Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Refactoring and What Else? An Exploratory Study on *Floss Refactoring*

Jaziel Silva Moreira

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: Refactoramento

Nome do Orientador

Wilkerson de Lucena Andrade

Everton Leandro Galdino Alves

Campina Grande, Paraíba, Brasil

S729e Moreira, Jaziel Silva.

Refatoring and What Else?: an exploratory study on *floss refatoring* / Jaziel Silva Moreira. – Campina Grande, 2019.

87 f. : il.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2019.

"Orientação: Prof. Dr. Wilkerson de Lucena Andrade, Prof. Dr. Everton Leandro Galdino Alves".

Referências.

1. *Floss-refatorings*. 2. Edições Extras. 3. Repositórios de Software. 4. Estudo Empírico. I. Andrade, Wilkerson de Lucena. II. Alves, Everton Leandro Galdino. III. Título.

CDU 004.41(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA MARIA ANTONIA DE SOUSA CRB 15/398

"REFACTORING AND WHAT ELSE? AN EXPLORATORY STUDY ON FLOSS REFACTORING"

JAZIEL SILVA MOREIRA

DISSERTAÇÃO APROVADA EM 25/07/2019

CAMPINA GRANDE – PB

Resumo

Refatoramento é uma atividade que visa melhorar a qualidade do *design* e a legibilidade do código de um sistema sem alterar seu comportamento externo. Refatoramentos são responsáveis por quase 30% de todas as edições de um software. Embora os refatoramentos sejam edições que preservam o comportamento, estudos mostram que desenvolvedores tendem a aplicar mudanças comportamentais intencionais ao lado de refatoramentos (*floss-refactoring). Floss-refactorings* são conhecidos por serem propensos a erros e requererem revisão de código. No entanto, pouco foi feito para entender como os desenvolvedores relacionam refatoramentos às edições extras. Deste modo, neste trabalho propomos uma estratégia para a extração de dados de *floss-refactorings*, que pode ser usada para extrair informações detalhadas sobre os refatoramentos e edições extras aplicadas ao longo do histórico de versões do repositório de um projeto Java. Além disso, para entender melhor como os desenvolvedores realizam *floss-refactorings* no mundo real, conduzimos uma investigação empírica para descobrir como as edições extras são aplicadas com base nos refatoramentos encontrados em um *commit*. Nós analisamos os *commits* de 45 repositórios em que as edições de refatoramentos foram aplicadas juntamente com edições extras. Nossos resultados mostraram que, dependendo do refatoramento realizado, há mudanças na probabilidade de algumas edições extras serem aplicadas. Por exemplo, a introdução de novos métodos é mais comum quando um *Extract Method* ou um *Rename Method* é executado. Outras edições, como a remoção de um método, a introdução de um novo atributo em uma classe ou mesmo edições específicas dentro de métodos, também apresentaram mudanças significativas em sua probabilidade. Além disso, 14,4% das edições extras foram realizadas dentro de entidades refatoradas, das quais as entidades alteradas mais comuns foram *Invocação de Método*, *If Statement*, *Declaração da Variável*, *Atribuição* e *Statement de Retorno*. No entanto, a probabilidade de cada tipo de entidade específica a ser alterada varia dependendo do refatoramento executado. Em geral, os padrões de relacionamento entre refatoramentos e edições extras encontrados neste trabalho podem ser usados para guiar a revisão de código, ajudar os desenvolvedores a evitar a introdução de faltas relacionadas a *floss-refactorings*, e orientar outras pesquisas relacionadas a refatoramentos.

Palavras-chave: *Floss-refactorings*, Edições extras, Repositórios de software, Estudo empírico.

Abstract

Refactoring is an activity that aims at improving design quality and code readability of a system without changing its external behavior. It accounts for nearly 30% of all edits in a software life cycle. Although refactorings are behavior-preserving edits, studies show that developers tend to apply intentional behavioral change edits alongside refactorings (*floss-refactoring*). Floss-refactorings are known to be error-prone and require code revision. However, Little has been done to understand how developers relate refactorings to extra edits. Thus, in this work we propose a strategy for floss-refactoring data extraction, which can be used for extracting detailed information about the refactorings and extra edits applied throughout the versioning history of a Java project repository. In addition, to better understand how developers perform floss refactoring in real world, we conducted an empirical investigation to find out how extra edits are applied based on the refactoring found in a commit. We mined repositories of 45 open-source projects and analyzed all commits where refactoring edits were performed along with non-refactoring edits. Our results showed that, depending on the refactoring performed, there are changes on the likelihood of some extra edits to be applied. For instance, the introduction of new methods is more common when an *Extract Method* or a *Rename Method* is performed. Other edits, such as the removal of a method, introduction of a new attribute to a class, or even specific inner method edits, also presented significant change on its likelihood. Moreover, 14,4% of the extra edits were performed inside refactored entities, from which, the most common entities changed were *Method Invocation*, *If Statement*, *Variable Declaration*, *Assignment*, and *Return Statement*. However the likelihood of each specific entity type to be changed varies depending on the refactoring performed. Overall, the relationship patterns between refactorings and extra edits found in this work can be used to guide code revision, help developers to avoid faults related to floss refactoring, and to guide other refactoring-related researches.

Keywords: Floss Refactoring, Extra Edits, Software Repositories, Empirical Study.

Agradecimentos

Agradeço primeiramente a Deus, pelo dom da vida e a benção da saúde;

Aos meus pais, José Nunes Moreira e Marineuza Silva Moreira, bem como minha irmã, Jardeane Silva Moreira, pelo grande amor e suporte que me ofereceram durante toda minha vida, nunca medindo esforços em me ajudar no meu desenvolvimento acadêmico;

Aos meus orientadores, Wilkerson de Lucena Andrade e Everton Leandro Galdino Alves, que me ensinaram tudo que era necessário para meu trabalho, sempre com paciência, dedicação, empenho e respeito. Sem eles, esse trabalho não seria possível;

A Damiana Vicente e familia, que me acolheram durante esta etapa, pelo apoio, estando sempre dispostos a ajudar;

A todos os membros do SPLab, em especial aos meus companheiros que passaram pela sala 104, Raul Andrade, Raphael Pontes, Isabelle Lima, Guilherme Gadella e Marcos Nascimento, pelas trocas de conhecimento, apoio, e, especialmente, por sua amizade;

Aos professores e funcionários da COPIN e do DSC;

Ao Conselho Nacional de Desenvolvimento Científico e Tecnologico (CNPq) pelo apoio e suporte financeiro fornecido a este trabalho.

Contents

Lista de Símbolos

XP - *Extreme Programming*

TDD - *Test-driven development*

IDE - *Integrated Development Environment*

API - *Application Programming Interface*

TF-IDF - *Term Frequency-InverseDocument Frequency*

GUI - *Graphical User Interface*

AST - *Abstract Syntax Tree*

LOC - *Lines of Code*

OR - *Odds Ratios*

List of Figures

List of Tables

Lista de Códigos Fonte

Chapter 1

Introduction

A software should provide the functionality and performance required by its users and also be easy to maintain [51]. In this sense, the main purpose of Software Engineering is to support professional software development, helping developers throughout all aspects of software production. The main activities of software engineering, present in most of the processes, are specification, development, validation, and evolution of the software [51].

Software evolution plays an important role during software development, and continues to be important through the whole software lifetime. The system requirements tend to be adapted according to the client's needs, as well as its environment. Therefore, the development continues in order to attend these changes, even after the system delivery. Moreover, it may be necessary to fix bugs that appear during its execution.

The modifications made in a system, after it is released, are called software evolution. According to Sommerville [51], there are three types of software evolution: Bug Fix, Environmental Adaptation and Feature Introduction. Bug Fix consists in fixing a bug that passed unnoticed through the system validation. Environmental Adaptation is necessary when the client needs to use the system in a different environment from the one it was first intended. For instance, adapt an Android app to run on iOS. Feature Introduction consists in adding new functionalities required by the client. In practice, these three types often happen together. For instance, a new environment might allow the introduction of a new feature that was not possible before.

During software evolution, it is common for the code to deteriorate, which might turn it challenging to maintain, and error-prone. Long methods and duplicate code fragments are examples of bad smells that might make the code difficult to maintain (e.g., hard to understand and reuse). Preventive maintenance is another kind of maintenance that intends to prevent bugs from being introduced. That is, to improve the system's future maintainability by changing its structure without adding/removing functionalities [7].

In this context, *refactoring* is an activity that aims at removing bad smells by updating the structure of a software without changing its external behavior [15]. It is often applied to improve aspects such as design quality, code readability, etc. Studies show that refactoring can avoid code deterioration [26; 31]; reduce energy consumption [42]; and maintenance costs [23], specially in agile projects where requirement changes are frequent during development. Due to its positive impact on the development cycle, refactorings are widely applied, accounting for approximately 30% of all performed edits [47].

1.1 Problem Definition

Despite the intention of improving software quality, the refactoring application often leads to error-proneness. For instance, when a developer applies a *Rename Method* refactoring without proper care, he might end-up overriding a method from a superclass. In order to prevent chaos during refactoring activities, Fowler [15] assembled a catalog with the mechanics of 72 refactorings types. However, several studies have found there is a strong correlation between the timing and location of refactorings and bug fixes [2; 24]. A field study run with Microsoft developers [25] shows that 77% of the survey participants perceived that refactoring comes with a risk of introducing subtle bugs and functionality regression. The risk of bug introduction is even higher, since most refactorings are manually performed [1; 32; 33; 55]. Murphy et al. [32] found that about 90% of refactoring edits are done manually. Expert developers prefer manual to automated refactoring [34]. Moreover, even refactoring tools are not free of bug introduction [30; 49].

Even simple refactorings, such as an *Extract Method* can be tricky. An *Extract Method* refactoring encompasses the extraction of a code fragment into a separate method. It is known to be one of the most widely performed refactorings since it can be used for fixing a variety of design problems [52]. Fowler [15] proposes the mechanics for a systematic

Extract Method. It comprises nine steps and combines microcode edits with several runs of an existing test suite. However, practical circumstances (e.g., time shortage) often prevents the use of Fowler's mechanics every time an *Extract Method* is needed. Therefore, subtle faults can be introduced when an *Extract Method* is intended. For instance, the newly created method may add/break override/overload contract. Bavota et al. [2] found that the *Extract Method* refactoring often induces bug fixes.

On top of all, studies have shown that, in practice, developers often interweave refactoring and non-refactoring edits - *floss-refactoring* [33; 45; 50]. By combining behavior changing edits alongside refactorings (e.g., an edit for improving readability and the introduction of new features), a developer might increase the likelihood of introducing faults. For instance, Silva et al. [45] stated that developers often perform *Extract Method*s for avoiding code replication. After refactoring, the newly created method is then called in several places. However, suppose a developer, aiming at attending a requirement from one of the callers, introduces a new condition checking in the newly added method. If this extra edit is performed without the proper impact analysis, it may negatively impact other callers.

Refactoring-aware code revision is an emerging topic that has gained notoriety due to its practical benefits [1; 10; 18]. Due to its complexity, and possible impact, developers should review their edits considering the refactorings performed. The goal is to both detecting possible faults, and/or confirming their intentions. This need is even more evident when floss-refactoring, since different concerns may be mixed in a single commit. Alves et al. [1] discussed two types of refactoring anomalies that require revision: *missing* and *extra edits*. The latter are the edits that go beyond a pure refactoring transformation. Therefore, even intentional extra edits require special attention. Recent works [1; 9; 18] highlight this issue and propose new tools for helping developers to review and/or confirm the intention of extra edits.

However, little has been done on characterizing extra edits on floss-refactoring. We believe that, by better understanding how developers often relate refactoring to extra edits, advances can be done to assist refactoring related activities, such as refactoring fault detection, refactoring revision, and to guide developers on when and how to perform floss-refactoring. For instance, Kim et al. [24] pointed out that a support for floss-refactoring is needed, due to frequent errors. The current tool support in this sense is quite limited by focusing only on refactoring detection (e.g., [39; 46; 53]), and/or on highlighting extra edit between two versions of a code (e.g., [1; 18]). We believe that before any tool support, it is important to better understand developers common practices and how they combine refactorings and extra edits. Conclusions in this sense can then help the development of new solutions/tools for refactoring review, validation, and/or systematic analysis. Such information can also help other researches. For instance, empirical investigations tend to use random code changes to simulate real refactoring faults [1; 14]. By discovering how developers combine refactorings with other edits in the real world, researchers can then provide a better simulation on floss-refactoring faults.

1.2 Motivating Example

In this section, we present a situation where a refactoring is applied together with other edits that change the behavior of a method. To exemplify how *Extract Method* refactorings are often combined with extra edits, and the need for proper revision, consider the code in Figure 1.1[1] from the *BroadleafCommerce* repository[2][3]. The *BroadleafCommerce* project is an e-commerce framework designed for facilitating the development of enterprise and commerce-driven websites. Figure 1.1(a) shows the original version of the *persistCopyObjectTree* method, which is responsible for persisting the *copy* object provided as parameter. For that, the method performs a sequence of analyzes in order to decide the adequate treatment for the provided object (lines 2 and 7).

In a single commit, a series of edits were performed (Figure 1.1(b)). Code insertions are marked with '$+$', deletion with '$-$', extra edits are underlined. As we can see, an *Extract Method* edit was performed: the *If Statement*, along with its body (Figure 1.1(a) - lines 7-9), was extracted to the *persistPart* method (Figure 1.1(b)). However, in the same commit, a series of other extra edits were included: a new persistence strategy was introduced, the method signature and some condition expressions were updated.

Comparing the two versions of the *persistCopyObjectTree* method, we can see that there is a big difference in how the system stores the object. In its previous version, the method

[1]The code has been adapted for didactic purposes

[2]https://github.com/BroadleafCommerce/BroadleafCommerce

[3]Commit: a270461a25509c9b6bfc2396e416ef4f58d9a4ce

```
1  persistCopyObjectTree(Object copy,Set library, MultiTenantCopyContext context) {
2    if (library.contains(copy)) {
3      return;
4    }
5    library.add(copy);
6    ...
7    if (!genericEntityService.sessionContains(copy)) {
8      [if body]
9    }
10   context.checkLevel1Cache();
11 }
```

(a) original code

```
1     persistCopyObjectTree(Object copy,Set<Integer> library, MultiTenantCopyContext context){
2         if (library.contains(System.identityHashCode(copy))) {
3           return;
4         }
5        library.add(System.identityHashCode(copy));
6         ...
7  -      if (!genericEntityService.sessionContains(copy)) {
8  -        [if body]
9  +      if (copy.getClass().getAnnotation(Embeddable.class) == null) {
10 +        persistPart(copy, context);
11        }
12 -      context.checkLevel1Cache();
13    }
14
15 + persistPart(final Object copy, Context context) {
16 +     if (!genericEntityService.sessionContains(copy) &&
17 +         !genericEntityService.idAssigned(copy)) {
18 +      IdentityExecutionUtils.runOperationByIdentifier(
19 +          [if body]);
20 +     }
21 + }
```

(b) Extract Method refactoring with extra edits

Figure 1.1: An example floss refactoring. (a) The original code. (b) Code after an *Extract Method* refactoring. Lines 7-9 are extracted to create a new method `persistPart`. In the same commit the developer adds a series of extra edits (underlined statements).

saves the object itself (Figure 1.1(a) - line 5), in its new version it stores only the hash code (Figure 1.1(b) - line 5). As a consequence, the verification of the method was changed to check the hash code (Figure 1.1(b) - line 2).

Even though those extra edits might be intentional, they may lead to subtle faults and are worth double-checking. For instance, the developer could forget to update some verifications dealing with the *library* throughout the code. This fault would not throw an exception because the *contains* method compares instances of *Object*. Therefore, an unchanged verification may pass unnoticed. Moreover, the cache verification at the end of the source method (Figure 1.1(a) - line 10) was deleted in the new version (Figure 1.1(b) - line 12). A refactoring-aware revision would require the developer to confirm the intention of these extra edits and carefully analyze their impact.

Figure 1.1(b) (lines 15-21) presents the extracted method. Again, the extracted code also includes several extra edits. The condition expression was updated to verify the object's ID (line 17). With this edit, some of the objects that previously would be persisted by this method may not pass the new verification. Moreover, the extracted statements now run in a particular thread (line 18). This extra edit alone, if not well treated, might open the system to a set of unforeseen faults related to concurrent programming.

In summary, the presented commit was clearly a non-behavior-preserving refactoring. Several extra edits were applied along with an *Extract Method* (i.e., Statement Insert, Statement Delete, Statement Update, and Condition Expression Change), which increased the likelihood of fault introduction. By analyzing the relationship between refactoring and extra edits, we intend to help researchers/developers to better understand how developers perform refactorings in the real world, improving code review activities, and, therefore, provide new solutions to minimize the error-proneness of floss-refactoring. For instance, as future work, we intend to develop a mutation tool for introducing faults based on the refactorings found in a given system. Such tool may help other works that need to run controlled empirical studies using refactoring faults.

1.3 Objective

During refactoring activities, developers usually do not follow the mechanics proposed by Fowler [15] to prevent unexpected behavior changes. According to Silva et al. [45], refactorings are mainly driven by requirements changes instead of bad smells, as it was initially proposed. As discussed in the previous section, it is common for developers to apply extra edits during the refactoring process - floss-refactoring.

