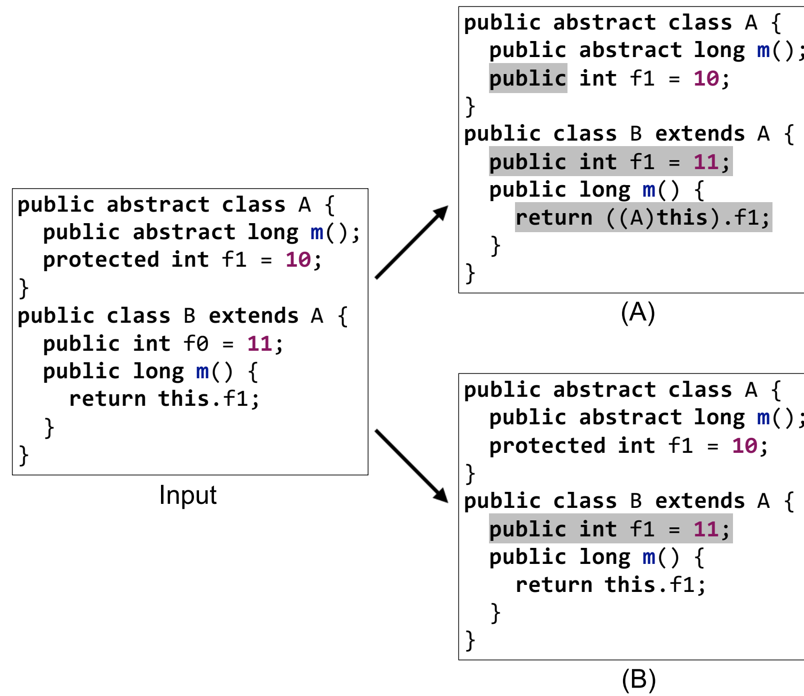


Figure 3.8: Rename Field refactoring.

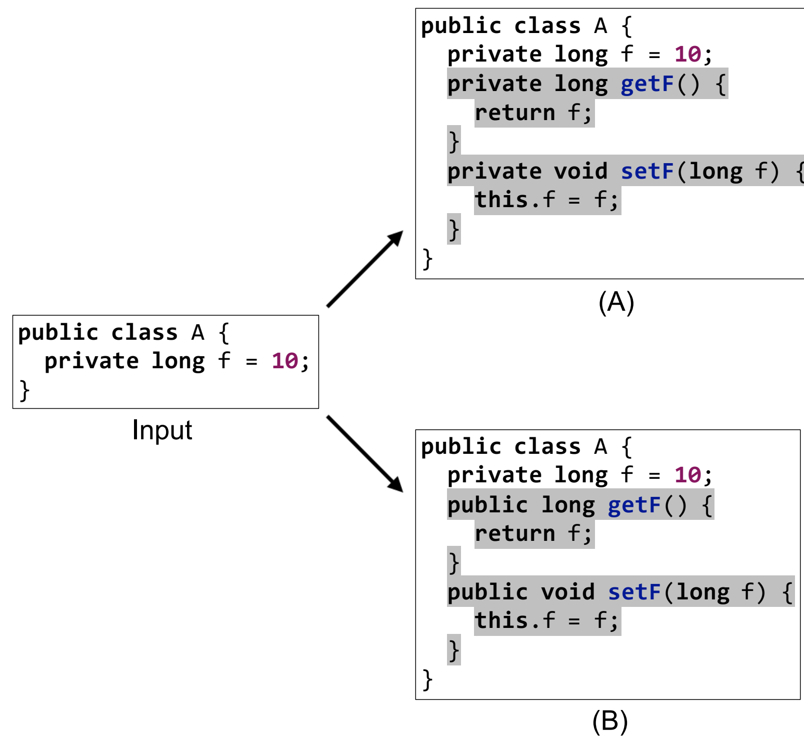


Figure 3.9: Encapsulate Field refactoring.

also no consensus in the context of tool developers about the meaning of refactoring names. Moreover, we only analyze the developers' answers that use a tool to apply a refactoring. Over 50% of the time, developers do not expect the output of the preferred tool when applying a refactoring.

3.1.5 Threats to Validity

Next, we identify some threats to validity of this evaluation.

Internal Validity. We provide questions with input and options to output yielded by one tool or IDE. Developers may paste the input code in their preferred IDE and apply the refactoring used in the question to answer. We have evidence they did not do this because the answers were varied. Moreover, we observe that developers affirmed use an IDE but prefer outputs yielded by another IDE.

External Validity. We provide questions with only one refactoring type applied to small Java programs with at most 10 LOC. The results of this survey can only be interpreted in the context of the considered refactoring types. We intend to survey more developers as future work.

3.2 STUDY II: Refactoring Implementations

In this section, we evaluate our approach to identify differences in refactorings implemented by tool developers using 27 refactoring implementations of Eclipse JDT, NetBeans, and JRRT of 10 refactoring types. First, we overview our approach (Section 3.2.1). Next, we explain the experiment definition (Section 3.2.2). Sections 3.2.4 and 3.2.5 present and discuss the results, respectively. Section 3.2.6 describes some threats to validity, and Section 3.2.7 summarizes the main findings.

3.2.1 Approach

In this section, we explain our approach to identify differences in refactorings implementations. Next, we overview our approach, and explain how we detect differences and categorize them.

Overview

The main steps of our approach are the following. First, it automatically generates programs as inputs for a refactoring using JDOLLY (Step 1). JDOLLY [19] is an automated and bounded-exhaustive Java program generator [19; 47; 48] based on Alloy, a formal specification language [49]. JDOLLY receives as input the refactoring type, a skip number to reduce the number of generated programs, and an Alloy specification, which includes specific characteristics and the scope of the programs. The refactoring is automatically applied to each generated program (Step 2). Finally, we use Differential Test [51] to identify differences in two refactoring implementations (Step 3). Figure 3.10 illustrates the main steps.

Detecting Differences in Refactoring Implementations

Our approach compares the application of two refactoring implementations of the same refactoring type using the same input program. We use an Abstract Syntax Tree (AST) differencing algorithm (*GumTree* [52]) to compare the outputs of the refactoring implementations. It has an improved detection of move actions over the previous work [53]. It performs a per-file comparison. The approach receives two compilable programs as input,

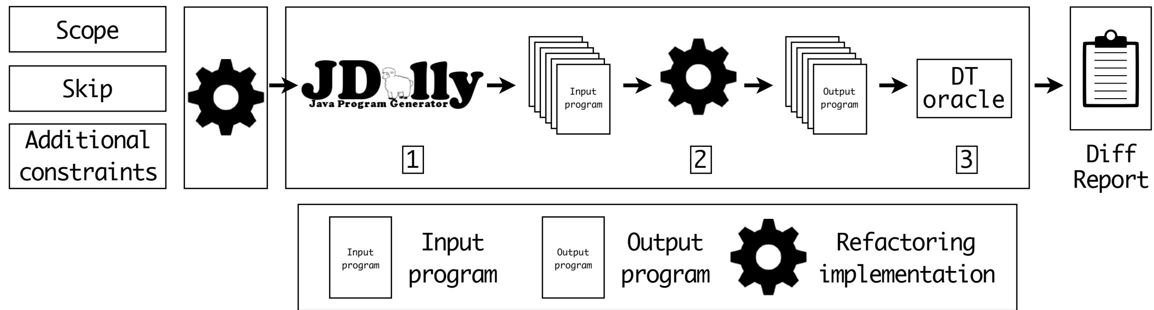


Figure 3.10: An approach to detect differences in refactoring implementations. First, JDOLLY automatically generates programs as inputs (Step 1). For each generated program, the refactoring implementation attempts to apply the transformation (Step 2). Finally, it identifies differences by using Differential Testing Oracle (Step 3).

and yields a list of actions for each class indicating the differences. For simplicity, the comparison considers the first program as the source, and the second as the target. The goal is to detect all possible actions that could be performed in the source to yield the target. For each file, *GumTree* yields a list of actions to change the AST of the source program to yield the AST of the target program. *GumTree* detects four kinds of actions: insert, delete, update, and move. The order of declarations (fields and methods) does not change the semantics in Java, therefore, we discard lists that contain only move actions. We manually inspect the differences to analyze whether there is a difference in both refactoring implementations, in one of them, or it is a false positive. Even *GumTree* having an improved detection of move actions, it may yield lists with other actions (for example, update actions) from a comparison composed only by move actions. We consider this scenario as a false positive.

For example, consider the program presented in Listing 3.1. Suppose we would like to apply the refactoring Push Down Method to `m`. First, we apply it using the refactoring implementation in Eclipse JDT 4.5. It yields the program presented in Listing 3.2. Then we apply the same refactoring using NetBeans 8.2, and it yields the program presented in Listing 3.3.

Listing 3.1: Input program to apply the Push Down Method refactoring.

```
1 package p1;
2 public class A {
3   public int m() {
```

```
4       return 1 ;
5     }
6   }
7
8   package p2;
9   import p1.*;
10  public class B extends A {}
11
12  package p1;
13  public class B extends A {}
```

Listing 3.2: Push Down Method refactoring application of Eclipse JDT.

```
1   package p1;
2   public class A {}
3
4   package p1;
5   public class B extends A {
6     public int m() {
7       return 1 ;
8     }
9   }
10
11  package p2;
12  import p1.*;
13  public class B extends A {
14    public int m() {
15      return 1 ;
16    }
17  }
```

Listing 3.3: Push Down Method refactoring application of NetBeans.

```
1   package p1;
2   public class A {}
3
4   package p1;
5   public class B extends A {
6     public int m() {
```

```
7       return 1 ;
8     }
9   }
10
11  package p2;
12  import p1.*;
13  public class B extends A {}
```

The approach performs a per-file analysis to the output programs presented in Listings 3.2 and 3.3. It yields a list of actions for each file. For the class `p2.B`, it yields the following list: *[DEL Modifier, DEL PrimitiveType, DEL SimpleName, DEL SimpleName, DEL MethodInvocation, DEL ReturnStatement, DEL Block, DEL MethodDeclaration]*. *DEL* denotes the delete action, and it is followed by an AST element. We manually analyze it. *GumTree* has a web-based view that helps us to graphically visualize the list of actions for each file. For example, Figure 3.11 shows the list of actions to the output programs presented in Listings 3.2 and 3.3. We identify that the NetBeans refactoring implementation pushes down a method to one subclass, while the Eclipse JDT refactoring implementation pushes down `m` to all subclasses. They perform different transformations. Since the resulting classes `p1.A` and `p1.B` are identical in Eclipse JDT and NetBeans implementations, *GumTree* yields an empty list of actions.

3.2.2 Definition

The goal of our experiment is to analyze the proposed approach to observe the extent of differences in refactoring implementations for the purpose of evaluating it with respect to the refactoring names in the perspective of tool developers. For this goal, we address the following research questions:

- Q1 What refactoring types have differences when comparing Eclipse and NetBeans refactoring implementations?

  We count the number of differences detected by our approach for each type of refactoring implementation.

- Q2 What refactoring types have differences when comparing Eclipse and JRRT refactoring implementations?

B.java

```
package p2;
import p1.*;
public class B extends A {
  public int m() {
    return 1 ;
  }
}
```

B.java

```
package p2;
import p1.*;
public class B extends A {}
```

Figure 3.11: The web-based diff view of *GumTree*.

We count the number of differences detected by our approach for each type of refactoring implementation.

- Q3 What refactoring types have differences when comparing NetBeans and JRRT refactoring implementations?

  We count the number of differences detected by our approach for each type of refactoring implementation.

3.2.3 Planning

In this section, we describe the subjects used in the experiment and its instrumentation. We ran the experiment on a Desktop computer 3.6 GHz core i7 with 16 GB RAM running Ubuntu 12.04 and JDK 1.7. We evaluated 27 refactoring implementations of Eclipse JDT 4.5, NetBeans 8.2, and JRRT (02/mar/13) of 10 refactoring types. We used a scope of two packages, three or four classes, up to four methods, and up to three fields to generate the programs for each refactoring type in JDOLLY [19]. We identified a number of compilation errors, behavioral changes, and overly strong conditions in refactoring implementations using a similar setup [47; 54]. It generates small Java programs containing abstract classes, abstract methods, and interfaces.

3.2.4 Results

Our approach generated 157,339 small Java programs with at most 10 LOC. It found differences in 3 refactoring types (Move Method, Push Down Field/Method) when comparing Eclipse and NetBeans refactoring implementations. In addition, it found differences in 5 refactoring types (Rename Field/Method, Move Method, Encapsulate Field, Push Down Method) when comparing NetBeans and JRRT refactoring implementations. Moreover, comparing Eclipse and JRRT refactoring implementations our approach found differences in 9 refactoring types (Rename Field/Method, Move Method, Add Parameter, Push Down Field/Method, Encapsulate Field, Pull Up Field/Method). Table 3.1 summarizes all differences found.

Refactoring	Programs	Differences		
		Eclipse x NetBeans	Eclipse x JRRT	NetBeans x JRRT
Pull Up Method	12,927	-	7,871	-
Pull Up Field	42,051	-	41	-
Push Down Method	3,462	449	554	449
Push Down Field	23,528	32	200	245
Add Parameter	13,319	3,823	5,698	9,170
Encapsulate Field	13,956	7,752	1,302	2,418
Rename Field	6,267	374	2,642	1,752
Rename Class	11,842	0	1,711	0
Rename Method	21,568	0	6,521	13,712
Move Method	8,419	4327	749	1,122

Table 3.1: Number of differences found by our approach. Programs = number of programs; Differences = number of transformations applied by both implementations that have different output programs; - = we could not evaluate the NetBeans implementations.

In the comparison between Eclipse and JRRT, our approach found five differences only in the Move Method refactoring. Besides that, our approach found four differences in Encapsulate Field refactoring when comparing NetBeans and JRRT. And, it found another four differences in the Push Down Method refactoring in the same pairwise comparison. On the other hand, our approach found four differences in the Push Down Field refactoring in the three tool comparisons. Overall, only 6.8% of all performed comparisons of the

refactoring implementations do not have differences. We manually classified the differences in categories presented in Table 3.2. Each difference can represent an indication that the refactoring name is not clear.