Therefore, the main goal of this work is to help researchers and developers to better understand how floss refactorings are performed in practice. For that, we focused on investigating the relationship between refactorings and extra edits, regarding their appearance[4] and number[5], as well as analyze how these extra edits are performed. To achieve this goal, we split it into more specifics ones:

- Investigate which extra edits are more likely to appear in a commit with the presence of a given refactoring;
- Investigate how the refactorings performed by developers affects the number of extra edits in a commit;
- Investigate details about how extra edits are applied.

Regarding the relevance of our work, we believe that, by identifying what extra edits are related to the presence of refactoring and how its number is affected, we can help the development of new tool support for floss-refactoring review, validation and/or impact analysis. Also, analysing how extra edits are performed can help on floss-refactoring analysis, by finding how frequent extra edits appear inside refactored entities, and what entities are usually changed. We choose to analyze the whole commit when analyzing the presence and number of extra edits to better understand what happens with a system when it is refactored, while the analysis on how extra edits are applied focuses on the edits applied on refactored entities. All this information can also help researchers better design and model their studies. Thus, their simulation of code change would reflect real world scenarios.

1.4 Contributions

In summary, the main contributions presented in this work are:

- An approach for automatic floss-refactoring data extraction;

[4]Whether the extra edit appears on a commit with refactoring.

[5]The number of occurrences of the extra edit in a commit with refactoring.

- Findings on the extra edit presence along with refactorings;
- An analysis on how these extra edits are performed;
- A dataset extracted from git repositories for future floss-refactoring studies.

In this work, we describe our approach for automatically extracting floss-refactoring data, and we discuss the reasoning behind each choice. Our findings showed that depending on the refactoring performed, there are changes on the likelihood of some extra edits to be applied, as well as in its number of occurrences. For instance, there were 27% increase in chances of appearing an *Additional Functionality* in a system during the *Extract Method* refactoring, and, after its first occurrence, there was 13% increase in the number of occurrence for each *Extract Method* refactoring.

As for the inner method changes, some types of statements are more likely to be modified. For instance, there was approximately 13% more *Method Invocation* statements introduced in a system for this refactoring, while the increase for the *Return Statement* was only 7%. Last but not least, we provided the dataset extracted during our exploratory study.

1.5 Structure

In Chapter 2, we provide some background to help understand this work. In Chapter 3 we explain our strategy for floss-refactoring data extraction. In Chapter 4, we describe the empirical study carried out in this work to understand extra edits patterns when developers apply different types of refactoring transformations. In Chapter 5 we investigate how these extra edits are performed. In Chapter 6, we present some related work. Finally, in Chapter 7, we present the concluding remarks of this dissertation, as well as possible future works.

Chapter 2

Background

In this chapter, we present the background of concepts used in this document, necessary for understanding this work. First, we describe program refactoring on its basic, explaining their implementation and related faults. Then, we describe the tools used in the analysis.

2.1 Program Refactoring

During software development, as the system grows, its complexity tend to grow as well. Methods too long, duplicated code fragments, unclear method and field names are examples of code problems that hinder its readability. In 1992, Opdyke [35] coined the term refactoring in his Ph.D. thesis as a way to minimize such problems. A set of structural changes applied to a software aiming to improve quality aspects, such as readability, reusability, and maintainability, while its observed behavior remains unchanged.

Opdyke formally defined refactorings as *generalizing the inheritance hierarchy, specializing the inheritance hierarchy* and *using aggregations to model the relationships among classes*. Later, Fowler [15] assembled a catalog[1] with 72 refactorings types, each one with specificities regarding purpose and granularity. Fowler also introduced the term *bad smells* to represent code problems related to quality, which are indicators for code structures that need improvements.

Refactorings are widely used during software development, accounting for nearly 30% of the modification of a system [47]. In Agile Methods, such as XP [3] and TDD [4], refac-

[1]https://refactoring.com/catalog/

torings are even more crucial for the development process, since it is used to improve the system architecture and readability continuously.

In the past years, there have been many initiatives for developing refactoring tools. The first refactoring tool was proposed by Roberts [41] for the Smalltalk language [19]. Afterward, researchers have improved correctness and applicability of refactorings. Nowadays, most popular Integrated Development Environments (IDEs), such as Eclipse, IntelliJ, NetBeans, and Visual Studio, include support for automated refactoring. However, even with all the support provided by the refactoring tools, studies have shown that developers usually apply refactoring operations manually [32; 33; 34]. According to Vakilian et al. [55], there are many reasons for the underuse of refactoring tools, such as usability, awareness, trust, among others.

2.1.1 Refactoring implementation

Fowler's catalog describes over seventy different types of refactorings. In his catalog, he describes the motivation and mechanics for each refactoring type. To demonstrate the mechanics presented by Fowler, consider the code in Figure 2.1(a). By analyzing this code, a developer can see a comment in the method's body. According to Fowler, comments are often used as a deodorant for bad smells in the code. That is, even though code comments improve code readability, their presence can indicate the need to restructure the code. If the comment explains what a code block does, Fowler recommends the application of an *Extract Method* refactoring, where the newly created method's name should explain its function.

In a *Extract Method* refactoring, a code fragment is extracted to a new method with a meaningful name, capable of explaining the method's purpose. To perform this edit, Fowler's mechanics suggest a procedure comprised of nine steps:

1. Create a new method with a name that explain what it does;
2. Copy the extracted code from the source method into the newly created method;
3. Scan the extracted code for references to any variables that are local in scope to the source method. These are going to be local variables and parameters in the new method;

```java
public class C1{
  public void printOwing(Invoice invoice) {
    printBanner();
    double amount = calculateAmount(invoice);

    //print details
    System.out.println("name: " + invoice.client);
    System.out.println("amount: " + amount);
  }
  ...
}
```

(a) original code

```diff
 public class C1{
   public void printOwing(Invoice invoice) {
     printBanner();
     double amount = calculateAmount(invoice);

-    //print details
-    System.out.println("name: " + invoice.client);
-    System.out.println("amount: " + amount);
+    printDetails(invoice,amount);
   }

+  public void printDetails(Invoice invoice, double amount) {
+    System.out.println("name: " + invoice.client);
+    System.out.println("amount: " + amount);
+  }
 ...
 }
```

(b) Code after Extract Method refactoring

Figure 2.1: Example of Extract Method refactoring (Fowler [15], Adapted by the author).

4. See whether any temporary variables are used only within this extracted code. If so, declare them in the new method as temporary variables;

5. Look to see whether any of these local-scope variables are modified by the extracted code. If one variable is modified, see whether the extracted code can be treated as a query and assign the return to the variable concerned. If this is inconvenient, or if there is more than one such variable, the method cannot be extracted as it stands;

6. Pass into the target method as parameters local-scope variables that are read from the extracted code;

7. Compile after dealing with all the locally-scoped variables;

8. Replace the extracted code in the source method with a call to the new method;

9. Compile and test.

In Figure 2.1(b) a *Extract Method* refactoring was performed. A commented code fragment was extracted to a new method whose name explains the code purpose, making a comment unnecessary. Usually, in the *Extract Method* refactoring, only the refactored method is modified. However, there are refactorings that may have implications across multiple classes, where a developer might need to update all the affected classes, such as a *Move Method* refactoring. As its name suggests, in a *Move Method* refactoring, a developer moves a method from a class to another where it makes more sense. The mechanics proposed by Fowler for this refactoring is comprised of 11 steps, including the update of all references to the refactored method:

1. Examine all features used by the source method that are defined in the source class. Consider whether they also should be moved. If a feature is used only by the method you are about to move, you might as well move it too;

2. Check the sub- and super-classes of the source class for other declarations of the method;

3. Declare the method in the target class;

4. Copy the code from the source method to the target. Adjust the method to make it work in its new home;

5. Compile the target class;

6. Determine how to reference the correct target object from the source;

7. Turn the source method into a delegating method;

8. Compile and test;

9. Decide whether to remove the source method or retain it as a delegating method;

10. If you remove the source method, replace all the references with references to the target method;

11. Compile and test.

To demonstrate the mechanics for the *Move Method* refactoring, consider the code present in Figure 2.2(a). The *overdraftCharge* method takes into account the type of account to determine the form of calculation to be used. If more account types are added, and new rules for calculating the overdraft for each type, it would make more sense for the *overdraftCharge* method to be in the *AccountType* Class. In Figure 2.2(b), a *Move Method* refactoring was performed. Some adjustments were necessary to make the method fit in its new place, such as adding a parameter (Figure 2.2(b) - line 26) and updating a method call from within the method (Figure 2.2(b) - line 27). In addition, since the source method was deleted, it was necessary to update all its references to refer the new one (Figure 2.2(b) - lines 16-17).

2.1.2 Refactoring Faults

Despite the intention of improving software quality, the refactoring application often leads to error-proneness [25]. Studies have found that refactored code segments are usually the target of bug fix after the refactoring application [2; 24]. Moreover, developers perform most refactorings manually, even though most popular IDEs offer built-in refactoring tools, which increases the risk of bug introduction [1; 32; 33; 55]. According to Murphy et al. [32], about 90% of all refactorings edits are manually applied. Nevertheless, recent studies have shown that even widely used tools perform incorrect refactorings, unexpectedly changing the system behavior [30; 49]. Therefore, manual refactoring remains developers' standard procedure for performing refactoring tasks [25].

By decomposing the refactorings into micro steps, including compiling and testing, Fowler's mechanics [15] tries to turn refactoring a systematic process. However, even with the high acceptance of these mechanics, they do not prevent developers from introducing faults in the system. There are many cases where the developer's expertise and knowledge about the system have an impact on the refactoring's outcome [17]. For instance, Figure 2.3

```java
public class Account{
  public double overdraftCharge() {
    if (type.isPremium()) {
      double result = 10;
      if (daysOverdrawn > 7)
        result+=(daysOverdrawn -7)*0.85;
      return result;
    }
    else
      return daysOverdrawn * 1.75;
    }

  public double bankCharge() {
    double result = 4.5;
    if (daysOverdrawn > 0)
      result += overdraftCharge();
    return result;
  }

  private AccountType type;
  private int daysOverdrawn;
}

public class AccountType{
  ...
}
```

(a) original code

```diff
  public class Account{
-   public double overdraftCharge() {
-     if (type.isPremium()) {
-       double result = 10;
-       if (daysOverdrawn > 7)
-         result+=(daysOverdrawn -7)*0.85;
-       return result;
-     }
-     else
-       return daysOverdrawn * 1.75;
-     }

    public double bankCharge() {
      double result = 4.5;
      if (daysOverdrawn > 0)
-       result += overdraftCharge();
+       result += type.overdraftCharge();
      return result;
    }

    private AccountType type;
    private int daysOverdrawn;
  }

  public class AccountType{
+   public double overdraftCharge(int daysOverdrawn) {
+     if (isPremium()) {
+       double result = 10;
+       if (daysOverdrawn > 7)
+         result += (daysOverdrawn - 7) * 0.85;
+       return result;
+     }
+     else
+       return daysOverdrawn * 1.75;
+   }
    ...
  }
```

(b) Code after Move Method refactoring

Figure 2.2: Example of Move Method refactoring (Fowler [15], Adapted by the author).

shows a situation where a developer with little knowledge about the system applies a *Move Method* refactoring and ends up changing the system behavior.

In this example, consider the code presented in Figure 2.3(a). Suppose that, by some rea-

```
1   public class C1 {
2     public int calculate(int x, int y){
3       return x * y;
4     }
5     ...
6   }
7   public class C2 {
8     public int calculate(int x, int y){
9       return x + y;
10    }
11    ...
12  }
13  public class C3 extends C1{
14    ...
15  }
```

(a) original code

```
1     public class C1 {
2       public int calculate(int x, int y){
3         return x * y;
4       }
5       ...
6     }
7     public class C2 {
8  -    public int calculate(int x, int y){
9  -      return x + y;
10 -    }
11      ...
12    }
13    public class C3 extends C1{
14 +    public int calculate(int x, int y){
15 +      return x + y;
16 +    }
17      ...
18    }
```

(b) Code after Move Method refactoring

Figure 2.3: Example of Move Method refactoring changing a system behavior.

son, a developer decides to apply a *Move Method* refactoring, moving the method *calculate* from class *C2* (Figure 2.3(a) - lines 8-11) to class *C3* (Figure 2.3(b) - lines 14-16). Apparently, the system behavior remains unchanged, since the system is free of compilation errors. However, the method recently placed in class *C3* overrides a method with the same signature but different behavior from the super class *C1* (Figure 2.3(b) - lines 2-4). As consequence, all entities that uses the method from class *C1* through objects from class *C3*, will receive a return different from the expected.

Another common reason for refactoring faults is floss-refactoring [24], which refers to scenarios when developers apply refactoring edits alongside other changes, i.e. non-refactorings changes. As shown previously, even following Fowler's mechanics, there still is a chance of introducing fault. By applying non-refactoring changes simultaneously, a developer increases the chances of fault introduction, since it is necessary to analyze the impact of all changes in other code entities. To show how extra edits can introduce subtle faults, consider the code present in Figure 2.4(a).

In the example, the method *getStr* (Figure 2.4(a) - lines 2-9) analyses if there is a certain character in a given string. In a positive case, it returns a sub-string of the original string. In a negative case, it returns an empty string. Then, the *printStrLen* method (Figure 2.4(a) -

```
1   public class C1{
2     public String getStr(String str){
3       if(str.contains(":")){
4         int index = str.indexOf(":") + 1;
5         return str.substring(index);
6       }else{
7         return "";
8       }
9     }
10    public int printStrLen(String str){
11      String aux = getStr(str);
12      return aux.length();
13    }
14  }
```

(a) original code

```
1      public class C1{
2  -     public String getStr(String str){
3  +     public String getString(String str){
4          if(str.contains(":")){
5            int index = str.indexOf(":") + 1;
6            return str.substring(index);
7          }else{
8  -         return "";
9  +         return null;
10         }
11       }
12       public int printStrLen(String str){
13 -       String aux = getStr(str);
14 +       String aux = getString(str);
15         return aux.length();
16       }
17     }
```

(b) Code after Rename Method refactoring

Figure 2.4: Example of floss refactoring (Rename Method).

lines 10-13) returns the length of the returned string. In order to improve code readability, a developer decides to perform a *Rename Method* refactoring (Figure 2.4(b) - lines 2,3,13,14). However, instead of following strictly Fowler's mechanics, the developer also changes the the method return (Figure 2.4(b) - lines 8,9). Now, instead of returning an empty string, it returns a *null* object.

This change has no apparently impact on the system behavior, since it does not present any compilation error. However, depending on the argument provided to the *printStrLen* method, it can throw a *RuntimeException*, breaking the system. For instance, if the string provided as argument is *"abcd"*, the *getString* method will return a *null* object (Figure 2.4(b) - line 9). Then, the *printStrLen* method will try to call a method from the object returned (Figure 2.4(b) - line 15), but, since it is a *null* object, the system will throw a *NullPointerException*, which is a *RuntimeException*.

2.2 Analysis tools

In this section, we present an overview of the tools used to support our investigation.

2.2.1 RefDiff

The knowledge about refactoring changes in a system can help researchers and developers to understand a software evolution. This information has helped researchers investigate why developers refactor [45], how they refactor [33], when refactorings induce bugs [2], and when developers use tool to support the refactoring activity [55].

The information about refactoring activities can also help developers during code revision and system integration. During a field study at Microsoft, Kim et al. [25] verified that, when large refactorings operations are applied, the developers present difficulty in reviewing or integrating code changes. The difficulty presented by developers is basically related to rename and move of code elements. Thus, by automatically identifying the refactorings operations, it is possible to develop visual support for reviewing refactorings, automatically adapt the source code to a refactored API, merge conflicts, among others.

However, many of the existing approaches faced problems such as precision, recall, or the need to compile the code before identifying the refactorings. Thus, to fill this gap, Silva et al. [46] proposed an automated approach that identifies refactorings through a combination of static analysis and code similarity. The RefDiff tool runs directly on version history repository and is capable of identifying 13 refactoring types that are among the most used.

The tool's algorithm consists basically in two main phases: Source Code Analysis and Relationship Analysis. The first phase builds a model to represent high level code entities before and after code changes (e.g. classes, attributes and methods). The second phase finds relationships between the entities present in the code before and after the changes through code similarity heuristics. One key aspect of the algorithm is the use of an adaptation of TF-IDF (Term Frequency-Inverse Document Frequency) [43] as the similarity index. The TF-IDF is a well-known information retrieval technique, which reflects the importance of a term to a document, while taking into account the collection the document belongs.

Therefore, the tool is able to find refactorings throughout the whole commit, regardless of the refactoring type[2] and whether it contains extra edits. As output, the tool not only provide the refactoring type applied, but the complete signature of the refactored entity before and after the refactoring. For instance, if a developer executes the tool on the code examples presented throughout this chapter, its output would contain the information presented in

[2]Considering the 13 refactoring types it can identify

Table 2.1.

Table 2.1: RefDiff Output

Code Example	Refactoring	Entity Before	Entity After
Figure 2.1	Extract Method	C1.printOwing(Invoice))	C1.printDetails(Invoice,double)
Figure 2.2	Move Method	Account.overdraftCharge()	AccountType.overdraftCharge(int)
Figure 2.3	Move Method	C2.calculate(int,int)	C3.calculate(int,int)
Figure 2.4	Rename Method	C1.getStr(String)	C1.getString(String)

2.2.2 SafeRefactor

In order to ensure the safe application of refactorings, Soares et al. [48] proposed SafeRefactor, a tool for checking the safety application of refactorings in sequential Java programs. SafeRefactor is an Eclipse plugin[3] that receives a program version and the refactoring to be applied using the Eclipse refactoring API. By generating and running unit test suites, it decides whether the refactoring to be applied by Eclipse will change the system's behavior. Also, it has the alternative of receiving a refactored program version instead the refactoring to be applied. The data can be provided to the tool through its GUI or by command line, which can be useful for automated experiments.

After receiving the input data, if it receives refactoring to be applied, the SafeRefactor first refactors the source code using the Eclipse refactoring API. Otherwise, it continues to the next step. In the second step, it identifies the common methods between the source and refactored programs through static analysis. To check the programs' behavior, SafeRefactor generates a test suite for the common methods using *Randoop* [36], a tool for automatic random generation of unit tests. The next step is to run the tests in both source and target programs. For the refactoring to be considered safe, the test's outcome has to be the same for both programs. That is, if a test passes in one version, but fail in the other, the refactoring is considered not safe, that is, the transformation changed the system behavior.

Saferefactor has also been used for testing refactoring engines (e.g., [29]), which have several faults in well-known tools (e.g., Eclipse Refactoring tool).

[3]www.dsc.ufcg.edu.br/~spg/saferefactor

2.2.3 ChangeDistiller

In order to keep a software useful as long as possible, it is necessary to adapt it to new requirements and new environments. To aid developers and researchers on understanding the changes applied to a software, many techniques were developed [16; 56; 57]. These techniques relate changes based on the developers who applied them and detect maintainability hot spots. However, these techniques do not consider structural changes in the source code. In this context, Fluri et al. [12] developed ChangeDistiller, a tool that extracts fine-grained source code changes from two versions of a program. The authors improved the Chawathe's algorithm [8] for extracting changes in hierarchically structured data.

The ChangeDistiller's algorithm uses the Abstract Syntax Trees (AST) representation of a source code to extract detailed information about statements editing. It analyses the changes based on basic tree edit operations such as insert, delete, update or move of tree nodes. To compute these operations between two ASTs, the algorithm uses bigram string similarity to match string between nodes (e.g., variable declaration, method invocation, and so forth), and, for matching source code structures (such as if statements or loops), it uses the subtree similarity of Chawathe et al.[8].

After the matching process, the algorithm calculates the edit script necessary to transform the source tree into the changed one. The edit is composed by insert, delete, update or move of statements in the code. The edit operations are then classified as source code changes. For that, the authors use the taxonomy presented in their previous work [13], which defines 35 change types. In this work, Fluri et al. present a taxonomy for classifying source code change types based on code revision. The taxonomy can differentiate different levels on the code, as well as their impact on other code entities. Table 2.2 presents a subset of ChangeDistiller edit classification, as well as their description.

If we run ChangeDistiller in source codes previously shown, it would yield which changes were applied. For instance, consider a developer executes the tool in class *C1* present in Figure 2.1. Table 2.3 shows the edits found by the tool.

The tool identifies five edits in the code. The tool also provides some information about the found edits besides the edit's classification and the changed entity, such as the parent entity, which is important for analysing the context where the edit was applied. However, when dealing with entities like methods and classes, the tool only provides the entity's signature, lacking information about the entity's body structure.

2.3 Concluding Remarks

In this chapter, we covered the main concepts related to our research such as program refactoring, which is an activity that aims to improve the source design while its external behavior remains unchanged. We explained Fowler's mechanics for implementing some refactorings that are among the most common, and how the developer's lack of knowledge about the system, as well as intentional extra edits, can compromise the system behavior.

Moreover, we showed some tools useful for floss-refactoring analysis such as RefDiff, for detecting refactoring across software repository versioning history, SafeRefactor, for checking the safety of refactoring implementations, and ChangeDistiller, for listing source code edits.

Table 2.2: Extra Edit Types Description

Extra edit	Description
Statement Insert	A new statement was inserted to a certain method's body.
Statement Delete	A statement from a certain method is not present in its new version.
Statement Update	A statement from a certain method was updated.
Additional Functionality	A new method was added to a certain class.
Statement Parent Change	A statement was moved to a different code structure.
Removed Functionality	A method no longer exists in a certain class.
Condition Expression Change	The condition of a certain loop/control was updated.
Additional Object State	A new field was added to a certain class.
Removed Object State	A field no longer exists in a certain class.

Table 2.3: ChangeDistiller Output

Extra edit	Changed Entity	Parent Entity
Comment Delete	//print details	printOwing(Invoice invoice)
Statement Delete	System.out.println("name: " + invoice.client);	printOwing(Invoice invoice)
Statement Delete	System.out.println("amount: " + amount);	printOwing(Invoice invoice)
Statement Insert	printDetails(invoice,amount);	printOwing(Invoice invoice)
Additional Functionality	printDetails(Invoice invoice, double amount)	class C1

In the next chapter, we explain how we use the concepts and tools presented in this chapter to develop a strategy for floss-refactoring data extraction.

Chapter 3

Floss-Refactoring Extraction Strategy

Based on the background presented in the previous chapter, we developed a strategy for automatically extracting floss-refactoring data where detailed information about the refactorings and the non-refactoring edits are extracted from versioning history of maven java projects. In this strategy, we considered floss-refactoring the software versions that contain refactorings and presented behavior change, intentionally or not. We describe this process in the next sections.

3.1 Approach

In this section, we explain our approach for the floss-refactoring extraction. The extraction process is based on analysing commits from available open source projects and their repositories. In order to mine the repositories, we defined the data extraction process presented in Figure 3.1, which comprises three steps.

For each project P:

1. From P's repository, get SC_r, where $SC_r = sc1, sc2, ..., scn$ and sci is a pair of sequential commits that includes at least one refactoring edit;
2. From SC_r, find SC_e, where SC_e is a subset of SC_r containing only non-behavior-preserving pairs of commits, e.g., commits that include extra edits;
3. For each sci in SC_e, decompose the change edits and analyze which edits are part of the refactoring performed in the commit, and which edits are extra.

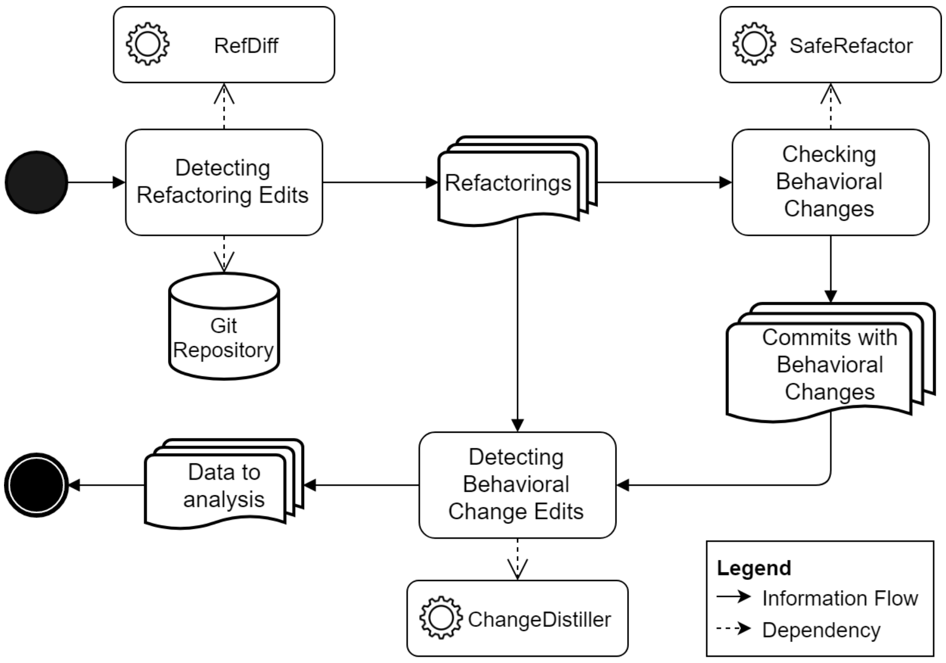
RefDiff
SafeRefactor
Detecting Refactoring Edits
Refactorings
Checking Behavioral Changes
Git Repository
Commits with Behavioral Changes
Data to analysis
Detecting Behavioral Change Edits
ChangeDistiller
Legend
Information Flow
Dependency

Figure 3.1: Data Extraction Process

The first activity consists of filtering commits that contain refactoring edits. This activity is extremely important because the remaining activities depend on this output. For that, we used the RefDiff tool [46]. RefDiff is a tool that detects refactorings performed between two versions of a Java program. It supports 13 types of refactorings that are listed among the most common ones [33]. The refactorings detectable by the RefDiff tool are: *Extract Method*, *Inline Method*, *Rename Method*, *Move Method*, *Extract Superclass*, *Extract Interface*, *Move Attribute*, *Pull Up Method*, *Pull Up Attribute*, *Push Down Method*, *Push Down Attribute*, *Rename Class*, *Move Class*, and *Move And Rename Class* (this is a specific case where developers apply both refactorings as if it was one). It is worth to mention that the tool can detect refactorings even when multiple refactoring types are applied in a single commit. Regarding RefDiff accuracy on detecting refactorings, we refer to its reported precision (100%) and recall (88%) [46].

Although other tools are designed for the same purpose (e.g., [11; 39; 53]), we chose RefDiff due to its high precision and because it works directly on software repositories. The refactoring detection tool RMiner [53] also works on software repositories and claims to have higher accuracy. However, by the time the tool was released, our study was in a later stage and we could not use it. Nevertheless, we run a manual validation of samples of refactorings detected by RefDiff, which provide similar accuracy for our dataset. To exemplify this step, if this step is executed in the code in Figure 2.4, it would detect a *Rename Method* refactoring, where the *getStr* method is renamed to *getString* (Figure 2.4(b) - lines 2,3).

After mining the projects' repositories, we filtered the commits to work only with the ones with behavioral changes, i.e., a pair of commits that are not pure-refactoring. For that, we used the SafeRefactor tool [48], which checks for safe refactoring edits. It analyzes two versions of a program and, by generating and running sets of unit tests, it indicates whether the program's behavior remained unchanged. Thus, in the context of our study, only pairs of commits that *failed* in SafeRefactor's analysis were used, since that means there are refactoring edits combined with behavior-changing ones. On the other hand, *pass* outputs from SafeRefactor were discarded since it indicates that no behavior-changing edit was performed between the two versions. In other words, in our study, we consider there is a behavior-change when the system that used to pass in a regression test suite, fails in the same suite when edit(s) are performed. Following the example from the previous step, SafeRefactor would detect a change in the system behavior, since, as we explained in Subsection 2.1.2, with the edits applied in the code (Figure 2.4(b) - lines 8,9), the system will throw a NullPointerException.

Although very helpful, in the context of our study, we faced several issues due to SafeRefactor's practical limitations. For instance, SafeRefactor's execution requires as input the directory where the source code files are placed. However, multi-modules projects (projects composed by the aggregation of two or more modules) often have multiple source code directories. Therefore, for analysing multi-modules projects, we run SafeRefactor on each module individually. In this case, if any module presents behaviour change, we classified the commit as behavior-changing.

SafeRefactor also requires a time limit for the test generation. To guide this definition, Soares et al. [47] run an empirical study that evaluated the coverage of the test suite generated by the tool with different time limits. The longer the timeout, the greater the number of tests generated. However, the coverage of the tests was not linearly proportional to the number of tests. As the number of tests increases, the coverage grows less. With this evaluation, the authors found that by doubling the time limit, the generated suites had an increase of only 2%. Thus, in our study, we used Soares's recommendation with a generation threshold of

120 seconds.

Moreover, due to practical limitations on its analysis process, SafeRefactor requires compilation error free versions of the project under test. To cope with this limitation, we decided to work only with Maven projects, which minimizes dependency issues that could be introduced during software evolution [27]. However, even with proper dependency management, several commits still presented compilation errors. This goes according to Tufano et al. [54] findings that state that 62% of a change history cannot be successfully compiled (as we will see in Section 4.2). Those problematic versions were then discarded. In multi-module projects, many versions presented compiling errors in some modules while others compiled successfully. For these cases, we classified the whole version based on the compiled modules. That is, even if only one module compiled, we classified the entire version as behavior-changing or behavior-preserving based on the module's output.

In our third step, we analyzed the extra edits types from commits that combine refactorings with behavioral changes. For that, we used the ChangeDistiller tool [12] and its change type classification [13] (see Table 2.2). ChangeDistiller is a tool that compares two versions of an Abstract Syntax Tree (AST) and classifies the source code edits performed between them. We chose this tool due to its fine granularity analyzes and feedback. Moreover, several other change-based empirical studies also use ChangeDistiller [1; 9; 24; 40; 53].

ChangeDistiller receives two versions of a given class (v1 and v2) and lists all edits performed between v1 and v2. However, since our study focuses on extra edits, we needed to filter ChangeDistiller's output list to remove all edits that were part of refactorings edits. For instance, when a *Extract Method* is performed, ChangeDistiller classifies the extracted method as an *Additional Functionality*. Therefore, we extended ChangeDistiller to include a module that analyzes both ASTs and ChangeDistiller's output list and, based on the refactorings returned by RefDiff, returns the edits that are not part of refactoring changes. Figure 3.2 shows how our filter interacts with ChangeDistiller during the 3rd step, detecting behavioral change edits.

The step of detecting behavioral change edits takes as input a commit and its parent. Then, it uses ChangeDistiller to find all edits applied in all classes in the commit and transfer them to the filter. However, to enable the filter's analysis, we added extra information to the

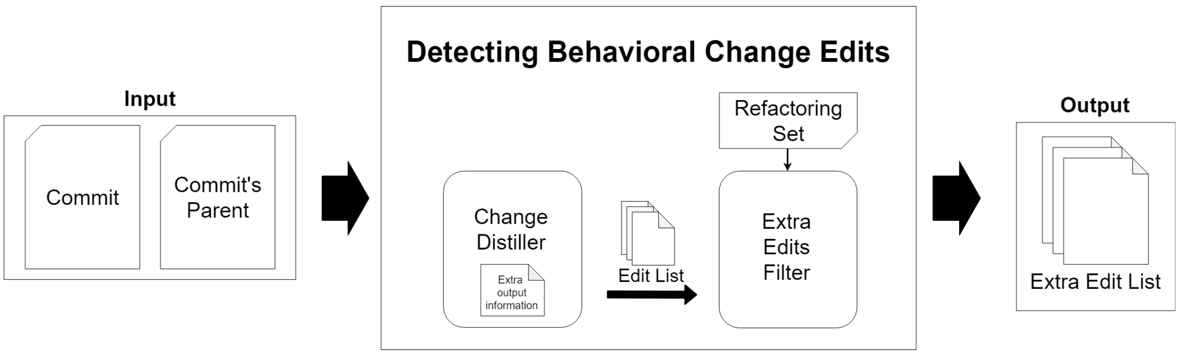

Figure 3.2: Detecting Behavioral Change Edits Diagram

ChangeDistiller's output (we will describe them in the next subsection). In the filter, all edits are analysed based on the refactoring set provided by RefDiff. Finally, as output, we have a list with the extra edits present in the commit. To exemplify this step, consider the code in Figure 2.4. Table 3.1 shows the edits ChangeDistiller would yield, and how the filter would classify them, regarding whether the edit is part of a refactoring or not. Therefore, only the edit where *Part of Refactoring* is *No* is returned by the strategy.

Table 3.1: Strategy Example

Extra edit	Changed Entity	Parent Entity	
Method Renaming	getStr(String)	C1	Yes
Statement Update	return "";	getStr(String)	No
Statement Update	String aux = getStr(str);	printStrLen(String)	Yes

In the next subsection, we provide details about the changes in ChangeDistiller's output, and how the filtering works in the next subsection.

3.1.1 Extra Edits Filter

For filtering extra edits, we first check whether each edit returned by ChangeDistiller is related to any refactoring returned by RefDiff. When a refactoring related edit is found, a specific treatment is applied based on the refactoring type it is related to. However, the details about the changed entities returned by ChangeDistiller were not enough for the analysis. Thus, we needed to extend ChangeDistiller so it can provide extra information that are important to our analysis, such as declaration details and body structure, when available. Figure 3.3 shows the ChangeDistiller's original output information and the additions we made.

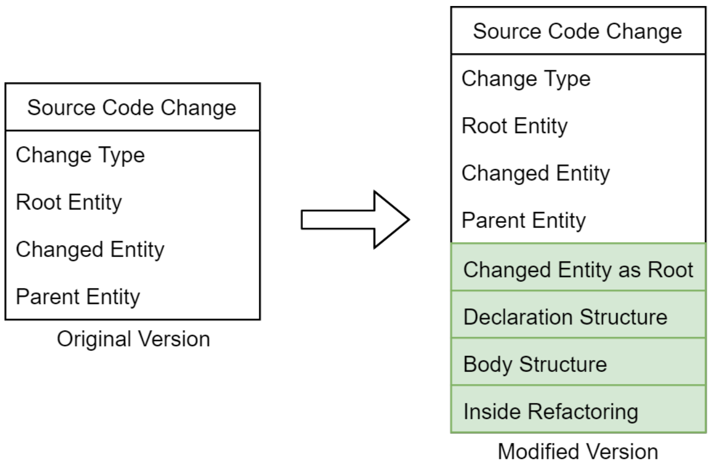

Figure 3.3: ChangeDistiller Output Modification

The original version presented information about the *Change Type, Changed Entity, Parent Entity* and the *Root Entity*. The information contained in each attribute, except Change Type, consists basically in the entity type, its unique name and its modifiers (e.g., private, public, final, static). The extra information added provide essential information such as the declaration and body structure of the changed entities (*Declaration Structure, Body Structure*), allowing detailed comparisons between entities. It was also necessary to add the changed entity as root entity (*Changed Entity as Root*) because some analysis required it as starting point. In addition, we added the *Inside Refactoring* attribute to allow us to analyze how frequent extra edits are carried out inside refactored entities. We marked the *Inside Refactoring* whenever an edit was found inside a refactored method or attribute.