Difference Type	Refactoring	Comparison	#Diffs
It incorrectly implements of the SET method	Encapsulate Field	Eclipse x JRRT	1
The order of the abstract modifier is different	Encapsulate Field, Rename Field/Method, Add Parameter, Push Down Field/Method, Pull Up Method, Move Method	Eclipse x JRRT; NetBeans x JRRT	8
The GET or SET methods with a private accessibility	Encapsulate Field	NetBeans x JRRT	1
The GET or SET methods should have the same accessibility of the original field	Encapsulate Field	NetBeans x JRRT	1
It increases the field accessibility	Rename Field, Push Down Field, Pull Up Field	Eclipse x JRRT; NetBeans x JRRT	4
It decreases the method accessibility	Add Parameter	Eclipse x JRRT	1
The parameter is not added to the method	Add Parameter	Eclipse x JRRT	1
The field is not added to the target	Push Down Field	Eclipse x NetBeans	1
It does not update all field calls	Push Down Field	Eclipse x JRRT	1
It makes a class or method abstract	Pull Up Method	Eclipse x JRRT	1
It increases the method accessibility	Push Down Method, Move Method	NetBeans x JRRT	2
It does not remove the source class method	Pull Up Method	Eclipse x JRRT	1
It does not add the import to create the new object	Push Down Method	Eclipse x JRRT; NetBeans x JRRT	1
The method is not added to all subclasses	Push Down Method	Eclipse x NetBeans	1
It includes a ClassName.this when accessing an object	Move Method	Eclipse x JRRT	1
It includes a this when accessing a field	Move Method	Eclipse x JRRT; Eclipse x NetBeans	1
It creates a local variable	Move Method	Eclipse x JRRT	1

Table 3.2: Type of differences found by our approach. Difference Type = it specifies the difference found; Comparison = it specifies the comparison that showed the related difference; #Diffs = number of differences.

3.2.5 Discussion

In this section, we discuss the results of our evaluation concerning the differences detected, and JDOLLY.

Differences in Refactoring Implementations. We identify a number of differences between refactoring implementations. Only the Rename Class refactoring type does not have differences in all refactoring implementations. Our approach found only one difference in the Rename Method refactoring when comparing Eclipse and JRRT, and when comparing NetBeans and JRRT. Similarly, our approach found only two differences in the Rename Field refactoring in the same comparison.

For example, Listings 3.2 and 3.3 shows a difference in the Push Down Method refactoring in implementations of Eclipse JDT 4.5 and NetBeans 8.2 detected by our

approach. The input program presented in Listing 3.1 contains three classes: `p1.A`, `p2.B` that extends `A`, and `p1.B` that extends `A`. Class `A` declares method `m` returning 1. Applying the Push Down Method refactoring to move method `m` from class `A` to class `B`, we identify a difference. NetBeans (Listing 3.3) pushes down the method `m` to only one class `B`.

As another example, our approach detects whether the refactoring implementation updates a field/method access (Listings 3.4 and 3.5), or includes the *this* qualifier (Listings 3.6 and 3.7). We cannot detect differences whether some tool removes a file since our checker using *GumTree* performs a per-file analysis.

Listing 3.4: Rename Field refactoring application of NetBeans.

```
1  package p0;
2  abstract public class A {
3    public abstract long m();
4    protected int f = 10;
5  }
6
7  package p1;
8  import p0.*;
9  public class B extends A {
10   public long m() {
11     return this.f;
12   }
13   public int f = 11;
14 }
15
16 package p1;
17 public class C extends B {
18   protected int f = 12;
19 }
```

Listing 3.5: Rename Field refactoring application of JRRT.

```
1  package p0;
2  abstract public class A {
3    public int f = 10;
4    abstract public long m();
5  }
```

```
6
7  package p1;
8  import p0.*;
9  public class B extends A {
10   public int f = 11;
11   public long m() {
12     return ((A)this).f;
13   }
14 }
15
16 package p1;
17 public class C extends B {
18   protected int f = 12;
19 }
```

Listing 3.6: Move Method refactoring application of Eclipse JDT.

```
1  package p0;
2  abstract public class A {
3    protected abstract long m(int a);
4    private long m(long a) {
5      return 1;
6    }
7    public long m() {
8      return this.f.n(2);
9    }
10   public B f = null;
11 }
12
13 package p0;
14 import p1.*;
15 public class B implements C {
16     protected long n(int a){
17         return 0;
18       }
19 }
20
21 package p1;
```

```
22 public interface C {}
```

Listing 3.7: Move Method refactoring application of NetBeans.

```
1  package p0;
2  abstract public class A {
3    protected abstract long m(int a);
4    private long m(long a) {
5      return 1;
6    }
7    public long m() {
8      return f.n(2);
9    }
10   public B f = null;
11 }
12
13 package p0;
14 import p1.*;
15 public class B implements C {
16     protected long n(int a) {
17         return 0;
18     }
19 }
20
21 package p1;
22 public interface C {}
```

Previous works [55; 41] state that developers do not care whether refactorings change the observable behavior. We do not filter out behavioral changes in our current approach, but we can improve it by using SAFEREFACTOR [56] before Step 3 (Figure 3.10). SAFEREFACTOR automatically evaluates whether two versions of a program have the same behavior by automatically generating test cases only for the common methods impacted by the change. It identified a number of bugs in refactorings implementations of Eclipse JDT, JRRT, and NetBeans [19]. We use SAFEREFACTOR to analyze the programs that our approach identifies differences. SAFEREFACTOR identified 8 out of 28 programs with behavioral changes.

JDOLLY. To avoid state explosion, we adapted the scope for each refactoring type and added some constraints (optimizations). We added some constraints in the Java metamodel implemented in JDOLLY 3.0 to reduce this rate of uncompilable programs. After adding the new constraints, we have reached a rate of 90.9% of compilable programs generated by JDOLLY. In the Encapsulate Field and Add Parameter refactorings all generated programs compile. The lowest rate is 56.3% in the Push Down Method refactoring. However, it was one of the refactoring type that we found the most number of differences. The average rate of compilable programs in JDOLLY 1.0 was 68.8% [19].

Although the new constraints have reduced the number of Alloy instances, the Pull Up Field specification has more than a million instances using a scope of three classes and two methods, fields, and packages. This small scope coupled with a high number of Alloy instances indicates the expressiveness of JDOLLY. In the previous approach, JDOLLY generated at most 30,186 Alloy instances to generate useful programs [19]. After the addition of the new constructs (abstract classes, abstract methods, and interfaces), JDOLLY 3.0 had to deal with a number of Alloy instances 30 times higher than JDOLLY 1.0 for the same scope, which increased the cost to evaluate the refactoring implementations. Furthermore, we have to deal with memory leaks in the Eclipse JDT API. To alleviate these problems and reduce the costs to run the experiment, we choose a skip of 25 to generate programs in 5 refactoring types. In 4 out of 10 refactoring types, we used no skip.

Evaluating Other Refactoring Types. As a feasibility study, we evaluated the Inline Method refactoring implementations of Eclipse JDT, NetBeans, and JRRT. The goal is to analyze whether our approach can evaluate a refactoring type that applies a transformation inside a method body. Instead of using programs generated by JDOLLY in Step 1, we select 266 programs used in the test suites of Eclipse JDT, NetBeans, and JRRT, to evaluate the Inline Method refactoring implementation. The programs contain some Java constructions not considered in JDOLLY 3.0 such as: richer method bodies, generics, arrays, loops, among others. Then, we followed the same steps used in our approach. We found six differences (one in Eclipse JDT, two in NetBeans, and three in JRRT) using our approach related to (i) introducing the *this* modifier in a field access, (ii) deletion of all statements of the selected method, and (iii) not including an explicit cast.

For instance, consider the program presented in Listing 3.8. After applying the Inline Method refactoring of NetBeans 8.2 in the `add` method call, it yields the program presented in Listing 3.9. Notice that it removes the `add` method call in the resulting program.

Listing 3.8: Input program to apply the Inline Method refactoring.

```
1 public class Test {
2   static String[] f;
3   public static void main(String[] args) {
4     add((f = args).length, field.hashCode());
5   }
6   static int add(int x, int y) {
7     return y + x;
8   }
9 }
```

Listing 3.9: Inline Method refactoring application of NetBeans 8.2.

```
1 public class Test {
2   static String[] f;
3   public static void main(String[] args) {
4   }
5 }
```

Figure 3.12: The Inline Method refactoring implementation of NetBeans 8.2 removes some statements.

As another example, suppose a developer would like to inline the `Integer.parseInt` method. In NetBeans and JRRT, we cannot apply it. In Eclipse JDT, it depends on whether the developer has access to the `Integer.parseInt` source code. In case the developer has access, it is possible to inline it, and the tool does not remove `Integer.parseInt`.

3.2.6 Threats to Validity

Next, we identify some threats to validity of this evaluation.

Internal Validity. In fact, there is no consensus among refactoring tool developers about the refactoring names. Several refactoring implementations may introduce the same differences. The diversity of the generated programs may be related to the number of detected differences. The higher the diversity, more differences our approach may find. So, the scope, constraints, and skip used by JDOLLY control the number of generated programs, and consequently may also hide possible differences.

External Validity. We evaluated 27 refactoring implementations available in mainstream IDEs (Eclipse JDT and NetBeans) and in one academic tool (JRRT). A survey carried out by Murphy et al. [46] shows that Java developers commonly use the Pull Up refactoring. We evaluated the Pull Up Field and Pull Up Method refactorings. We plan to evaluate more refactoring types, and evaluate refactoring implementations of other IDEs, such as IntelliJ.

3.2.7 Answers to the Research Questions

Next, we answer our research questions.

- Q1 What refactoring types have differences when comparing Eclipse and NetBeans refactoring implementations?
  Our approach found 3 types of difference in 3 (Move Method, Push Down Field/Method) out of 10 refactoring types when comparing Eclipse and NetBeans refactoring implementations.

- Q2 What refactoring types have differences when comparing Eclipse and JRRT refactoring implementations?
  Our approach found 21 types of difference in 9 (Rename Field/Method, Move Method, Add Parameter, Push Down Field/Method, Encapsulate Field, Pull Up Field/Method) out of 10 refactoring types when comparing Eclipse and JRRT refactoring implementations.

- Q3 What refactoring types have differences when comparing NetBeans and JRRT refactoring implementations?
  Our approach found 7 types of difference in 5 (Rename Field/Method, Move Method,

Encapsulate Field, Push Down Method) out of 10 refactoring types when comparing NetBeans and JRRT refactoring implementations.

3.3 Study III: Refactoring Detection Tool

Refactoring detection algorithms have been crucial to a variety of applications: (i) empirical studies about the evolution of code, tests, and faults, (ii) tools for library API migration, (iii) improving the comprehension of changes and code reviews [11]. The tool *RMiner* [11] implements a novel technique, and has a improved precision and recall over the previous works [37; 38; 30]. *RMiner* currently supports the detection of 21 refactorings types (Extract Method, Inline Method, Rename Class/Method, Move Class/Field/Method, Pull Up Field/Method, Push Down Field/Method, Extract Superclass/Interface, Move and Rename Class, Extract and Move Method, Move Source Folder, Change Package (Move, Rename, Split, Merge), and Extract Variable).

In this section, we compare the meaning of the refactoring names used by *RMiner* to the meaning used by tool developers through 18 refactoring implementations of Eclipse JDT, NetBeans, and JRRT of 7 refactoring types. First, we overview our approach (Section 3.3.1). Next, we explain the experiment definition (Section 3.3.2). Sections 3.3.4 and 3.3.5 present and discuss the results, respectively. Section 3.3.7 summarizes the main findings.

3.3.1 Approach

In this section, we explain our approach to compare the meaning of the refactorings names used by a refactoring detection tool to the meaning of the refactoring names used by tool developers. We consider the researchers' perspective by using a tool (*RMiner* [11]) implemented by researchers to compare to refactorings implemented by refactoring tools through the analysis of the outputs yielded by Eclipse JDT, NetBeans, and JRRT.

Steps 1 and 2 are exactly the same of Figure 3.10. We reuse transformations that yield well-formed programs presented in Section 3.2. We consider 20,193 transformations of 7 refactoring types performed by Eclipse JDT, 36,723 transformations of 7 refactoring types performed by JRRT, and 19,387 transformations of 4 refactoring types performed by NetBeans. We do not consider all refactoring types evaluated in Section 3.2, since *RMiner* does not implement detection of three of them (Add Parameter, Encapsulate Field, and Rename Field). The only difference is in Step 3, in which We provide each pair of programs (input and refactored version) to *RMiner* performs the refactoring detection. Figure 3.13

illustrates the main steps.

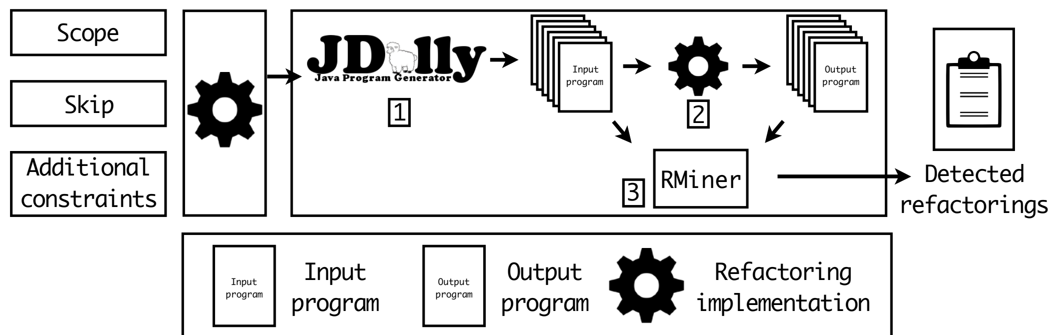


Figure 3.13: An approach to detect differences in refactoring implementations and refactoring detection tools. First, JDOLLY automatically generates programs as inputs (Step 1). For each generated program, the refactoring implementation attempts to apply the transformation (Step 2). Finally, we run *RMiner* in each transformation, and we check whether *RMiner* yields the same refactoring type applied by the refactoring implementation.

For example, consider Listings 3.10 and 3.11. Listing 3.11 is the result of the Push Down Method refactoring application performed by Eclipse JDT in the input program (Listing 3.10). The *RMiner* tool yields a list of detected refactorings when it receives the Listings 3.10 and 3.11 as input. We compare the results of this list with the refactoring applied by the tool to identify whether they are the same.