ChangeDistiller runs over two versions of a single class, therefore, it was necessary to map the classes to its respective pair across the system versions. Thus, to address renamed and moved classes, we mapped the previous and new signatures based on the *Rename Class* and *Move Class* refactorings found by RefDiff. For each class in the source version, we checked whether its signature was changed by one of the mentioned refactoring, if so, we compared it with the class with the new signature provided by RefDiff. For new classes, we checked whether they were extracted entities from *Extract Superclass* or *Extract Interface* refactorings by comparing its signature with the ones in the refactoring set.

Since new classes can contain refactored entities, they were compared to an empty version of themselves, in order to list their entities. These entities are then classified as entities introduction (e.g., *Additional Functionality, Additional Object State*). It is important to mention that these changes are used only for future refactoring analysis, and are not listed as extra edits, but as disposable edits instead, since the edit under analysis in this situation is an *Additional Class*. For similar reasons, a similar approach was used for all deleted classes, with the entities being classified as entities removal (e.g., *Removed Functionality, Removed Object State*).

Since ChangeDistiller analyzes one class at a time, refactorings that comprises moving operations, such as moving an entire method or field to a different class (e.g., *Move Attribute*, *Pull Up Method*), are treated by ChangeDistiller as two separated changes: the removal of the refactored entity (the field or method to be moved) from the source class, and its introduction to the target class. In those cases, our module reruns ChangeDistiller specifically on the refactored entity searching for changes between its previous and new version. The tool compared the *Declaration Structure* and *Body Structure* of both entities, using the *Changed Entity as Root* as root of the edits found in the analysis. Any edit found by this analysis is considered an extra edit, since modifications in its body or signature are not part of the refactoring, except updated calls for refactored entities. Updated statements containing a call to any of the refactored entities returned by RefDiff, regarding move and rename refactorings, are not considered extra edits, since updating callers is part of the refactoring mechanics. Moreover, if the refactored entity was moved to a new class, or from a removed one, we use the previously mentioned disposable edits in the analysis.

We used a similar approach for dealing with the *Rename Method* refactoring. We also rerun ChangeDistiller on the refactored entity, comparing the *Declaration Structure* and *Body Structure* of source and target entities, but instead of moving the method to a different class, its name is the only modification expected, while its body should remain unchanged. For the *Extract Method* and *Inline Method*, since they may work with a series of statement transference, we perform a more complex analysis. Listing 3.1 shows the pseudo-code of the algorithm used for filtering *Extract Method*-related edits.

For each *Extract Method* refactoring found by RefDiff, we run EMFilter to select the extra edits applied in the extracted and source method. The EMFilter uses the algorithm in Listing 3.1 to classify whether the edits found by ChangeDistiller are extra edits or not. The changes classified as part of the refactoring (not an extra edit) are excluded from the edit list, remaining, in the end, only extra edits. First, the algorithm tries to match identical statements between the statements excluded from the source method and the ones present in the extracted method (lines 2 - 11). The matches found in the first step are considered part of a refactoring, and, therefore, excluded from the edit list.

Then, it identifies the statements that were updated during the refactoring process by verifying its similarity (lines 12 - 22). We used a similarity threshold of 60%, which is the same similarity level of ChangedDistiller. The selected statements are then removed from their previous lists and added as an update to a different list. To determine the edit type of the updated statements, we used the ChangeDistiller's classifier (line 19). The remaining unmatched statements are considered extra edits (with some exceptions that will be discussed later). To determine the edit type of the remaining statements from the extracted method, we also used the ChangeDistiller's classifier (lines 23 - 25). As for the remaining changes from the source method, they were added to the final edit list, as well as the updated statements and the new statements from the extracted method (lines 26 - 28).

```
1   EMFilter (m1:original Method, m2: extracted Method){
2       l1 = changes in m1
3       l2 = staments from m2
4       changes = {}
5
6       foreach s in l1{
7           if(l2.contains(s)){
8               l1.remove(s)
9               l2.remove(s)
10          }
11      }
12      updates = {}
13      foreach s1 in l1{
14          foreach s2 in l2{
15              if (similarity(s1,s2) >= 0.6){
16                  l1.remove(s1)
17                  l2.remove(s2)
18                  update = newUpdate(s1,s2)
19                  updates.add(Distiller.classify(update))
20              }
21          }
22      }
23      inserts = {}
24      foreach s in l2
25          inserts.add(Distiller.classify(s))
26      changes.addAll(updates)
27      changes.addAll(l1)
28      changes.addAll(inserts)
29      foreach s in changes{
30          if (isCaller(s) OR isSimpleReturn(s))
31                  changes.remove(s)
32      }
33  }
```

Código Fonte 3.1: Extract method filter algorithm

Finally, we verify if any of the final edits make references to any refactored entity. When an edit refers to a refactored entity, we consider the edit as part of the refactoring, since update references is a step in the refactoring process (lines 29 - 32). However, since we work only with static analysis, we do not have binding information. Therefore, we used the following classification criteria to decide whether a statement calls a refactored entity. Moreover, we also analyzed all edits returned by ChangeDistiller using these criteria.

For each call to a refactored method:

1. The edit must be an update;
2. The statement must contain a method call with identical name to the refactored method;
3. The method call must provide the same number of parameters the refactored method requires[1].

For each call to refactored fields:

1. The edit must be an update;
2. The statement must contain a variable call with identical name to the refactored field;
3. The variable called in the statement must not be a local variable:
    (a) Called from an object instance (e.g., obj.field); or

[1]We were not able to analyze the objects type due to polymorphic complications

   (b) There must not be a variable declaration with identical name to the refactored field in the method where the statement is placed.

For each call to refactored Classes:

1. The edit must be an update;
2. The update must be in:
   (a) Field type;
   (b) Method's return type;
   (c) Parameter type;
   (d) Variable declaration type.
3. The type must be identical to the refactored class.

We also check whether the edit is a *Return Statement*, once the introduction of *Return Statements* is expected in *Extract Method* refactorings. When a *Return Statement* is found, we check whether it is a simple return, i.e., if the statement contains only simple values (constant/variable) or none. In these cases, the edit is considered part of the refactoring (lines 29 - 32) and excluded from the final list. It is important to mention that all Return Statements containing more than simple values (e.g. arithmetic/Boolean operations, method invocations) were analysed to check whether it was extracted from the source method during matching process (lines 6 - 22). To better illustrate, in Listing 3.4, we show an example where a *Return Statement* is composed by a code extracted during a *Extract Method* refactoring.

For the example, we used a method that returns the number of days the given year has. In the example, the *leap year* verification (line 2) is extracted to a new method and placed directly on the *Return Statement* (line 10). During the matching process, the *If Statement* is matched with the *Return Statement*. Since there is no change in the extracted code, the edit is considered part of the refactoring and, therefore, excluded from the final edit list. If the condition was changed, then, it would be considered a *Condition Expression Change* (see the edit description in Table 2.2).

For the *Inline Method* refactoring, we analyze the inlined and destiny methods following a similar process, where it tries to match the statements introduced in the destiny method with

```
public class C{
    public int dayInYear(int year){
        if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
            return 366;
        else
            return 365;
    }
}
```

(a) Original code

```
public class C{
    public int dayInYear(int year){
  -     if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
  +     if (isLeapYear(year))
            return 366;
        else
            return 365;
    }

  + public boolean isLeapYear(int year){
  +     return ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0);
  + }
}
```

(b) Code after Extract Method refactoring

Figure 3.4: Return Statement Example.

the statements from the inlined method. Then, the remaining unmatched statements from the inlined method are classified by the ChangeDistiller's classifier as deleted statements.

3.2 Limitations

Construct Validity: Our approach for floss-refactoring data extraction relies on three different tools (RefDiff, SafeRefactor, and ChangeDistiller) and their accuracy may directly affect its output. We refer to their work regarding their accuracy [12; 46; 48]. Silva and Valente [46] evaluation states that RefDiff can detect refactoring edit with 100% precision and 88% recall. Regarding ChangeDistiller, Fluri et al. [12] evaluate its precision and recall (78% and 98%, respectively). We run a sampling test that validated both tools results in the context of our investigation. Moreover, since ChangeDistiller does not differentiate code changes within refactorings entities, we extend it by filtering its outputs based on the refactorings found by RefDiff tool. The filtering process was based on ChangeDistiller analysis and the Fowler's guide for each refactoring type [15]. Moreover, test cases were run for verifying the introduced features.

As for SafeRefactor, it uses automatically generated test suites to determine whether versions of a system are pure-refactoring. Therefore, its analysis might not detect all behavior changes. However, since our study works only with non-behavior-preserving pairs of commits, we used only versions that SafeRefactor presented failing test cases. Therefore, possible limitation on SafeRefactor analysis does not impact our study, since failing test cases guarantee behavior changes.

Internal Validity: One may argue that SafeRefactor's capability of detecting behavior changes may be influenced by the chosen time limit for test generation. However, Soares et al. [47] found that when considering generations beyond 120 seconds, the suites do not present considerable coverage increases. Therefore, our study consider 120 seconds as generation threshold for SafeRefactor.

3.3 Concluding Remarks

In this chapter, we presented our strategy for extracting floss-refactoring related data. With this strategy, we are able to extract detailed information about refactoring operations and extra edits applied simultaneously across the software repository's versioning history. Our data extraction process consists of three steps. For each step, we used an state-of-art specialist tool. We used RefDiff for finding the refactorings operation across all versioning history of the systems under analysis. SafeRefactor filtered out the commits with refactoring that did not present behavior change, i.e. pure refactoring commits. Finally, ChangeDistiller was responsible for listing the edits performed in the commit with behavior change. However, ChangeDistiller is not able to identify whether an edit is part of a refactoring operation or not. Thus, we developed a filter able to filter edits that are part of refactorings operations by taking into account Fowler's catalog and the refactorings found by RefDiff. In the next chapter, we present an empirical study conducted to investigate how refactorings and extra edits relate to each other, i.e. investigate whether there is an association pattern between them regarding their presence and frequency.

Chapter 4

Empirical Study: What extra edits should we expect during floss-refactoring?

By using the data extraction process presented in the previous chapter, we carried out an empirical study to understand extra edits patterns when developers apply different types of refactoring edits. In this chapter, we describe the design of this study, as well as its results and discussion.

4.1 Motivation and Procedure

Even though refactoring's main purpose is to improve readability without changing the system's behavior, studies have shown that this activity rarely happens alone [33; 45]. It is common for developers to apply extra edits during the refactoring process, which might increase the error-proneness of the activity. Yet, as far as we know, there is no empirical study in the literature on the likelihood of specifics types of extra edits to be applied alongside refactorings

In this context, we conducted an empirical study for analyzing how developers combine refactorings with extra edits, in order to identify relationship patterns between extra edits and different refactoring types. In this study, we analysed software repositories, selecting commits that contain both, refactorings and extra edits, to check whether there is or not any association pattern. Thus, we defined this study through the GQM approach [6] where the main goal of our study was to understand the following:

- How developers combine refactorings with extra edits?

Such information could be used to support the development of new tools for floss refactoring review, as well as to help researchers better design and model their studies, improving their simulations of real code changes. Thus, to help us to achieve our goal, we defined two specific research questions:

- **RQ1:** Which extra edits are more likely to appear alongside a given refactoring edit?
- **RQ2:** How the number of extra edits changes based on the refactorings performed by developers?

To answer these questions we took into account the following metrics:

- **M1:** The change in the likelihood of an extra edit appear based on the refactorings performed;
- **M2:** The change in the amount of an extra edit regarding the amount of refactorings;

The first metric (M1) takes into account the change (increase or decrease) in the odds of extra edits appear regarding the refactorings performed by the developers, and, it is used to answer the first research question (RQ1). The second metric (M2) analyzes the change (increase or decrease) in the number of extra edits presented alongside refactorings. This second metric is used to answer the second research question (RQ2).

4.2 Subject Selection

In our investigation, we mined 45 open-source Java projects using the data extraction strategy presented in Chapter 3. The projects used, and its characteristics such as LOC (Lines of Code), total of commits and compiling errors, are presented in Table 4.1.

As we can see in Table 4.1, the number of extra edits is much higher than refactorings. Thus, based on the numbers in the table, and on the Silva et al.'s findings [45], which indicates that refactorings are mainly driven by requirements changes rather than bad smells, we

Table 4.1: Selected Projects

Project	LOC	Total Commits	Compiling Errors	Selected Commits	Refactorings	Extra Edits
codefollower/Lealone	100789	1255	82	175	2249	8601
nutzam/nutz	92671	5883	275	59	153	2798
AsyncHttpClient/async-http-client	30812	3935	126	45	480	2264
BroadleafCommerce/BroadleafCommerce	185869	15545	1	32	129	3093
vkostyukov/la4j	13480	857	20	30	149	2132
thymeleaf/thymeleaf	40910	1645	172	29	174	3612
mrniko/redisson	101069	3743	127	26	129	2386
Athou/commafeed	9037	2521	70	23	52	441
dropwizard/metrics	19098	2375	1	23	101	691
alibaba/druid	296482	5824	397	16	81	1712
graphhopper/graphhopper	60847	3431	383	16	82	1141
Graylog2/graylog2-server	147138	14139	174	15	55	719
clojure/clojure	40728	3167	209	14	32	590
bennidi/mbassador	5520	334	2	14	88	833
apache/incubator-dubbo	101610	2491	212	12	68	284
opentripplanner/OpenTripPlanner	92171	8999	213	10	165	3016
notnoop/java-apns	5626	358	1	9	27	156
jline/jline2	10620	587	16	7	20	270
spring-projects/spring-data-mongodb	11176	2109	152	7	37	865
zeromq/jeromq	46558	348	3	6	26	1790
square/retrofit	20589	861	4	5	15	74
NLPchina/ansj_seg	13612	497	27	4	6	176
apache/giraph	98392	1065	160	4	59	1129
spring-projects/spring-hateoas	18521	400	32	4	6	151
brettwooldridge/HikariCP	12278	1836	32	3	9	169
AdoptOpenJDK/jitwatch	69726	580	2	3	85	68
jopt-simple/jopt-simple	9350	241	0	3	21	153
Kailashrb/scribe-java	5821	254	4	3	7	58
selendroid/selendroid	35201	960	2	3	26	580
xetorthio/jedis	30861	723	3	2	5	8
belaban/JGroups	122196	10124	840	2	2	33
HubSpot/Singularity	64665	2182	53	2	10	294
spring-projects/spring-data-neo4j	4530	538	47	2	20	78
undertow-io/undertow	138964	4042	5	2	14	212
Atmosphere/atmosphere	41453	4861	14	1	4	25
crashub/crash	40531	1844	125	1	1	31
cucumber/cucumber-jvm	28972	1929	0	1	1	16
HdrHistogram/HdrHistogram	9996	490	19	1	8	148
addthis/hydra	91989	2066	15	1	1	34
jayway/rest-assured	30967	1529	1	1	2	20
robovm/robovm	923138	1746	108	1	179	1494
scobal/seyren	6993	385	2	1	1	35
spring-projects/spring-data-jpa	18621	1059	49	1	2	24
spring-projects/spring-data-rest	2295	960	58	1	1	25
square/wire	31486	632	0	1	9	103
Total	3283358	121350	4238	621	4791	42532

could say that refactorings appeared to facilitate or supplement extra edits. Figure 4.1 shows the frequency of each refactoring type considering all analyzed commits. The refactoring types found in the projects used in our study go along with Murphy-Hill et al.'s findings regarding most common refactorings, such as *Move Method, Extract Method* and *Rename Method* refactorings [33]. Moreover, the column *Compiling Errors* shows the number of commits that were not able to be compiled, which does not include commits from multi-module projects where at least one module compiled.

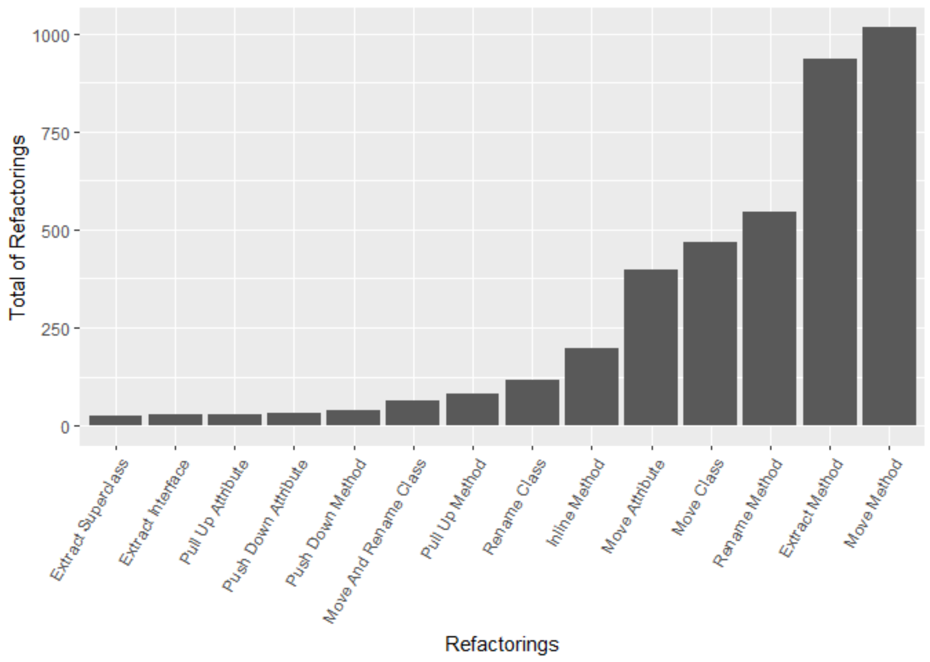

Figure 4.1: Refactoring frequency

Regarding the commits containing each refactoring type, we can take a look at Figure 4.2. One interesting fact is that the ranking of most common refactorings considering the number of refactoring performed was not the same of the ranking of refactoring considering the number of commit containing each refactoring type. This indicates that some refactorings, when used, are performed many times in few commits, while others are performed a few times in many commits. For instance, there are few commits with the presence of the *Move Method* refactoring, however, in these commits, the number of occurrences of this refactoring is high.

As for the extra edits, due to the time constraints, we considered the eight extra edit types with the highest number of occurrences. Moreover, the number of commits used goes according Peduzzi recommendation of [38] for sample sizes. Figure 4.3 shows the extra edits frequency throughout the selected commits. The description of the selected extra edit types can be found in Table 2.2. Moreover, the dataset collected during this empirical investigation is available online[1] for future floss-refactoring studies.

[1]Dataset available at: https://github.com/moreiraJS/Floss-refactoring-data

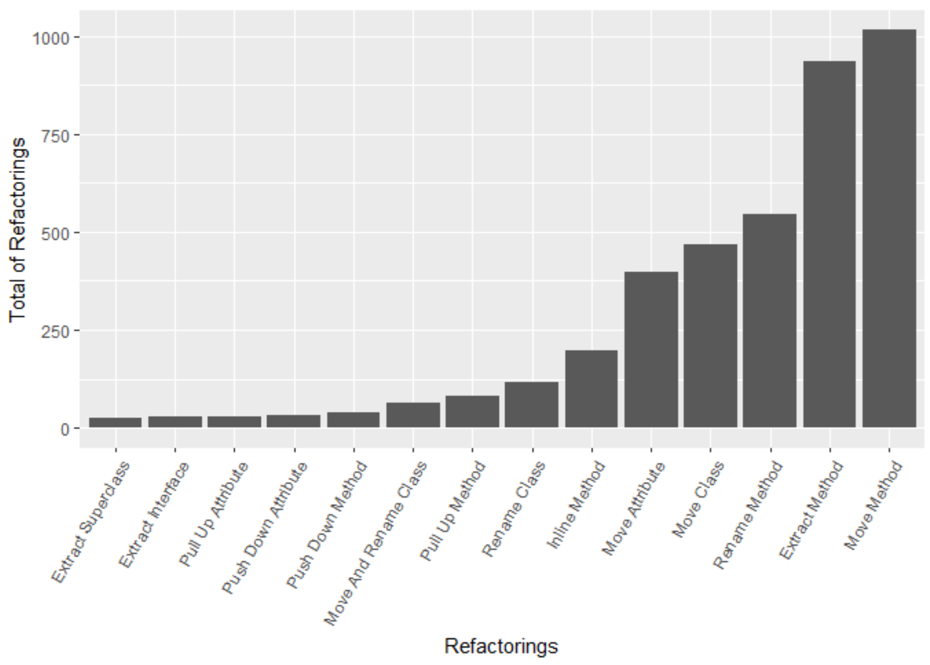

Figure 4.2: Commits with each refactoring type.

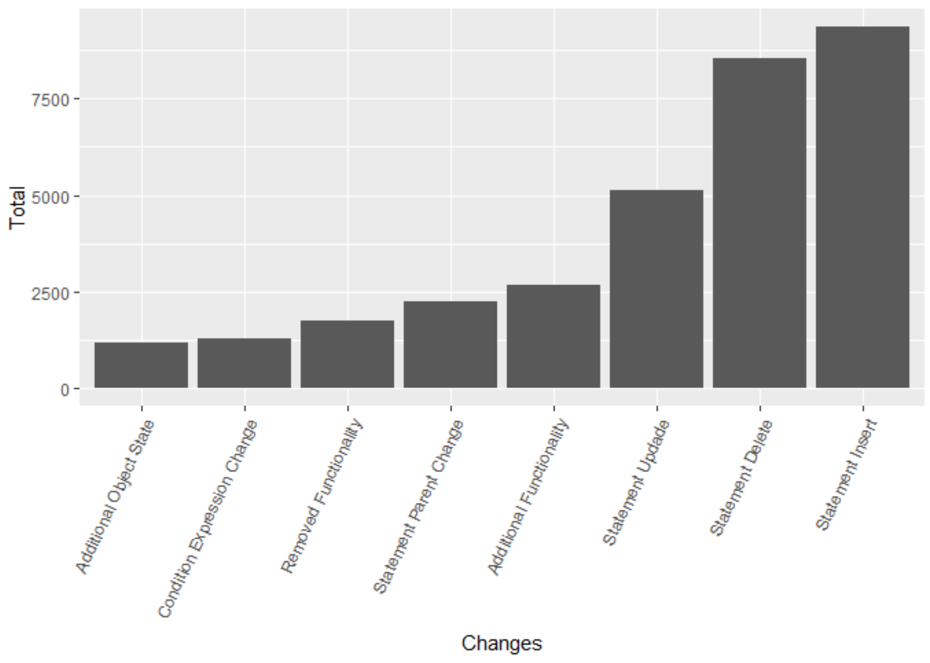

Figure 4.3: Extra Edits Frequency

4.3 Analysis Method

To analyze how developers combine refactorings with extra edits, we performed a series of regression analysis. Regression is a powerful statistical method for examining the relationship between two or more variables. We built regression models that relate refactorings to each extra edit. It is important to highlight that, although a regression model is often used as a prediction artifact, it can also be used as a describing tool for evidencing how the variables (dependent and independent) relate to each other. Several works use regression models to describe the relationship between two or more variables [5; 28; 37; 44], where the coefficients of the regression model represent the impact of the independent variables on the dependent variables.

Since we want to describe the extra edits based on the refactorings operations, our independent variable is the *refactoring type*, while the *extra edit* is the response (or dependent) variable. Before building the regression models, we first run a normality test that attested that the collected data does not follow a normal distribution. Then, we investigated what regression method best fits our data, based on its distribution. According to Hilbe [22], the most common distributions for counting data are Poisson and Negative Binomial. However, we could not use Poisson since it assumes a data where mean and variance are the same, and our data variance is at least 19 times greater than the mean (Table 4.2). On the other hand, the Negative Binomial estimates both parameters independently, allowing the distribution to deal with overdispersion (variance greater than the mean) and underdispersion (variance lower than the mean) [20].

Table 4.2: Data Details

Extra Edits	Zero Count	Mean	Variance
Statement Insert	145	15.07	1217.08
Statement Delete	178	13.76	1465.53
Statement Update	154	8.25	396.14
Additional Functionality	236	4.30	107.22
Statement Parent Change	362	3.59	133.46
Removed Functionality	359	2.83	66.15
Condition Expression Change	365	2.06	42.79
Additional Object State	348	1.88	35.74

Another common problem when dealing with counting data is the great number of zero counts. Considering that our dataset with 621 observations, the number of zero counts per extra edit type is very high (Table 4.2). According to Hilbe [21], Hurdle models are typically used to model the zero count. Therefore, for our analysis, we used *Hurdle Negative Binomial* models.

A Hurdle Negative Binomial model is divided into two parts, its Logit and Count models. The first is a regular logistic regression, which relates dependent variables with independent variables in order to predict the probability of the dependent variable being different from zero [0;1], i.e., its presence or absence. The Count model predicts the frequency of the dependent variable based on the independent variable starting from one ($[1, \infty]$). Equation 4.1 shows the general structure of the models we built, where *ExtraEdit* is the predicted frequency of an extra edit[2], *Refactoring* is the number of refactorings of a specific type in a given commit, and β_0 and β_1 are the found statistical coefficients.

$$ExtraEdit = \frac{e^{\beta_0+\beta_1 Refactoring}}{1 + e^{\beta_0+\beta_1 Refactoring}} \tag{4.1}$$

A Hurdle model is based on the assumption that the data is generated by two distinct processes, one before crossing the zero barrier and another process after, even though the processes' nature is not specified [21]. In practice, this means that factors that influence the first occurrence of an event may not influence how often this event occurs and vice versa. In other words, the significant predictors in predicting if an extra edit appears in a commit might not be significant in predicting the frequency it appears. As a result, predictors usually achieve different coefficients and significance levels in each model.

Moreover, we analyzed whether the independent variables generate significant impact on the dependent variables. To better understand our results, we analyzed the relation between the variables using *Odds Ratios* (OR), which is given by e^C, where C represents the coefficient of the independent variable in each model [21]. The Odds Ratios indicate the increase, or decrease, in the likelihood of the dependent variable for a unit increase of the independent variable. For instance, if the Logit model relates the *Extract Method* refactoring with a *Statement Insert* extra edit with an OR of 1.1, it means that for each *Extract Method* refactoring, there are 10% higher chances of appearing a *Statement Insert* edit in the same

[2]The logit model varies from [0;1], while the count model ranges from [1; ∞]

commit. Considering a Count Model in the same scenario with an OR of 1.2, it means a 20% increase in the number of *Statement Inserts* for each *Extract Method.* On the following subsections, we describe the relationship between refactorings and extra edits based on the analysis method described.

4.4 Results and Discussions

4.4.1 Which extra edits are more likely to appear alongside a given refactoring edit?

To answer the first research question, we used the Logistic part of the Hurdle Negative Binomial model. Table 4.3 reports the Odds Ratio (OR) obtained by each model. It contains the selected extra edits for each refactoring that achieved significance for at least one extra edit. The significance of the ORs is represented by the number of “ * ” after each value, where “ *** ” represents a significance level higher than 99.9%, while “ ** ” represents a significance level higher than 99%, and “ * ” a significance level higher than 95%.

To answer our first research question, we focused on the Logit model ORs presented in Table 4.3. The values in the Logit column represent the change in the likelihood of a respective extra edit appear in a commit (Section 4.1 - M1). Interestingly, all significant OR were higher than one, indicating an increase in the likelihood of the extra edit to be applied. Following, we discuss the results considering each model, independently. Moreover, we gave special attention to the *Extract Method, Rename Method* and *Inline Method* refactoring, since they had more significant ORs.

Extract Method

As we can see, all selected extra edits presented significant ORs for the *Extract Method* refactoring. Interestingly, all ORs values were higher than one, which means that there is an increase in the likelihood of these extra edits appearing in the same commit when a developer performs an *Extract Method* refactoring. For instance, the OR for *Additional Functionality* edit was 1.266680, which means that there are approximately 27% more chances of a new method to appear alongside an *Extract Method*, besides the extracted one. This finding sug-

Table 4.3: ORs from Logistic models between refactorings and extra edits.

Refactoring	Extra edit	Logit (M1)	Refactoring	Extra edit	Logit (M1)
Extract Method	Statement Insert	1.431857 ***	Rename Method	Statement Insert	1.026465
Extract Method	Statement Delete	1.455638 ***	Rename Method	Statement Delete	1.130114
Extract Method	Statement Update	1.401631 ***	Rename Method	Statement Update	1.041945
Extract Method	Additional Functionality	1.266680 ***	Rename Method	Additional Functionality	1.249350 **
Extract Method	Statement Parent Change	1.088286 **	Rename Method	Statement Parent Change	1.086418
Extract Method	Removed Functionality	1.087413 **	Rename Method	Removed Functionality	1.205934 ***
Extract Method	Condition Expression Change	1.116335 ***	Rename Method	Condition Expression Change	1.127316 *
Extract Method	Additional Object State	1.124656 ***	Rename Method	Additional Object State	1.111510 *
Move Attribute	Statement Insert	1.006585	Inline Method	Statement Insert	1.229395
Move Attribute	Statement Delete	1.014015	Inline Method	Statement Delete	1.302112 *
Move Attribute	Statement Update	1.021679	Inline Method	Statement Update	1.588910 *
Move Attribute	Additional Functionality	1.015841	Inline Method	Additional Functionality	1.057116
Move Attribute	Statement Parent Change	1.051892	Inline Method	Statement Parent Change	1.261771 **
Move Attribute	Removed Functionality	1.090687 *	Inline Method	Removed Functionality	2.038140 ***
Move Attribute	Condition Expression Change	1.071811	Inline Method	Condition Expression Change	1.221231 **
Move Attribute	Additional Object State	1.048800	Inline Method	Additional Object State	1.265517 **
Rename Class	Statement Insert	0.880308	Pull Up Method	Statement Insert	1.201190
Rename Class	Statement Delete	0.896036	Pull Up Method	Statement Delete	1.316673
Rename Class	Statement Update	1.288226	Pull Up Method	Statement Update	1.334808
Rename Class	Additional Functionality	0.873339	Pull Up Method	Additional Functionality	1.422740
Rename Class	Statement Parent Change	1.011210	Pull Up Method	Statement Parent Change	1.061242
Rename Class	Removed Functionality	1.047473	Pull Up Method	Removed Functionality	1.229994
Rename Class	Condition Expression Change	1.273402 *	Pull Up Method	Condition Expression Change	1.309655 *
Rename Class	Additional Object State	0.970098	Pull Up Method	Additional Object State	1.152468

gests that developers often combine *Extract Method* edits with feature introductions and/or updates, which corroborates with Palomba et al. [37] that state that this refactoring is related to the introduction of new features. Listing 4.1 shows an example from the *Broadleaf-Commerce*[3] project where in the same commit there is an *Extract Method* and an all new non-refactoring-related method was introduced.

```
1 + public static <G extends Throwable, J extends
2 +     RuntimeException> RuntimeException
3 +     refineException(Throwable e) {
4 +   return refineException(RuntimeException.class,
5 +     RuntimeException.class, null, e);
6  }
```

Código Fonte 4.1: Method created to overload an extracted method.

The *refineException* method, which is called in line 4, was created during an *Extract*

[3]https://github.com/BroadleafCommerce/BroadleafCommerce

Method edit. This method was extracted to be used by two other methods, besides the original one. All these changes were applied as part of a system update. The introduction of new methods can be error-prone, once it can override or be overridden by other methods, making the system behavior different from the expected. In this context, a code review is recommended, where the developer would give it a second look to ensure that the new method does not impact on the refactoring, and/or does not override, or is overridden, by any other method. Even though method overriding may be a reason for this refactoring [45], this can still be error-prone since the method that overrides the extracted method might not meet requirements from the source method.

The *Additional Object State* edit presented 12% increase in its likelihood. This edit increases the number of possible states of an object, which can directly impact comparison functions. Comparison functions takes into account specific aspects of an object, or the whole object, to return its result. However, by adding a new object state, the comparison function might return a different result from the expected. For instance, if the software uses a comparison function that compares the whole object, it might be necessary to develop a specific function to compare the other aspects except the new one. As for the *Remove Functionality* edit, its OR presented approximately 9% increase.

As we can see in Table 4.3, the *Statement Delete* extra edit presents the highest chances of appearing when an *Extract Method* is performed. There is a 45% increase on the likelihood of a developer perform a *Statement Delete* in a commit every time an *Extract Method* refactoring is performed. For the *Statement Insert* extra edit, we found the second highest OR value, i.e. there is an increase of 43% in the chances of a *Statement Insert* occur for each *Extract Method*. For the *Statement Update*, the increase is 40%, while it is nearly 12% for the *Condition Expression Change* edit, and 8% for the *Statement Parent Change* edit. In practice, it means that developers, when applying *Extract Method* edits, tend to change both the extracted and non-refactoring-related methods. These extra edits are worth revision, since often lead to behavior changes. The impact caused by these changes can be even greater when occurring in the extracted method, once the change made might seem harmless in the method, but have direct impact on its caller.

Listing 4.2 presents a method from the *Druid* project[4], where, in a single commit, code

[4]https://github.com/alibaba/druid

fragments from 26 different methods were extracted with *Extract Method* edits. However, even though the *Extract Method* refactorings were performed to reduce code duplicity, not all fragments were repetitive. The condition on line 2 was extracted from 24 methods, while lines 5-8 were extracted from the other two methods. Finally, Line 9 is a new condition added (extra edit).

```
1    public static boolean checkParameterize(SQLObject x) {
2      if (Boolean.TRUE.equals(x.getAttribute(ParameterizedOutputVisitorUtils.
        ATTR_PARAMS_SKIP))) {
3        return false;
4      }
5      SQLObject parent = x.getParent();
6      if (parent instanceof SQLDataType //
7          || parent instanceof SQLColumnDefinition //
8          || parent instanceof SQLServerTop //
9  +       || parent instanceof SQLAssignItem
10     ) {
11       return false;
12     }
13       return true;
14   }
```

Código Fonte 4.2: Method extracted for reducing code duplicity.