Listing 3.10: Input program to apply the Push Down Method refactoring.

```
1  package p0;
2  import p1.*;
3  public class B extends A {
4    public long n() {
5      return 0;
6    }
7    public long m() {
8      return q();
9    }
10 }
11
12 package p1;
13 import p0.*;
14 abstract public class A {
```

```
15    public abstract long n();
16    public long q() {
17      return new B().n();
18    }
19  }
20
21  package p1;
22  import p0.*;
23  public class C extends B {}
```

Listing 3.11: Result of the Push Down Method refactoring application performed by Eclipse JDT.

```
1   package p0;
2   import p1.*;
3   public class B extends A {
4     public long n() {
5       return 0;
6     }
7     public long m() {
8       return q();
9     }
10    public long q() {
11      return new B().n();
12    }
13  }
14
15  package p1;
16  import p0.*;
17  abstract public class A {
18    public abstract long n();
19  }
20
21  package p1;
22  import p0.*;
23  public class C extends B {}
```

We categorize the results into undetected (*RMiner* disagrees with the refactoring applied

by the tool), detected (*RMiner* identifies the same refactoring applied by the tool), and difference (*RMiner* yields a different refactoring type, or more than one refactoring type for the same pair of programs).

3.3.2 Definition

The goal of our experiment is to analyze the proposed approach to compare the meaning of the refactoring names used by researchers to the meaning used by tool developers for the purpose of evaluating it with respect to differences among *RMiner* and refactoring tools implementation. For this goal, we address the following research questions:

- Q1 What refactoring implementations have disagreements when comparing *RMiner* refactoring detection and Eclipse JDT refactoring implementations?
  We count the number of refactorings types with differences for each type of refactoring implementation.

- Q2 What refactoring implementations have disagreements when comparing *RMiner* refactoring detection and NetBeans refactoring implementations?
  We count the number of refactorings types with differences for each type of refactoring implementation.

- Q3 What refactoring implementations have disagreements when comparing *RMiner* refactoring detection and JRRT refactoring implementations?
  We count the number of refactorings types with differences for each type of refactoring implementation.

3.3.3 Planning

In this section, we describe the subjects used in the experiment and its instrumentation. We ran the experiment on a Desktop computer 3.6 GHz core i7 with 16 GB RAM running Ubuntu 12.04 and JDK 1.7. We use *RMiner* 1.0.0. We evaluated 18 refactoring implementations of popular IDEs (Eclipse JDT 4.5 and NetBeans 8.2), and one academic tool (JRRT (02/mar/13) [8]) of seven refactoring types from Section 3.2. We consider 20,193 transformations of 7 refactoring types performed by Eclipse JDT, 36,723 transformations

of 7 refactoring types performed by JRRT, and 19,387 transformations of 4 refactoring types performed by NetBeans. We consider a pair of programs as an input program (such as Listing 3.1) and a refactored version of this program yielded by the refactoring implementation (such as Listing 3.2) to use as input to *RMiner*.

3.3.4 Results

Our approach analyzes 76,303 transformations applied to small Java programs generated by JDOLLY with at most 10 LOC in 18 refactoring implementations of 7 refactorings types. Overall, *RMiner* detects 68% of all analyzed transformations applied by Eclipse JDT, NetBeans, and JRRT.

In some refactoring types, such as the Pull Up Method refactoring, *RMiner* disagrees with over 96% of the refactorings applied by Eclipse JDT and JRRT. For example, Eclipse JDT applies the Pull Up Method refactoring in the input program presented in Listing 3.12 and yields the outcome presented in Listing 3.13. *RMiner* disagrees with the application performed by Eclipse JDT in the programs presented in Listings 3.12 and 3.13. Despite this, *RMiner* agrees with all applications of 5 out of 18 refactoring implementations. It detects all applications of Pull Up Field, and Push Down Method/Field refactorings by Eclipse JDT, and all applications of Pull Up Field and Push Down Field refactorings by JRRT. Moreover, *RMiner* yields a different refactoring type, or more than one refactoring type in 1.9%, 2.8%, and 3.9% of the refactored programs from Eclipse JDT, NetBeans, and JRRT, respectively.

Listing 3.12: Input program to apply the Pull Up Method refactoring.

```
1  package p1;
2  import p0.*;
3  abstract public class A extends B {
4    protected abstract long m(long a);
5  }
6  package p0;
7
8  import p1.*;
9  public class B implements C {
10   protected long m(int a) {
11     return this.n(2);
```

```
12    }
13    private long n(int a) {
14      return 0;
15    }
16  }
17
18  package p1;
19  public interface C {}
```

Listing 3.13: Result of the Pull Up Method refactoring application performed by Eclipse JDT.

```
1   package p1;
2   import p0.*;
3   abstract public class A extends B {}
4
5   package p0;
6   import p1.*;
7   public abstract class B implements C {
8     protected long m(int a) {
9       return this.n(2);
10    }
11    private long n(int a) {
12      return 0;
13    }
14    protected abstract long m(long a);
15  }
16
17  package p1;
18  public interface C {}
```

In some cases, *RMiner* detected more refactoring types in a transformation applied by the tool. For example, 34% of the detection of the Move Method refactoring applied by Eclipse JDT yields other types of refactorings in *RMiner*, such as Push Down Method. The same occurs in 22% of detection of the Move Method refactoring applied by NetBeans and JRRT. For example, NetBeans applies the Move Method refactoring in the input program presented in Listing 3.14 and yields the outcome presented in Listing 3.15. *RMiner* yields

other refactoring type (Push Down Method refactoring) when performing the detection in this transformation. Table 3.3 summarizes the results.

Listing 3.14: Input program to apply the Move Method refactoring.

```
1  package p1;
2  public interface C {}
3
4  package p1;
5  import p0.*;
6  public class A implements C {
7    protected long m(int a) {
8      return 0;
9    }
10   public B f = null;
11 }
12
13 package p0;
14 import p1.*;
15 abstract public class B extends A {
16   protected abstract long n(long a);
17   public abstract long n(int a);
18   public long n() {
19     return m(2);
20   }
21 }
```

Listing 3.15: Result of the Move Method refactoring application performed by NetBeans.

```
1  package p1;
2  public interface C {}
3
4  package p1;
5  import p0.*;
6  public class A implements C {
7    public B f = null;
8  }
9
10 package p0;
```

```
11 import p1.*;
12 abstract public class B extends A {
13   protected abstract long n(long a);
14   public abstract long n(int a);
15   public long n() {
16     return m(2);
17   }
18   protected long m(int a) {
19     return 0;
20   }
21 }
```

Refactoring	Eclipse				NetBeans				JRRT			
	#Pairs	Undetected	Detected	Difference	#Pairs	Undetected	Detected	Difference	#Pairs	Undetected	Detected	Difference
Pull Up Method	8,761	8,497	264	0	-	-	-	-	8,414	8,150	264	0
Pull Up Field	315	0	315	0	-	-	-	-	315	0	315	0
Push Down Method	1,208	0	1,208	0	988	0	986	2	1,963	249	1,714	0
Push Down Field	229	0	229	0	1,526	111	1,415	0	276	0	276	0
Rename Class	2,069	690	1,379	0	-	-	-	-	7,994	1,952	5,135	907
Rename Method	6,512	240	6,272	0	14,416	1,104	13,312	0	15,368	1,032	14,336	0
Move Method	1,099	0	717	382	2,457	15	1,887	555	2,393	0	1,862	531
Total	**20,193**	**9,427**	**10,384**	**382**	**19,387**	**1,230**	**17,600**	**557**	**36,723**	**11,383**	**23,902**	**1,438**

Table 3.3: Summary of *RMiner* detection results. #Pairs = number of transformations; Undetected = *RMiner* does not yield the refactoring type applied by the refactoring implementation; Detected = *RMiner* identifies the refactoring applied by the refactoring implementation; Difference = *RMiner* yields a different refactoring type, or more than one refactoring type for the same pair of programs.