By analyzing these refactorings independently, in the first set of *Extract Method* refactorings (code extracted from 24 methods), the original code checked whether a specific attribute from the parameter is true (line 2). However, the updated code extended the condition to also analyze the object's parent (lines 5-9). This is an example of an *inner-method extra edit* where a developer should consider the impact the edits might generate to its callers. When it comes to an *Extract Method* refactoring, if the extracted code is modified with extra edits, the behavior of the source method is likely to change, and consequently, all callers of the original method can also be impacted. Moreover, the methods from which the condition in line 2 was extracted did not present the same behavior since the *checkParameterize* method introduced new restrictions. This more restrictive behavior may prevent some previously accepted events from occurring.

As mentioned, the method in Listing 4.2 is composed of code fragments from different methods. As a result, the refactorings impacted each others behavior. In other words, a code fragment can be considered an extra edit depending on the perspective. By the perspective of the first *Extract Method* (code extracted from 24 methods), lines 5-9 are extra edits, once they were not present on its source method. By the perspective of the other *Extract Method*, lines 2-4 are extra edits for the same reason: the condition was not present on its source method. Moreover, in this *Extract Method*, we also have an example of *Condition Expression Change*. The original version of the extracted condition did not verify the instance of *SQLAssignItem* (line 9). An error in a condition expression can have a great impact on a method's behavior, once it can avoid the execution of an entire code block. Therefore, we believe all those extra edits deserve proper revision to ensure that all behavior changes were intentional, and, consequently, no fault was introduced.

Rename Method

To address the edits along the *Rename Method* refactoring, we built a set of regression models whose ORs are also presented in Table 4.3. Different from the *Extract Method* refactoring, where all extra edits had significant increase when the refactoring is performed, for the *Rename Method* refactoring, only half of the edits achieve significant values. Interestingly, with exception of the *Condition Expression Change* edit, only outer-method edits presented significant change in its likelihood. It might suggest that the *Rename Method* refactoring is related to feature change, with the addition of new functionalities and object states, as well as the removal of probable obsolete methods. This corroborates with Palomba et al. [37], whose results showed correlations between the *Rename Method* refactoring and the introduction of features in some cases.

The extra edit with the highest OR is the *Additional Functionality* edit, having 25% more chances of appearing. Listing 4.3 shows an example of a new method created along with a renamed method from the *Lealone* project[5]. The developer renamed the method to better describe its functionality, while creating new methods to provide specific signatures.

```
- public Result asyncQuery(int maxRows) {
+ public Result executeQuery(int maxRows) {
        return query(maxRows);
  }

+ public Result executeQueryAsync(int maxRows) {
+       return executeQuery(maxRows);
```

[5]https://github.com/lealone/Lealone

```
+ }
```

Código Fonte 4.3: Renamed method along with newly created method.

As we said before, the addition of new method can be error-prone, since it can overload, or be overridden by, other methods, creating an unexpected behavior. Thus, it is necessary for the developer to thoroughly analyse the class' hierarchy in order to prevent such unexpected behavior.

The *Removed Functionality* edit obtained the second highest OR, having 20% increase when the refactoring is performed. In the sequence, the *Condition Expression Change* edit had 12% more chances of appearing. In the last position, the *Additional Object State* edit achieved 11% increase. These evidences may suggests that this refactoring is usually used followed by, or as a consequence of, a feature update. In this scenario, the developers updates the method's behavior while changes its name for a more specific one.

These findings could support future investigations on faults related to refactoring in this context, which could lead to the development of specific tools for refactoring driven code review.

Inline Method

The analysis of the edits alongside the *Inline Method* refactoring resulted in six significant ORs, in which one presented the highest OR among all Logit models. For each refactoring performed, the odds of a *Removed Functionality* appear is doubled. It is important to mention that the method inlined in the refactoring is not considered for this statistic. This goes along with Silva et al. [45] findings, whose results showed that the *Inline Method* is often used when a method (caller or callee) has become too trivial after code changes, indicating the removal of all unnecessary components. It also explains why the *Statement Update* and the *Statement Delete* also presented such high ORs, which was 59% and 30% respectively. In Listing 4.4 we have an example from the *La4j* project[6] with an inlined method that had some statements deleted.

In this example, the *decompose* method (lines 1 - 10) was inlined because it was only used by another method whose function was only to call the inlined method. Thus, the

[6]https://github.com/vkostyukov/la4j

developer joined both method through the *Inline Method* refactoring. However, the inlined method has an additional parameter that required an *applicability* verification (Lines 2-3) to ensure it could be processed by the method, and, after the refactoring, the verification was deleted.

```
 1 - public Matrix[] decompose(Matrix matrix, Factory factory) {
 2 -     if (!applicableTo(matrix)) {
 3 -         fail("This matrix can not be decomposed with Cholesky.");
 4 -     }
 5 -     Matrix l = factory.createMatrix(matrix.rows(), matrix.rows());
 6 -     for (int j = 0; j < l.rows(); j++) {
 7 -         ...
 8 -     }
 9 -     return new Matrix[] { l };
10 - }
11
12   public Matrix[] decompose(Factory factory) {
13 +     Matrix l = factory.createMatrix(matrix.rows(), matrix.rows());
14 +     for (int j = 0; j < l.rows(); j++) {
15 +         ...
16 +     }
17 +     return new Matrix[] { l };
18 -     return decompose(matrix, factory);
19   }
```

Código Fonte 4.4: Method inlined to trivial caller.

Now, since the *decompose* method (lines 12 - 19) uses a class' attribute instead a parameter, the verification was deleted. Consequently, if the class' attribute does not have an appropriate verification, the method might end-up with an unexpected behavior. Thus, it is necessary for the developer to verify if the attribute always meets the method's requirements. Moreover, both edits, the *Statement Parent Change* and the *Additional Object State* edit, had 26% increase in its chances of appearing, and the *Condition Expression Change* edit had 22%.

Other Refactorings

Aside from the refactorings previously mentioned, other three models achieved significant coefficients, each one from a different refactoring. The first one is the model associating the *Pull Up Method* refactoring and the *Condition Expression Change* edit, where the extra edit had 31% more chances of appearing. The second significant model associates the *Rename Class* refactoring and, again, the *Condition Expression Change*, increasing the odds of the extra edit appearing in 27%. The last significant regression model associates the *Move Attribute* refactoring and the *Removed Functionality* edit with an OR of 9%.

Moreover, all the remaining refactorings, named *Move Method, Move Class, Extract Interface, Pull Up Attribute, Push Down Method, Extract Superclass* and *Push Down Attribute*, did not achieved statistical significance for any of the extra edits under analysis with the logistic modeling.

4.4.2 How the number of extra edits changes based on the refactorings performed by developers?

To answer the second research question, we focused on the Odds Ratio values for the Count part of the Hurdle Negative Binomial models. Table 4.4 shows the ORs of the selected extra edits for each refactoring that achieved significance for at least one extra edit. We also used the number of " * " to represent the significance level for each value.

The Count model relates the frequency of extra edits to the number of refactorings performed in a commit (Section 4.1 - M2). That is, the increase in the amount of the extra edits for each refactoring operation. It is important to notice that the significance of the ORs in the Logit model does not influence the significance of the Count models ORs. The independent description of each model is presented in the following subsections. Moreover, we gave special attention to the *Extract Method, Rename Method, Move Method* and *Inline Method* refactoring, since they had more significant ORs.

Extract Method

The first difference we can notice between the count and logit models is that the *Removed Functionality* edit with the *Extract Method* refactoring was statistically significant for the Logit model but not for the Count model. This means that the first appearance of this extra edit is related to the presence of the *Extract Method* refactoring (Logit model), but its frequency is not.

Table 4.4: ORs from Negative Binomial models between refactorings and extra edits.

Refactoring	Extra edit	Count (M2)	Refactoring	Extra edit	Count (M2)
Extract Method	Statement Insert	1.114533 ***	Rename Method	Statement Insert	1.177633 ***
Extract Method	Statement Delete	1.141161 ***	Rename Method	Statement Delete	1.170552 ***
Extract Method	Statement Update	1.094384 ***	Rename Method	Statement Update	1.259477 ***
Extract Method	Additional Functionality	1.134636 ***	Rename Method	Additional Functionality	1.211437 **
Extract Method	Statement Parent Change	1.148790 **	Rename Method	Statement Parent Change	1.070850
Extract Method	Removed Functionality	1.059898	Rename Method	Removed Functionality	1.223966 **
Extract Method	Condition Expression Change	1.108380 ***	Rename Method	Condition Expression Change	1.179107 **
Extract Method	Additional Object State	1.124595 **	Rename Method	Additional Object State	1.211214 **
Move Method	Statement Insert	1.048838 **	Move Attribute	Statement Insert	1.044783 *
Move Method	Statement Delete	1.078665 **	Move Attribute	Statement Delete	1.084910 *
Move Method	Statement Update	1.117353 **	move Attribute	Statement Update	1.104773
Move Method	Additional Functionality	1.072178 *	Move Attribute	Additional Functionality	1.045734
Move Method	Statement Parent Change	1.026005	Move Attribute	Statement Parent Change	1.023241
Move Method	Removed Functionality	1.093503 *	Move Attribute	Removed Functionality	1.037910
Move Method	Condition Expression Change	1.047443	Move Attribute	Condition Expression Change	1.028287
Move Method	Additional Object State	1.056980	Move Attribute	Additional Object State	1.040554
Inline Method	Statement Insert	1.401339 ***	Pull Up Method	Statement Insert	1.276392 *
Inline Method	Statement Delete	1.367791 ***	Pull Up Method	Statement Delete	1.281887 *
Inline Method	Statement Update	1.333152 ***	Pull Up Method	Statement Update	1.137456
Inline Method	Additional Functionality	1.436110 ***	Pull Up Method	Additional Functionality	1.319140 *
Inline Method	Statement Parent Change	1.451596 *	Pull Up Method	Statement Parent Change	1.642743
Inline Method	Removed Functionality	1.274526 **	Pull Up Method	Removed Functionality	1.192519
Inline Method	Condition Expression Change	1.285870 *	Pull Up Method	Condition Expression Change	1.195919
Inline Method	Additional Object State	1.594564 **	Pull Up Method	Additional Object State	1.396251 *
Pull Up Attribute	Statement Insert	1.814541 *	Extract Superclass	Statement Insert	0.458700 **
Pull Up Attribute	Statement Delete	2.028691 *	Extract Superclass	Statement Delete	1.002113
Pull Up Attribute	Statement Update	2.036660	Extract Superclass	Statement Update	0.832236
Pull Up Attribute	Additional Functionality	1.928165	Extract Superclass	Additional Functionality	1.120199
Pull Up Attribute	Statement Parent Change	2.489300 *	Extract Superclass	Statement Parent Change	1.457934
Pull Up Attribute	Removed Functionality	1.819811	Extract Superclass	Removed Functionality	1.290949
Pull Up Attribute	Condition Expression Change	2.171890	Extract Superclass	Condition Expression Change	1.301950
Pull Up Attribute	Additional Object State	2.139534	Extract Superclass	Additional Object State	0.671409

Another interesting case is the *Statement Parent Change*, which was the only extra edit that got a greater OR value for the Count model. For each occurrence of an *Extract Method* refactoring, is expected an increase of approximately 15% in the amount of *Statement Parent Change* edits, which was the highest OR for the Count models. As for the *Additional Object State*, both models, Logit and Count, achieved basically the same OR, 12% increase. The *Statement Delete* extra edit presented an 14% increase in its amount, while, for the *Statement Insert*, the increase was 11%.

The *Additional Functionality* edit presented 13% increase in its frequency for each *Extract Method* performed. For instance, consider the *refineException* method, which is called by the method presented in Listing 4.1 (line 4). This method was extracted to allow signature overload, in which new methods were introduced with a different set of parameters. The *Condition Expression Change* got approximately 11% increase and the *Statement Update* presented the smallest significant OR, only 9% increase.

These results could support, not only refactoring driven tools, but also tools for simulating floss-refactoring, as well as mutation tools, since it is necessary to know the number of each extra edit expected alongside specifics refactorings for the changes applied to be consistent with the real world.

Rename Method

Like the *Extract Method* refactoring, the edits along with the *Rename Method* achieved significant OR for seven from the eight models presented in Table 4.4. They are the *statement Update*, with the highest OR, having an increase of 26% in its amount for each refactoring performed, followed by the *Removed Functionality* with 22%, the *Additional Functionality* and the *Additional Object State*, both with 21% increase, the *Condition Expression Change* and the *Statement Insert*, with approximately 18%, and finally the *Statement Delete* edit, with an increase of 17% on its count.

The example in Listing 4.3 shows a situation where the number of new functionalities added to the system increased when the *Renamed Method* refactoring was performed. Along with the example, we explained how new functionalities can introduce unexpected behavior if the developer does not validate the changes carefully. On Listing 4.5, we have another example of a *Rename Method* applied along with extra edits that can have great impact on the system behavior. The code presented here was adapted from the JOpt-Simple repository [7].

```
- private static String typeIndicator( OptionDescriptor descriptor ) {
+ private String extractTypeIndicator( OptionDescriptor descriptor ) {
      String indicator = descriptor.argumentTypeIndicator();
-      if(indicator == null || String.class.getName().equals( indicator ))
-          return "";
-      return shortNameOf( indicator );
```

[7]https://github.com/jopt-simple/jopt-simple

```
 7 +     if ( !isNullOrEmpty( indicator ) && !String.class.getName().equals( indicator ) )
 8 +         return shortNameOf( indicator );
 9 +     return null;
10       }
```

Código Fonte 4.5: Method renamed alongside inner extra edits

In this example, the method's name was changed from *typeIndicator* to *extractTypeIndicator*. However, the developer not only applied the rename method, but, in the same commit, he/she included a series of extra edits. First, the condition was modified (lines 4,7). The developer inverted the condition's semantic and changed the code structure, by switching statements' places. This change, if applied without proper care, could change the system behavior. Moreover, the developer also replaced a simple boolean operation with a more complex method call. In addition to the previous conditions, now it checks if the indicator is empty.

Other change made is an example of a *Statement Update*, where, instead of returning an empty *String* when the indicator is *null* or equals to *java.lang.String*, it returns a *null* pointer. Although these extra edits might have been intentional, they turn the refactoring revision and validation tasks hard, since they introduce behavior changes. Moreover, they might introduce faults. In fact, the last change can generate a *RuntimeException* if the method's caller is not ready to receive a *null* pointer as a return. This fault can pass unnoticed to the project's developer, since it only reveals itself during execution. In this sense, a tool for floss-refactoring review could identify these extra edits and warn the developers about the possible impacts the edit might have in the system.

Move Method

In the case of *Move Method* refactoring, the extra edits presented an interesting phenomenon, where five Negative Binomial models presented significant OR, but no Logit model achieved significance. In other words, no extra edit had its first appearance related to the refactoring. In practice, considering the assumption which Hurdle models are based on (see Section 4.3), this could indicate that, when the refactoring is applied intending to move a method to an appropriate class (the refactoring original intention [15]), no extra edits are expected. In the other hand, if the refactoring is performed to allow other edits to be applied, then certain extra edits should be expected.

Moreover, even though the models were statistically significant, the increase on the number of extra edits are fairly low. The edit with the highest OR was the *Statement Update*, having only 12% increase, approximately. All other edits achieved ORs lower than 10%. The *Statement Insert* and the *Statement Delete* got, respectively, 5% and 8%. According to Silva et al. [45], besides moving a method to an appropriate class, the *Move Method* refactoring is also used in a small percentage for enabling reuse, override, and for removing duplication. This goes along with ours findings, where the *Additional Functionality* edit got an OR of 7%, and the *Removed Functionality* got 9%.

Inline Method

The Inline Method refactoring was the only refactoring to achieve significance in all Negative Binomial models. Unlike the *Move Method* refactoring, all edits got fairly high odds ratio, ranging from 27% to 59%, for the *Removed Functionality* and the *Additional Object State*, respectively. These high values, combined with the significance level above 99.9% for half edits, suggest that this refactoring is usually performed during complex modifications, specially since, according to Silva et al. [45], some codes are easier to understand without the method call.

More generic inner-edits, such as *Statement Insert*, *Statement Delete* and the *Statement Update*, got 40%, 37% and 33% increase in its frequency for each *Inline Method* performed with significance above 99.9%. In Listing 4.6, we have an example of these edits. This code fragment was extracted from the *La4j* project[8]. In the example, the first *transformRow* method (lines 1-7) was inlined in the second one (lines 9-19). During the process, several extra edits were applied.

First, the method call in line 2 was updated, and now, calls a different method. Second, the *For Statement* was deleted (line 3), and a *While Statement* was inserted in its place in the target method. This changes the code structure, since, the loop control now needs to be done with the addition of new statements, which leads to other edits, the *Statement Inserts* in lines 12, 14 and 15. In addition, since now the loop control is made using a *VectorInterator*, the statement from line 4 was also updated (line 16). All these changes might impact directly the

[8]https://github.com/vkostyukov/la4j

system behavior, either the different loop structure used, which requires a careful review in order to ensure all loop controls used are fit for the situation, or the different methods called, which can provide different outcome from the expected.

```
-public Matrix transformRow(int i, VectorFunction function, Factory factory) {
-     Matrix result = copy(factory);
-     for (int j = 0; j < columns; j++) {
-             result.set(i, j, function.evaluate(j, result.get(i, j)));
-     }
-     return result;
- }

  public Matrix transformRow(int i, VectorFunction function) {
-     return transformRow(i, function, factory);
+     Matrix result = copy();
+     VectorIterator it = result.rowIterator(i);
+     while (it.hasNext()) {
+         double x = it.next();
+         int j = it.index();
+         it.set(function.evaluate(j, x));
+     }
+     return result;
  }
```

Código Fonte 4.6: Method inlined alongside inner extra edits

Moreover, the *Additional Functionality* got even higher OR with the same significance level, having 44% increase on its count. As for the *Statement Parent Change* and the *Condition Expression Change*, they got respectively the second highest and the second smallest OR, 45% and 28%.

Other Refactorings

Aside from the models previously mentioned, four other refactorings had significant models. Two models for the *Move Attribute* refactoring got statistical significance. However, the values were fairly low, 8% for the *Statement Delete*, and only 4% for the *Statement Insert*. The *Pull Up Method* had significant values for two inner-method edits, the *Statement Insert* and the *Statement Delete*, both with approximately 28% increase, and two outer-method edits, the *Additional Functionality* and the *Additional Object State*, with respectively 32% and 40%.

Another refactoring with significant models is the *Pull Up Attribute*, which, even with its low frequency (see Figure 4.1 and Figure 4.2), got extremely high ORs, from which, for each refactoring performed, the number of *Statement Insert* edits increased in 80%, while the number of *Statement Delete* edits doubled. The extreme case is the *Statement Parent Change*, presenting the highest OR among all models, having an increase of almost 150% for each *Pull Up Attribute* applied.

The last refactoring to get a significant model is the *Extract Superclass*. However, unlike all other significant models, for each *Extract Superclass* there has been a decrease in the number of the extra edit. For each occurrence of the refactoring, it is expected a decrease of almost half of the *Statement Insert* edit. We believe that it happened because of its high level. This goes along with Silva et al. [45], whose result showed that this refactoring is usually applied to group certain features that can be shared by subclasses.

Moreover, the remaining refactoring, named *Move Class, Rename Class, Push Down Attribute, Push Down Method* and *Extract Interface*, did not achieved statistical significance for any of the extra edits under analysis with the Negative Binomial modeling method.

4.5 Threats to Validity

Construct Validity: Our study relies on the data extraction strategy presented in Chapter 3, and, therefore, inherits the limitations we address in its *Limitations* section (see Subsection 3.2).

Conclusion Validity: To assess the relationship between the refactorings and extra edits, we relied on Hurdle Negative Binomial models, which require a high number of observations. However, we used sample sizes bigger than the ones recommended by Peduzzi [38] for logistic regressions.

External Validity: Our study was restricted to Java-based, Maven projects, hosted on GitHub. Thus, we cannot generalize our findings beyond the projects used. However, since we used projects from different sizes and contexts, and we worked with a great number of projects and commits, we believe our results are representative for understanding how developers combine extra edits with the refactorings under analysis. Moreover, the commits found with refactorings were the ones detected by RefDiff. Other refactorings might remain undetected. However, our conclusions were derived from the refactorings found in the process, and we do not make any assumptions regarding any other refactoring edit. Finally, commits that did not compile were discarded since SafeRefactor requires compiled versions of a program. For multi-module projects, the behavioral change check was based on the modules that compiled.

4.6 Concluding Remarks

In this chapter, we presented an empirical study carried out to better understand what extra edits developers apply when they floss-refactor their code. We used our strategy for floss-refactoring data extraction, presented in Chapter 3 to extract the data from 45 open-source Java projects hosted on GitHub. We analysed the floss-refactoring information through Hurdle Negative Binomial modeling, which, even though is often used for prediction purposes, can be used for describing the relationship between two variables. As result, we found evidences that there are certain patterns for the appearance of extra edits depending on the refactorings applied in the commit. There was an increase on the odds of an extra edit appear and/or increase its count for most refactoring under analysis. Interestingly, there was only one OR under one, the *Statement Insert* for the *Extract Superclass*. When an *Extract Superclass* is performed, is expected a decrease in the number of statements introduced in the system. In the next chapter, we discuss about how these extra edits are performed.

Chapter 5

How Extra Edits Are Performed?

In order to get a more thorough analysis, we investigate how the extra edits are performed from two different perspectives: the proportion of extra edits inside refactored entities, and the type of entities that are more likely to be modified. In this analysis, we used the dataset collected with our approach during the empirical study.

5.1 How frequent extra edits are performed inside refactored entities?

So far, we have discussed about the extra edits that appears along with different refactorings. However, how frequent extra edits appears inside refactored entities is still unclear. Therefore, we analysed our data set to find out the proportion of extra edits inside the refactored entities.

To better understand these edits, consider the code presented in Listing 5.1. In the example, we have a code that prints the status on a screen depending on the grade provided as parameter. To improve readability, a *Rename Method* refactoring has been applied, where the method *checkGrade* is now called *gradeCheck* (lines 11-12). However, a series of extra edits were also applied in both methods, *printStatus* and *gradeCheck*. The output string from the *printStatus* was updated (lines 5-6), as well as the *Return Statement* from the *gradeCheck* (lines 13-14). In this analysis we refer to edits inside refactored entities, i.e. the *Return Statement* updated inside the *gradeCheck*, since it was the only method refactored in the example.

```
  public class C{
      public void printStatus(int grade){
-         if(checkGrade(grade))
+         if(gradeCheck(grade))
-             System.out.println("Approved");
+             System.out.println("You have been approved");
          else
              System.out.println("Failed");
      }

-     public boolean checkGrade(int grade){
+     public boolean gradeCheck(int grade){
-         return grade>5;
+         return grade>6;
      }
  }
```

Código Fonte 5.1: Inner Refactoring Example

The graph in Figure 5.1 shows the proportion of the edits performed inside refactored entities. In our data set, we have a total of 42.532 extra edit observations (see Table 4.1). From these, a total of 6.144 edits were performed inside an entity under refactoring activity, accounting 14,4%.

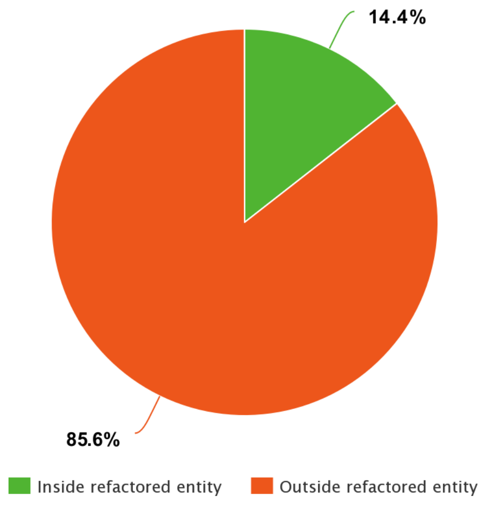

Figure 5.1: Proportion of extra edits inside refactored entities and extra edit outside refactored entities.

5.2 What type of entities are modified more frequently?

We also analyzed what type of entities were often modified in the edits performed inside the refactored entities. For instance, considering that the *Statement Insert* edit had a 10% increase for each *Extract Method* occurrence, we analyzed the type of those statements (e.g., a *Variable Declaration Statement*, an *If Statement*, a *Method Invocation*) when performed inside refactored entities. With this in mind, for each refactoring and extra edit type, we considered commits with both, refactorings and extra edits from the types under analysis. In addition, we considered the *Statement Update* edit and the *Condition Expression Change* as a single extra edit type, once the *Condition Expression Change* edits are conditional statements that were updated.

Moreover, in this analysis, we considered just the five most frequent entity types which, interestingly, were the same for all inner method edits: (*Method Invocation*, *If Statement*, *Variable Declaration Statement*, *Assignment* and *Return Statement*). Figure 5.2 shows the frequency of the entity types selected for all refactorings under analysis.

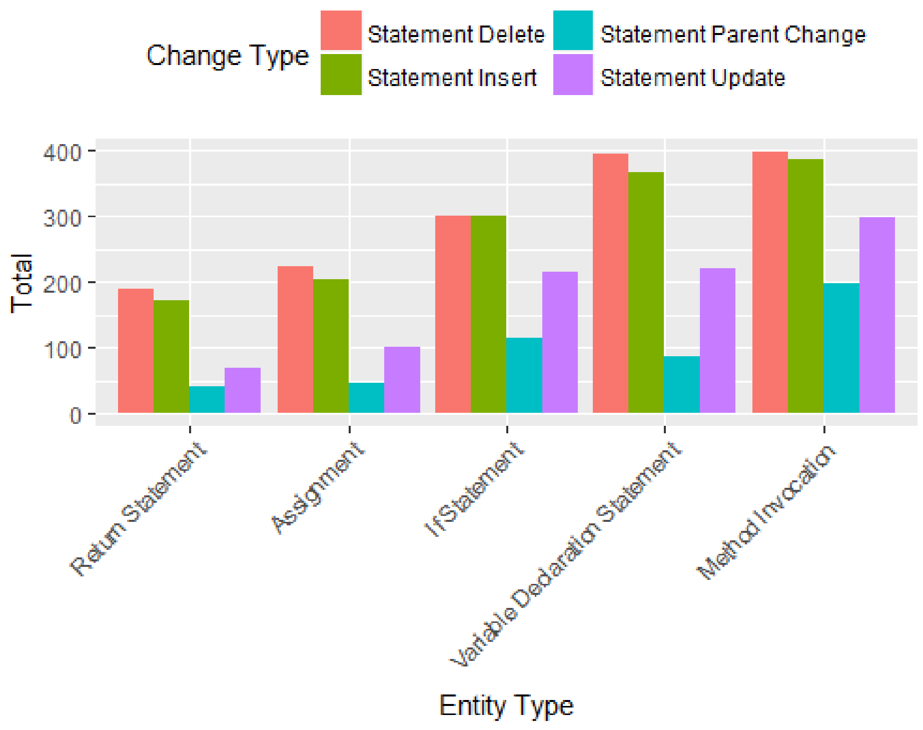

Figure 5.2: Entity type frequency for each change type.

As we can see in Figure 5.2, there seems to be a pattern, in which *Method Invocation* is the most frequent type and the *Return Statement* is the less frequent. The exception was the *Statement Parent Change*, once the *If Statement* appeared more frequently than *Variable Declaration Statement*. It is important to notice that in cases where a statement is composed by multiple types, the entity is classified based on the highest type level. For instance, the statement in Figure 2.4(b) - line 5 is classified as a *Variable Declaration Statement*, even though it is also composed by an *Assignment* and a *Method Invocation*.

To analyze the relationship between the refactoring and the entity types, again, we built a new set of Hurdle Negative Binomial models, but none of the Logit models achieved statistical significance. We believe this happened because, once we filtered the data by the extra edit type, the second process assumed by the Hurdle assumption became irrelevant. This filter is enough to change data distribution because, by selecting the commits with both specific edits, we reduced the number of zeros. Therefore, we used simple Negative Binomial regression. Table 5.1 shows the OR values for each entity and its respective edit type. The significant values (considering a significance level of $\alpha = 0.5\%$) are highlighted with " * ", following the same pattern of Table 4.3 and Table 4.4.

As we can see, the majority of the ORs for the *Extract Method* and the *Rename Method* achieved statistical significance, while the remaining refactoring had only one or two significant values. The *N/A* presented in Table 5.1 appeared due to its low frequency.

5.2.1 Extract Method

As we can see, all but two ORs were significant, only the *Return Statement* for the *Statement Update* and the *Assignment* entity type from *Statement Parent Change* was not significant. For the *Statement Parent Change*, the entity with the highest OR was the *Return Statement*, also presenting the highest significance. For each *Extract Method* performed in a commit, it is expected approximately 10% increase in its frequency, while the others significant entity types increased approximately 5%.

Among the entities in *Statement Insert*, the *Method Invocation* presented the highest OR, having approximately 13% more occurrences for each refactoring. The *Variable Declaration Statement* got the second highest OR, with 11%. The statement in line 5 from Listing 4.2 is an example of a *Variable Declaration Statement* inserted during this refactoring. In the same listing, we also have another example of *Statement Insert*, but in the form of *If Statement*,

Table 5.1: Relation Between Refactorings and Entity Types

Extract Method				
Entity Type	Statement Insert	Statement Delete	Statement Update	Statement Parent Change
Method Invocation	1.126139 ***	1.168150 ***	1.122285 ***	1.053073 *
If Statement	1.064817 ***	1.101383 ***	1.076528 ***	1.054566 *
Variable Decl. Statement	1.110671 ***	1.125572 ***	1.054875 ***	1.047229 *
Assignment	1.108629 ***	1.107649 ***	1.118541 **	1.041572
Return Statement	1.072907 ***	1.039767 **	1.048771	1.095825 ***
Rename Method				
Entity Type	Statement Insert	Statement Delete	Statement Update	Statement Parent Change
Method Invocation	1.226809 **	1.244084 **	1.137083 *	1.131139 **
If Statement	1.127213 *	1.213007 **	1.142504 *	1.106598
Variable Decl. Statement	1.246723 ***	1.223011 ***	1.124213 *	1.150811 **
Assignment	1.104380	1.137999 *	1.151105	0.962361
Return Statement	1.123182 **	1.092416 *	1.227473 *	1.042355
Move Method				
Entity Type	Statement Insert	Statement Delete	Statement Update	Statement Parent Change
Method Invocation	1.043613	1.052229	1.016353	1.023126
If Statement	1.021643	1.048998	1.020157	1.029284
Variable Decl. Statement	1.036153	1.044595	1.003620	1.025575
Assignment	1.062202 **	1.039351	1.043804	N/A
Return Statement	1.011065	1.015062	0.988781	N/A
Inline Method				
Entity Type	Statement Insert	Statement Delete	Statement Update	Statement Parent Change
Method Invocation	1.044890	1.052993	1.016713	1.014462
If Statement	1.028301	1.042696	1.027081	N/A
Variable Decl. Statement	1.043437 *	1.039709	1.008257	N/A
Assignment	1.056442 *	1.031887	N/A	N/A
Return Statement	1.024384	1.019813	N/A	N/A
Move Attribute				
Entity Type	Statement Insert	Statement Delete	Statement Update	Statement Parent Change
Method Invocation	1.117599	1.187635	1.071990	1.028088
If Statement	0.994708	1.137096	0.975798	1.039786
Variable Decl. Statement	1.150541	1.162600	1.077430	1.062847
Assignment	1.111982	1.019055	1.195739 *	1.053810
Return Statement	0.943340	1.045468	1.002026	1.126560

which appeared 6% more often for each refactoring. The *Assignment* and *Return Statement* entities presented an increase on its frequency of 11% and 7%, respectively.

For the *Statement Delete*, the *Method Invocation* entity is the most likely to be deleted, it occurred approximately 17% more when an *Extract Method* is performed, which was the highest OR from the entity types, regarding the *Extract Method* refactoring. Listing 5.2 shows a code fragment adapted from the *handleProcessingInstruction* method in the *Thymeleaf* project[1], which contains a series of statements that where deleted along with an *Extract Method*, including a *Method Invocation* statement (line 7).

```
 1   public void handleProcessingInstruction(final
 2       IProcessingInstruction iprocessingInstruction) {
 3    ...
 4 -   if (this.execLevel >= 0 &&
 5 -       gatheringType != GatheringType.NONE &&
 6 -       modelLevel >= gatheringModelLevel) {
 7 -     gatheringModel.add(iprocessingInstruction);
 8     }
 9    ...
10    }
```

Código Fonte 5.2: Statements deleted along with an Extract Method.

In Listing 5.2, we can also see an example of an *If Statement* that was also deleted in this commit (Lines 4-6), which is 10% more common when *Extract Method* is applied. The *Variable Declaration Statement* and the *Assignment* entities, appeared approximately 12% and 11% more often, respectively.

The *Return Statement* got the lowest OR, only 4% increase for each *Extract Method*. By analyzing the *Statement Update*[2], we found an increase of approximately 12% for the *Method Invocation* and the *Assignment*. For the *If Statement* entity the increase was 8%, while it was 5% for the *Variable Declaration Statement*.

In practice, these results could be useful for static analysis tools, who would have a starting point during analysis. Most of these edits could introduce unexpected behavior in the system, specially the *If Statement*, whose impact could easily pass unnoticed, since it could change the system behavior without throwing any exception, not even a *RuntimeException*.

[1]https://github.com/thymeleaf/thymeleaf

[2]If the type of entity changes during the *Statement Update*, the type reported for the analysis is the type before the edit is applied

5.2.2 Rename Method

For the *Rename Method*, 75% of the built models achieved significant values. For the *Statement Parent Change*, only two entities got significant ORs, the *Method Invocation* and the *Variable Declaration Statement* with respectively 13% and 15%.

For the *Statement Insert*, the *Variable Declaration Statement* entity was the most likely to be introduced, occurring approximately 25% more when an *Rename Method* is performed, which was also the highest OR from the entity types, regarding the *Rename Method* refactoring. The second highest OR was the *Method Invocation*, with 22%, followed by the *If Statement* and *Return Statement*, both with approximately 12% increase.