3.3.5 Discussion

In this section, we discuss the results of our evaluation in the context of the comparison among researchers (*RMiner*) and tool developers (outputs yielded by Eclipse JDT, NetBeans, and JRRT).

In some cases, *RMiner* agrees with the refactorings applied by the tools. For example, *RMiner* agrees with the Pull Up Field (Listings 3.16 and 3.17), and Push Down Field/Method refactorings applied by Eclipse JDT.

Listing 3.16: Input program to apply the Pull Up Field refactoring.

```
1  package p0;
2  public class A {
3    protected long n(long a) {
4      return 0;
5    }
6  }
7
8  package p0;
9  public class C extends A {
10   public long m() {
11     return this.n(2);
12   }
13   protected int f = 11;
14 }
15
16 package p1;
17 public class B {
18   public int f = 10;
19 }
```

Listing 3.17: Result of the Pull Up Field refactoring application performed by Eclipse JDT.

```
1  package p0;
2  public class A {
3    protected int f = 11;
4    protected long n(long a){
5      return 0;
6    }
7  }
8
9  package p0;
10 public class C extends A {
11   public long m() {
12     return this.n(2);
13   }
14 }
15
16 package p1;
```

```
17  public class B {
18    public int f = 10;
19  }
```

The approach disagrees with the refactoring application in three out of seven refactoring types applied by Eclipse JDT, in four out of seven refactoring types applied by JRRT, and three out of four refactoring types applied by NetBeans. Moreover, *RMiner* disagrees with over 7% of Push Down Field and Rename Method applied by NetBeans. The same occurs in over 24% of Rename Class refactoring application.

The approach identify differences in 5 out of 18 refactoring implementations. It detects differences in the Move Method refactoring implemented by Eclipse JDT, NetBeans, and JRRT, in the Push Down Method refactoring implemented by NetBeans, and in the Rename Class refactoring implemented by JRRT. Moreover, *RMiner* identifies more than one application of the same refactoring in one transformation, such as the application of Push Down Method/Field and Pull Up Field refactorings.

The differences found in the Move Method refactoring in all comparisons identify other refactoring type. For example, *RMiner* yields a list with Push Down Method and Pull Up Method refactorings for some pairs of transformations that NetBeans applied the Move Method refactoring.

3.3.6 Threats to Validity

Next, we identify some threats to validity of this evaluation.

Internal Validity. The differences found between *RMiner*, and Eclipse JDT, NetBeans, and JRRT, may not appear in other refactoring detection tools. Since *RMiner* has better accuracy than previous refactoring detection tools, differences may be find in other refactoring detection tools.

External Validity. We evaluated one transformation applied to one small Java program generated by JDOLLY. A similar scenario may happen in coarse-grained transformations applied to real programs. We considered refactoring implementations in popular IDEs (Eclipse JDT and NetBeans), but we cannot generalize our results to other tools and

languages. We noticed that some transformations applied by Eclipse JDT and NetBeans are similar to IntelliJ. So, we believe that similar results may also occur when considering IntelliJ.

3.3.7 Answers to the Research Questions

Next, we answer our research questions.

- Q1 What refactoring types have disagreements when comparing *RMiner* refactoring detection and Eclipse JDT refactoring implementations?

  The approach found 48.57% of disagreements in four out of seven refactoring implementations when comparing *RMiner* and Eclipse JDT refactoring implementations.

- Q2 What refactoring types have disagreements when comparing *RMiner* refactoring detection and NetBeans refactoring implementations?

  The approach found 9.22% of disagreements in four out of seven refactoring implementations when comparing *RMiner* and NetBeans refactoring implementations.

- Q3 What refactoring types have disagreements when comparing *RMiner* refactoring detection and JRRT refactoring implementations?

  The approach found 35% of disagreements in five out of seven refactoring implementations when comparing *RMiner* and JRRT refactoring implementations.

Chapter 4

Conclusions

In this work, we conduct a mixed-method study from different perspectives to better understand the meaning of the refactoring names. The first study considers the developers' perspective by conducting a survey with 107 developers of popular Java projects on GitHub to better understand the meaning of the refactoring names used by them in practice. Since most developers expect the refactoring output based on their experience, there is no consensus in any of the questions in our survey. This scenario may be even worse when considering coarse-grained refactorings applied to larger Java programs. A number of developers use IDEs to apply refactorings. However, over 50% of the time, the IDEs used by developers yield an output that is different from what they expect.

In the second study, we investigate to what extent refactoring implementations have the same understanding of the meaning associated with the refactoring names. We evaluate it automatically generating small Java programs as inputs using JDOLLY. Our approach attempts to apply a refactoring type using the refactoring implementations of mainstreams IDEs (Eclipse and NetBeans) and an academic tool (JRRT). We considered 10 types of refactorings in 27 refactoring implementations. Overall, only 6.8% of the refactoring applications do not have differences. These results give evidences that refactoring names have different meanings in the context of refactoring implementation' developers.

In the third study, we compare the meaning of the refactoring names used in *RMiner* to refactoring implementations implemented by Eclipse, NetBeans, and JRRT. We analyze 18 refactoring implementations of 7 refactoring types. We provide an input program and a refactored program yielded by Eclipse, NetBeans, and JRRT in Section 3.2. *RMiner*

does not detect 46% of the refactorings applied by Eclipse JDT, 30% of JRRT, and 6% of NetBeans in 7 refactoring types. Our results show evidences of misunderstandings of the current refactoring names among developers, and tool developers. The misunderstandings explained in our work may be a starting point to improve them. Moreover, researchers used refactoring detection tools to empirically study [20; 21; 22; 23; 24; 25; 26] software evolution, and to support other software engineering tasks, such as library adaptation [27; 28; 29; 30], software merging [31], code completion [32; 33], and code review [34; 35; 36]. Since we found differences in *RMiner*, and Eclipse and NetBeans in Section 3.3, some refactoring implementations of Eclipse and NetBeans are different (Section 3.2), and developers have different opinions about transformations applied by popular refactoring implementations (Section 3.1), the previous results using refactoring detection tools should be revisited to check whether the meaning of the refactoring names used in each work is appropriate.

Our work shows differences between refactoring implementations and refactoring detection, and differences in understanding of refactoring types by developers. Some refactoring definitions use natural language to describe scenarios and application of it. This may be one cause of misunderstanding of the refactoring types presented in our study. Using formal methods to formalize all refactorings is a challenge considering all language statements that exist, such as the statements in the Java language. Some works formalize some refactoring types. This is a step to minimize the differences found by our work. Our approach presented in Section 3.2 can also help developers to improve refactoring implementations. They can compare the application of one refactoring implementation with other and merge the results to improve the actual refactoring implementations. Approaches to designing software, such as Design by Contract, may help to improve existing refactoring implementations.

Developers depend on the preferred IDE to predict a refactoring application but they can disagree with the application of it. This disagreement can generate rework because the applied transformation can be modified. This modification may impact refactoring detection tools. Minimizing the differences found between the refactorings tools presented in the second study may contribute to the evolution of refactorings detection tools. In addition, the developers that use these tools can change their understanding and agree with the tools

by minimizing misunderstandings.

4.1 Related Work

Opdyke and Johnson [2; 4] coined the refactoring term describing the process and identifying common refactorings. Roberts [43] automates the basic refactorings proposed by Opdyke. Later, Tokuda and Batory [57] demonstrate that the preconditions proposed by Opdyke are not sufficient to guarantee behavior preservation after applying transformations. Moreover, proving refactorings with respect to a formal semantics considering all language constructs constitutes a challenge [58]. In our work, we performed an analysis of the different interpretations inferred from refactorings presented in the literature.

Murphy-Hill et al. [18] find that the names of refactorings assigned by the refactoring tools are a distraction to the developer because it can vary from one environment to another. For example, Fowler's Introduce Explaining Variable [5] is called Extract Local Variable in Eclipse. Moreover, refactoring names differ between Eclipse and IntelliJ. This confuses the developer. For example, Generify in IntelliJ appears as Infer Generic Type Arguments in Eclipse, and Introduce Field in IntelliJ appears as Convert Local Variable to Field in Eclipse. In our work, we conduct a mixed-method study, and find more evidences of misunderstandings in the refactoring names.

Vakilian et al. [17] study 26 developers working in their natural settings on their code for a total of 1,268 programming hours over three months to understand how they interact with automated refactorings. They identify factors that affect the appropriate and inappropriate uses of automated refactorings. For example, they show that disuse of automated refactorings occurs when a programmer performs a refactoring manually even though the IDE supports it. In addition, more than half of interviewees 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 interviewees could not describe the transformation automated by some refactoring, and did not use some automated refactorings because of their unpredictability. We find that even refactorings applied to small programs may also lead to misunderstandings.

Kim et al. [39] perform a field study of refactoring benefits and challenges at Microsoft through three complementary study methods: a survey, semi-structured interviews with professional software engineers, and quantitative analysis of version history data. With an exception of the Rename refactoring, more than a half of the participants said that they apply those refactorings manually. In our study, most developers (75.70%) use IDEs to apply refactorings.

Murphy-Hill et al. [40] present an analysis of four sets of data that provides new insight into how developers refactor in practice. Refactoring implementations themselves are underused, particularly when we consider refactorings that have a method-level granularity or above. We find that a number of developers use IDEs to apply refactorings. However, the output yielded by the preferred IDE is different from what they want.

Tempero et al. [41] conduct a survey with 3,785 developers to see the barriers of applying refactorings. They found that the decision of whether or not to refactor was due to non-design considerations. They mentioned inadequate tool support as a reason for not refactoring. In our study, developers use IDEs to apply refactorings even not expecting the same outcome yielded by the preferred IDE.

Daniel et al. [59] propose an approach for automated testing refactoring engines. The technique is based on ASTGEN, a Java program generator, and a set of programmatic oracles. To evaluate the refactoring implementations, they implemented six oracles that evaluate the output of each transformation. They use the oracles DT, and Inverse Transformations, to identify differences in refactoring implementations. The Inverse oracle checks whether after 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 find three differences using Differential Testing and Inverse oracles in 2 refactoring implementations of Eclipse and NetBeans of the Encapsulate Field refactoring. We compare the output of 27 refactoring implementations and find 21 differences in 9 out of 10 refactoring implementations. In a previous work, Soares et al. [19] show that JDOLLY generates more interesting programs than ASTGEN.

Steimann and Thies [12] identify that mainstreams IDEs such as Eclipse, NetBeans, and

IntelliJ are flawed when it comes to maintaining accessibility. They identify scenarios where the application of existing refactorings such as Pull Up Members causes unexpected changes to program behavior. In our study, we observe an access modification in the application of the Pull Up Field refactoring performed by Eclipse but developers (45.79%) agreed with this modification.

Schäfer et al. [8] present a number of Java refactoring implementations. They translated a Java program to an enriched language that's easier to specify and check preconditions, and apply the transformation. They aim to improve correctness and applicability of the Eclipse refactoring implementations. In our work, we compare JRRT to mainstreams IDEs to identify differences in refactoring implementations. Our approach used in study II (Section 3.2) found 21 differences in 9 refactoring types when compared to refactoring implementations of Eclipse. Moreover, our approach used in study III (Section 3.3) found differences in 3.9% of the refactoring applications.

Jagannath et al. [60] presented the STG technique to reduce the costs of bounded-exhaustive testing by skipping some test inputs. They randomly select a skip up to 20 after generating each program. They evaluated it using ASTGEN and found that the technique took some seconds to find the first failure related to compilation error or engine crash in the refactoring implementations using STG. Different from them we use skips to identify differences in refactoring implementations in the second study (Section 3.2). Also, we use a fixed skip that is set by the user while they use a random skip. Moreover, we can execute using a different skip to find some missed differences.

Later, Gligoric et al. [61] propose UDITA, a Java-like language that extends ASTGEN allowing users to describe properties in UDITA using any desired mix of filtering and generating style in opposed to ASTGEN that uses a purely generating style. UDITA evolved ASTGEN to be more expressive and easier to use, usually resulting in faster program generation as well. They found four new bugs related to compilation errors in Eclipse in a few minutes. However, the technique requires substantial manual effort for writing test generators [62] since they are specified in a Java-like language. Soares et al. [19] find that UDITA does not generate some programs that JDOLLY generates using the same scope. We use a new version of JDOLLY in this work [54]. It generates programs considering more Java constructs.

Related approaches [19; 54; 48] proposed a technique to test refactoring engines by detecting bugs related to compilation errors, behavioral changes, and overly strong preconditions. It is based on JDOLLY and a set of automated oracles, such as SAFEREFACTOR [56] to identify behavioral changes and Differential Testing and Disabling Preconditions to identify overly strong preconditions. As opposed to ASTGEN and UDITA that use a Java-like language, JDOLLY only needs to declaratively specify the structures of the programs. In this work, we extend them [19; 54; 48] to identify differences between refactoring implementations and improve the expressiveness of JDOLLY by generating programs considering more Java constructs.

Cedrim et al. [63] analyze both the positive and negative impact of refactoring changes on the density of smells. They analyze if refactoring reduces smells, and if and to what extent specific refactoring types are often related to the introduction of new smells. They find that refactorings often touches smelly elements, but they are neutral suggesting that developers need more guidance to remove a code smell. Moreover, they find that 33.3% of the refactorings are related to the introduction of new smells. They use *RMiner* to support the detection of refactoring types. We compare the meaning of the refactoring names used in *RMiner* to refactoring implementations performed by Eclipse JDT, NetBeans, and JRRT (Section 3.3). *RMiner* disagrees with 48.57% of the refactorings applied by Eclipse JDT, 35% of JRRT, and 9.22% of NetBeans in 7 refactoring types. We found differences related to creates a local variable, does not update field calls, accessibility changes, and others. Our results show differences in the opinion of the researchers and tool developers about the refactoring names. This may influence the results of the work, and in other works that use tools to detect refactorings [37; 38; 30]. The tools may have detected more or less refactoring, or detect other refactoring types.

Medeiros et al. [64] propose a mixed-method evaluation of their refactoring catalog from the previous study [65] of the C language with ifdefs. The catalog contains 14 refactoring types to resolve undisciplined directives. They evaluate the refactoring types regarding the opinion of developers, number of application possibilities in practice, and behavior preservation. The results show that developers prefer the refactored version instead of the original program. We can use the approach of the second study (Section 3.2) to compare different refactoring implementations of the proposed catalog. We need to review

the refactoring names carefully to avoid misunderstandings.