In Listing 5.3, we have another example[3] from the *Thymeleaf* project[4], where a *Rename Method* was carried out along with extra edits. In the code fragment, we can see a *Method Invocation* introduced in the system, which is not part of any refactoring reported by *RefDiff*. The method in called in the statement is responsible for validating a boolean expression provided as parameter. This change, like a *Condition Expression Change*, can impact directly the system behavior, since this new method call can prevent the rest of the method from being executed.

```
- public ParsedFragmentMarkup parseTemplateFragment([parameters...]]) {
+ private ParsedFragmentMarkup parseFragment([parameters...]]) {
      Validate.notNull(configuration, "Engine Configuration cannot be null");
      Validate.notNull(context, "Context cannot be null");
      Validate.notNull(template, "Template cannot be null");
+      Validate.isTrue(fragment == null || forcedTemplateMode != null,
                "When a textual fragment is specified, (forced)
                template mode must be specified too");
      ...
  }
```

Código Fonte 5.3: Statement inserted along with a Rename Method.

From the five entities under analysis, four entities got significant increase for *Statement Update* during *Rename Method*. The *Return Statement* had the highest OR, with 23% increase. The code present in Listing 4.5 has an example of a *Return Statement* updated during a *Rename Method* (Lines 5 and 9). In this case, the author changed the return from an empty

[3]Adapted for learning purposes.

[4]https://github.com/thymeleaf/thymeleaf

string to a *null* point, which, as explained before, can break the system if not carefully reviewed. When compiling the system, no error is generated, but if one caller is not prepared for handling *null* points, it can throw a *RuntimeException*.

In the same listing, we also have an example of an *If Statement* that was updated, this entity appeared 14% more often for each refactoring. In the example, the developer inverted the condition's semantic and replaced a simple boolean operation with a more complex method call. By inverting the semantics, it was necessary to change the code structure, moving statements to different places. Without proper care, this change could completely change the methods behavior. Moreover, the *Method Invocation* and the *Variable Declaration Statement* got 14% and 12% increase, respectively.

As for the *Statement Delete* edit, it was the only one to achieve significance values for all entity types, from which the *Method Invocation* achieved the highest OR, having an increase of 24%. It was also the second highest OR from all ORs, regarding the *Rename Method*. In the sequence, we have the *Variable Declaration Statement* and the *If Statement*, with 22% and 21% respectively. Finally, we have the *Assignment* and the *Return Statement*, with 14% and 9% respectively, which was the smallest OR for the *Rename Method* refactoring.

5.2.3 Other Refactorings

Besides the two refactorings previously mentioned, only four more models achieved statistically significant ORs, for three different refactorings. For the *Move Method* refactoring, is expected an increase of 6% in the number of *Assignment* introduced in the code. The *Inline Method* refactoring had significant values for two models, the *Variable Declaration Statement* and the *Assignment*, both for the *Statement Insert* edit, with 4% and 6% increase, respectively. An example of these edits was presented in Listing 4.6, where three *Variable Declaration Statements* appeared along with an *Inline Method* refactoring. As for the *Move Attribute*, it is expected approximately 20% increase in the number of *Assignments* updated.

5.3 Threats to Validity

Construct Validity: This investigation was carried out using the dataset built during the empirical investigation presented in Chapter 4, and, therefore, inherits the threats we addressed in its *Threats to Validity* section (see Subsection 4.5).

Conclusion Validity: To assess the relationship between the refactorings and entity types from different extra edits, we relied on Negative Binomial models, which require a high number of observations, but less than Hurdle Negative Binomial.

External Validity: Since we used the dataset from the empirical investigation previously reported (Chapter 4), the same threats to the external validity are applied to this analysis.

5.4 Concluding Remarks

In this chapter, we discussed about how the extra edits are performed during Floss-refactorings. We found that 14,4% of the extra edits were applied inside refactored entities. From these, the *Method Invocation, If Statement, Variable Declaration Statement, Assignment* and the *Return Statement* were the most frequent entities to be modified. Moreover, these indications can be used by new supporting tools as starting points to guide their analysis of faulty extra changes. In the next chapter, we discuss about research that is related to ours.

Chapter 6

Related Work

Due to the importance of refactoring during software development, several studies have tried to provide a better understanding of how this activity is performed in practice.

To assess the benefits and challenges from refactorings, Kim et al. [25] conducted a field study at Microsoft. The authors applied surveys and interviews with professional software engineers and performed analysis on version history data. They found that frequently refactored modules are less likely to have post-release defects. Moreover, the participants emphasized the need for a proper tool support for minimizing the costs and risks related to refactoring. This work shows how important refactorings are for developers and that this activity can still be improved.

Tsantalis et al. [53] proposed an approach and tool for detecting refactoring between two versions of a system based on AST matching. It innovates by not requiring the project to be built nor the definition of any threshold. They compared their tool against RefDiff [46] (the tool we used in our study) and achieved better accuracy. However, by the time we run our study Tsantalis et al.'s tool had not been released yet, and we were not able to analyse its applicability. We run a sample testing in our dataset for checking the impact of the use of RefDiff, which proved to be an effective refactoring-detection option in the context of our study.

Bavota et al. [2] run an empirical study that assessed when refactorings induce bugs. They identified refactoring edits and used the SZZ algorithm to determine whether classes involved in refactoring are more likely to induce faults. They found that while some refactoring are harmless, others tend to induce fault, especially those involving class hierarchies.

However, their study does not consider any extra edits. We believe extra edits are also responsible for introducing faults during refactoring.

Murphy-Hill et al. [33] conducted a study on four data sets with more than 140,000 refactorings and 3,400 version control commits. They found that developers often apply refactorings alongside other edits. Moreover, developers not only apply refactoring edits alongside extra edits but, according to Silva et al. [45], they are mainly driven by requirements changes.

Kim et al. [24] investigated API-level refactorings in three large open source projects to understand its role on software evolution. They found that the number of bug fixes increases after API-level refactorings, either because refactoring edits introduced new bugs, or they helped developers to identify and fix previous bugs. Due to frequent floss-refactoring mistakes observed, the authors pointed out the need for tools to support the safe application of refactoring and non-refactoring edits together.

In this sense, Coelho et al. [10] carried out a systematic literature mapping on refactoring-awareness during code review. Their study gathered information about existing support, research trends and possible research topics. As result, the authors' main finding emphasise the lack of appropriate characterisation studies, accurate support to change set with multiple refactorings types simultaneously, as well as the need for studies on the effectiveness of refactoring-aware solutions for code review.

In this context, Ge et al. [17] proposed a refactoring-aware code review tool that, depending on the use, can separates refactorings and non-refactoring modifications, allowing the reviewer to focus on one part at a time. It uses the developer's refactoring tool usage log file to identify automatic refactorings performed in the code and separate them from non-refactoring modifications. It also detects manual refactorings applied to the code. However, in this scenario, it cannot separate refactorings from extra edits.

On the other hand, Alves et al. [1] propose RefDistiller, a static analysis approach for inspecting manual refactorings that highlights edits that go further than what an automatic tool consider as pure-refactoring. It also points missing edits that could impact the system behavior, such as update all references to the refactored entity when necessary.

All four papers, Kim et al. [24], Coelho et al. [10], Ge et al. [17] and Alves et al. [1], emphasize how extra edits can be tricky and should be carefully considered when performing floss-refactorings.

Palomba et al. [37] performed an exploratory study on the relationship between refactoring and other change categories[1]. They classified each commit to the categories based on the commits' log and found that refactorings that focus on improving code maintainability and comprehensibility (e.g., *Add Parameter, Consolidate Duplicate Conditional Fragments, Move Field, Remove Assignment to Parameters, Replace Magic Number with Constant, Replace Nested Condition Guard Clauses*) are more likely to appear alongside fault repairing modifications. Refactorings that focus on improving code cohesion (e.g., *Add Parameter, Extract Method, Replace Data with Object*) are more likely to appear alongside feature introduction modifications. Finally, developers tend to use refactorings for improving code comprehensibility (e.g., *Introduce Explaining Variable*, *Rename Method*) during general maintenance modifications. However, the authors considered only commits categories, ignoring all code edits applied by the developers in their commits.

The works previously listed are essential to the field and provide substantial contributions. However, none of them focus on investigating the relationship between the refactorings and the extra on the edit-level. Important questions remained open, such as *what code edits should we expect in a commit when a refactoring is performed?*; and *where and what statements are more likely to change?*. Our empirical study focused on these topics since we believe that a more in-depth understanding about those relationships may help researchers to understand how developers perform code evolution in practice, and consequently guide efforts for developing novel strategies a more safer and systematic code edition.

[1]Fault Repairing, Feature Introduction, General Maintenance.

Chapter 7

Concluding Remarks

Refactoring is intended to improve software design while keeping its external behavior unchanged. However, developers often apply refactorings combined with extra edits, either intentionally or not. This interleaving of refactoring with extra edits is known as floss-refactoring. By understanding how extra edits are applied during floss-refactoring transformations, one can help the assessment of refactoring code review, by guiding what edits should be expected depending on the refactoring performed. It can also help other empirical investigations by providing a deeper understanding on how floss-refactorings are performed, allowing a more accurate modeling of the real world. However, little has been done on understanding the relationship between refactorings and extra edits during floss-refactoring.

Thus, in this dissertion work, we present a novel approach for floss-refactoring data extraction where detailed information about refactorings and non-refactoring edits are extracted from versioning history of Maven Java projects. Our approach is composed by three steps, in which we used a series of state-of-the-art tools for mining repositories and collecting non-pure-refactoring commits. Then, it decomposes the commits in fine-grained edits for analysis. As output, we have refactoring and extra edits details, such as type of edit applied, the entity that was changed, its location, among others.

Moreover, we used our approach to run an empirical study that investigated how refactoring and non-refactoring changes relate to each other. After mining 45 Java projects, we built a set of regression models to visualize the relationship between refactorings and a series of extra edits. Our results showed that the introduction of new methods is more common when an *Extract Method* or a *Rename Method* is performed, having an increase in its number for the *Extract Method, Rename Method, Move Method, Inline Method* and the *Pull Up Method* refactoring. As for the method removal, it is more common for the *Extract Method, Rename Method, Move Attribute* and *Inline Method* refactoring. These refactorings, except the *Move Attribute*, also increases the odds of a new attribute to be introduced. Moreover, inner method extra edits are very likely to appear, such as the update, removal and, mainly, the introduction of new statements. While its first appearance is only related to the *Extract Method*, there is an increase on its amount for a total of 8 refactoring types. Developers tend to change condition expressions in both, loops and decision structures, and moved statements to different structures during some refactorings. The only OR decrease was reported by the *Statement Insert* when a *Extract Superclass* is performed, which decreases almost half of the number of statements introduced.

A deeper analysis, showed that 14,4% of the extra edits were performed inside refactored entities. From these, the type of statements that are most frequently changed along with refactorings are *Method Invocation*, *If Statement*, *Variable Declaration*, *Assignment*, and *Return Statement*. However, the likelihood of the change in each specific type varies depending on the refactoring performed.

As future work, we intend to update our approach for dealing with characteristics that are currently incompatible. Currently, our approach is only capable of dealing with Maven projects, and we plan to extend our approach for dealing with other Java build automation tools, such as Ant and Gradle. Moreover, we plan to extend our analysis to a larger variety of systems, analyze other refactoring types and extra edit types, as well as investigate how the intention behind each refactoring can impact the results. We also plan to investigate the impact that each extra edit may generate during software evolution. Furthermore, we plan to develop a tool for preventing floss-refactoring faults based on the extra edits patterns found in this work. Finally, for helping new research works on this field, we intend to work on a mutation tool for automatically introducing faults based on the refactorings found in a given system.

Bibliography

[1] E. L. G. Alves, M. Song, T. Massoni, P. D. L. Machado, and M. Kim. Refactoring inspection support for manual refactoring edits. *IEEE Transactions on Software Engineering*, 44(4):365–383, April 2018.

[2] 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 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, SCAM '12, pages 104–113, Washington, DC, USA, 2012. IEEE Computer Society.

[3] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[4] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[5] Peter Bodík, Moises Goldszmidt, and Armando Fox. Hilighter: Automatically building robust signatures of performance behavior for small-and large-scale systems. In *SysML*, 2008.

[6] Victor R Basili1 Gianluigi Caldiera and H Dieter Rombach. The goal question metric approach. *Encyclopedia of software engineering*, pages 528–532, 1994.

[7] Chapin. Do we know what preventive maintenance is? In *Proceedings 2000 International Conference on Software Maintenance*, pages 15–17, Oct 2000.

[8] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of*

the 1996 ACM SIGMOD International Conference on Management of Data, SIGMOD '96, pages 493–504, New York, NY, USA, 1996. ACM.

[9] Z. Chen, M. Mohanavilasam, Y. Kwon, and M. Song. Tool support for managing clone refactorings to facilitate code review in evolving software. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 288–297, July 2017.

[10] Flávia Coelho, Tiago Massoni, and Everton Alves. Refactoring-aware code review: A systematic mapping study. In *Proceedings of 3rd International Workshop on Refactoring*, 2019.

[11] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *Proc. of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 404–428, Berlin, Heidelberg, 2006. Springer-Verlag.

[12] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, Nov 2007.

[13] Beat Fluri and Harald C. Gall. Classifying change types for qualifying change couplings. In *Proc. of the 14th IEEE Int. Conference on Program Comprehension*, ICPC '06, pages 35–45, Washington, DC, USA, 2006. IEEE Computer Society.

[14] S. R. Foster, W. G. Griswold, and S. Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *2012 34th Int. Conference on Software Engineering (ICSE)*, pages 222–232, June 2012.

[15] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[16] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 190–198, Nov 1998.

[17] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 211–221, June 2012.

[18] X. Ge, S. Sarkar, J. Witschey, and E. Murphy-Hill. Refactoring-aware code review. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 71–79, Oct 2017.

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

[20] Joseph M. Hilbe. *Negative Binomial Regression*, pages 126–161. Cambridge University Press, 2014.

[21] Joseph M. Hilbe. *Problems with Zeros*, pages 172–209. Cambridge University Press, 2014.

[22] Joseph M. Hilbe. *Varieties of Count Data*, page 1–34. Cambridge University Press, 2014.

[23] M. Kaya, S. Conley, Z. S. Othman, and A. Varol. Effective software refactoring process. In *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, pages 1–6, March 2018.

[24] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 151–160, New York, NY, USA, 2011. ACM.

[25] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 50:1–50:11, New York, NY, USA, 2012. ACM.

[26] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[27] Vincent Massol and Timothy M O'Brien. *Maven: A Developer's Notebook: A Developer's Notebook.* " O'Reilly Media, Inc.", 2005.

[28] Nikolaos Mittas, Makrina Viola Kosti, Vasiliki Argyropoulou, and Lefteris Angelis. Modeling the relationship between software effort and size using deming regression. In *Proc. of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 7:1–7:10, New York, NY, USA, 2010. ACM.

[29] M. Mongiovi, G. Mendes, R. Gheyi, G. Soares, and M. Ribeiro. Scaling testing of refactoring engines. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 371–380, Sep. 2014.

[30] 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.

[31] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In *Balancing Agility and Formalism in Software Engineering*, pages 252–266. Springer, 2008.

[32] Gail C Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the elipse ide? *IEEE software*, 23(4):76–83, 2006.

[33] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[34] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. A comparative study of manual and automated refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 552–576, Berlin, Heidelberg, 2013. Springer-Verlag.

[35] William F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

[36] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[37] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. An exploratory study on the relationship between changes and refactoring. In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, pages 176–185, Piscataway, NJ, USA, 2017. IEEE Press.

[38] Peter Peduzzi, John Concato, Elizabeth Kemper, Theodore R Holford, and Alvan R Feinstein. A simulation study of the number of events per variable in logistic regression analysis. *Journal of clinical epidemiology*, 49(12):1373–1379, 1996.

[39] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, Sep. 2010.

[40] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. On the relation of refactorings and software defect prediction. In *Proc. of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 35–38, New York, NY, USA, 2008. ACM.

[41] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, Champaign, IL, USA, 1999. AAI9944985.

[42] Ana Rodriguez. Reducing energy consumption of resource-intensive scientific mobile applications via code refactoring. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 475–476, Piscataway, NJ, USA, 2017. IEEE Press.

[43] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[44] Weiyi Shang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Automated detection of performance regressions using regression models on clustered performance

counters. In *Proc. of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 15–26, New York, NY, USA, 2015. ACM.

[45] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 858–870, New York, NY, USA, 2016. ACM.

[46] Danilo Silva and Marco Tulio Valente. Refdiff: Detecting refactorings in version histories. In *Proc. of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 269–279, Piscataway, NJ, USA, 2017. IEEE Press.

[47] G. Soares, B. Catao, C. Varjao, S. Aguiar, R. Gheyi, and T. Massoni. Analyzing refactorings on software repositories. In *2011 25th Brazilian Symposium on Software Engineering*, pages 164–173, Sept 2011.

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

[49] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, 2013.

[50] Quinten David Soetens, Javier Perez, and Serge Demeyer. An initial investigation into change-based reconstruction of floss-refactorings. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 384–387. IEEE, 2013.

[51] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, USA, 9th edition, 2010.

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

[53] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proc.*

of the 40th Int. Conference on Software Engineering, ICSE '18, pages 483–494, New York, NY, USA, 2018. ACM.

[54] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4):e1838. e1838 smr.1838.

[55] 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 34th International Conference on Software Engineering*, ICSE '12, pages 233–243, Piscataway, NJ, USA, 2012. IEEE Press.

[56] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, Sep. 2004.

[57] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.