Mongiovi et al. [66] propose a tool to analyze refactoring applications and generate test cases only for impacted methods. They evaluate SAFEREFACTORIMPACT by the comparison to SAFEREFACTOR with respect to correctness in identifying the behavioral changes, time to analyze a transformation, number of methods considered for test generation, coverage of methods impacted, and relevant tests. The results show that SAFEREFACTORIMPACT detects more behavioral changes, has better results when analyzing larger programs, and is faster than SAFEREFACTOR to analyze small programs. We can use SAFEREFACTORIMPACT to evaluate the differences found to identify behavioral changes.

4.2 Future Work

As future work, we intend to evaluate more tools (such as IntelliJ), and more refactoring types to detect and study more differences. We intend to evaluate more refactoring types using the programs available in the test suites of Eclipse JDT, NetBeans, IntelliJ, and JRRT. Moreover, we aim to survey more developers by adding more refactoring types to identify misunderstandings.

Bibliography

[1] Meir Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[2] William Opdyke and Ralph Johnson. Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems. In *Proceedings of the Symposium Object-Oriented Programming Emphasizing Practical Applications*, SOOPPA, pages 274–282, 1990.

[3] E. Burton Swanson. The Dimensions of Maintenance. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 492–497, 1976.

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

[5] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

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

[7] Joshua Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004.

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

[9] Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for Reentrancy. In *Proceedings of the Joint Meeting of the European Software Engineering Conference*

and the Symposium on The Foundations of Software Engineering, ESEC/FSE, pages 173–182, 2009.

[10] Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring Using Type Constraints. *ACM Transactions on Programming Languages and Systems*, 33(3):9:1–9:47, 2011.

[11] 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.

[12] 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.

[13] Eclipse.org. Eclipse Project. http://www.eclipse.org, 2018.

[14] Oracle. Netbeans IDE. http://www.netbeans.org, 2018.

[15] JetBrains. IntelliJ IDEA. https://www.jetbrains.com/idea/, 2018.

[16] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Company, Inc., 2000.

[17] 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.

[18] 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.

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

[20] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. An Automatic Approach to Identify Class Evolution Discontinuities. In *Proceedings of the International Workshop on Principles of Software Evolution*, IWPSE, pages 31–40, 2004.

[21] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. When Does a Refactoring Induce Bugs? An Empirical Study. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM, pages 104–113, 2012.

[22] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015.

[23] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An Empirical Investigation into the Role of API-level Refactorings During Software Evolution. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 151–160, 2011.

[24] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. An Exploratory Study on the Relationship Between Changes and Refactoring. In *Proceedings of the International Conference on Program Comprehension*, ICPC, pages 176–185, 2017.

[25] Napol Rachatasumrit and Miryung Kim. An empirical investigation into the impact of refactoring on regression testing. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM, pages 357–366, 2012.

[26] Peter Weißgerber and Stephan Diehl. Are Refactorings Less Error-prone Than Other Changes? In *Proceedings of the International Workshop on Mining Software Repositories*, MSR, pages 112–118, 2006.

[27] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring Support for Class Library Migration. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 265–279, 2005.

[28] Johannes Henkel and Amer Diwan. CatchUp!: Capturing and Replaying Refactorings

to Support API Evolution. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 274–283, 2005.

[29] Zhenchang Xing and Eleni Stroulia. API-Evolution Support with Diff-CatchUp. *IEEE Transactions on Software Engineering*, 33(12):818–836, 2007.

[30] 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.

[31] Danny Dig, Kashif Manzoor, Ralph E. Johnson, and Tien N. Nguyen. Effective Software Merging in the Presence of Object-Oriented Refactorings. *IEEE Transactions on Software Engineering*, 34(3):321–335, 2008.

[32] Stephen Foster, William Griswold, and Sorin Lerner. WitchDoctor: IDE support for real-time auto-completion of refactorings. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 222–232, 2012.

[33] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. Reconciling Manual and Automatic Refactoring. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 211–221, 2012.

[34] Everton Alves, Myoungkyu Song, and Miryung Kim. RefDistiller: A Refactoring Aware Code Review Tool For Inspecting Manual Refactoring Edits. In *Proceedings of the International Symposium on Foundations of Software Engineering*, FSE, pages 751–754, 2014.

[35] Xi Ge, Saurabh Sarkar, and Emerson Murphy-Hill. Towards Refactoring-aware Code Review. In *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE, pages 99–102, 2014.

[36] Xi Ge, Saurabh Sarkar, Jim Witschey, and Emerson Murphy-Hill. Refactoring-Aware Code Review. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*, VL/HCC, pages 71–79, 2017.

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

[38] Danilo Silva and Marco Tulio Valente. RefDiff: Detecting Refactorings in Version Histories. In *Proceedings of the International Conference on Mining Software Repositories*, MSR, pages 269–279, 2017.

[39] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. An Empirical Study of Refactoring Challenges and Benefits at Microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, 2014.

[40] 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.

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

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

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

[44] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[45] Robert Fuhrer, Adam Kieżun, and Markus Keller. Refactoring in the Eclipse JDT: Past, Present, and Future. In *Workshop on Refactoring Tools*, pages 30–31. Springer-Verlag, 2007.

[46] Gail Murphy, Mik Kersten, and Leah Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[47] 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.

[48] Gustavo Soares, Melina Mongiovi, and Rohit Gheyi. Identifying overly strong conditions in refactoring implementations. In *Proceedings of the International Conference on Software Maintenance*, ICSM, pages 173–182, 2011.

[49] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.

[50] Daniel Jackson, Ian Schechter, and Hya Shlyahter. Alcoa: The Alloy Constraint Analyzer. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 730–733, 2000.

[51] William McKeeman. Differential Testing for Software. *Digital Technical Journal*, 10(1):100–107, 1998.

[52] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering*, ASE, pages 313–324, 2014.

[53] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[54] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Márcio Ribeiro, Paulo Borba, and Leopoldo Teixeira. Detecting Overly Strong Preconditions in Refactoring Engines. *IEEE Transactions on Software Engineering*, 44(5):429–452, 2018.

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

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

[57] Lance Tokuda and Don Batory. Evolving Object-Oriented Designs with Refactorings. In *Proceedings of the International Conference on Automated Software Engineering*, ASE, pages 89–120, 2001.

[58] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Challenge Proposal: Verification of Refactorings. In *Proceedings of the International Conference on Programming Languages Meets Program Verification*, PLPV, pages 67–72, 2008.

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

[60] Vilas Jagannath, Yun Young Lee, Brett Daniel, and Darko Marinov. Reducing the costs of bounded-exhaustive testing. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, FASE, pages 171–185, 2009.

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

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

[63] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. Understanding the Impact of Refactoring on Smells: A Longitudinal Study of 23 Software Projects. In *Proceedings of the International Symposium on Foundations of Software Engineering*, FSE, pages 465–475, 2017.

[64] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Transactions on Software Engineering*, 44(5):453–469, 2018.

[65] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, and Baldoino Fonseca. A Catalogue of Refactorings to Remove Incomplete Annotations. *Journal of Universal Computer Science*, 20(5):746–771, 2014.

[66] 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.