Isabella Vieira Ferreira

Assessing the Bug-Proneness of Refactored Code: Longitudinal Multi-Project Studies

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informática da PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Alessandro Fabricio Garcia

Rio de Janeiro
July 2018

Isabella Vieira Ferreira

Assessing the Bug-Proneness of Refactored Code: Longitudinal Multi-Project Studies

Dissertation presented to the Programa de Pós–graduação em Informática da PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the undersigned Examination Committee.

Rio de Janeiro, July 16th, 2018

Isabella Vieira Ferreira

Isabella Vieira Ferreira joined PUC-Rio in 2016 as a Master student in Software Engineering in the Informatics Department. Isabella obtained a BSc. Degree in Computer Science (2016) from the Federal University of São João del-Rei (UFSJ). During her undergraduate degree, she did an exchange program (2013-2014) at the University of Ottawa (uOttawa) - Canada. Her main research interests are refactorings, bugs, code smells, and software maintenance and evolution.

Bibliographic data

Vieira Ferreira, Isabella

Assessing the Bug-Proneness of Refactored Code: Longitudinal Multi-Project Studies / Isabella Vieira Ferreira; advisor: Alessandro Fabricio Garcia. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2018.

v., 90 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Degradação Estrutural do Código-Fonte;. 2. Refatoração;. 3. Propensão a Bugs;. 4. Manutenção de Software;. 5. Estudo Empírico.. I. Garcia, Alessandro. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

First and foremost, I would like to thank the almighty God for giving me strength, knowledge, and courage to move on. Without his blessings, this achievement would not have been possible.

My deepest gratitude goes to my whole family: my father Luciano Ferreira, my mother Carla M. B. V. Ferreira, and my sister Ana Clara V. Ferreira. Thanks for giving me the best opportunities, for never letting me give up, for believing in my dreams, and for supporting me in both sunny and stormy days. Without my family, I would not have come this far.

I would like to thank my beloved boyfriend, Marcos Paulo C. Rocha, for providing me with unfailing support and continuous encouragement throughout. Without him, this accomplishment would not have been possible.

A very special gratitude goes to my advisor Alessandro Garcia. I would like to thank professor Alessandro for encouraging me through hard times, for his invaluable advice for both my research and my career, and for giving me innumerable opportunities. His energy and excitement about what he does motivate me.

My sincere thanks also go to the members of my thesis defense, Leonardo Murta, and Marcos Kalinowski. I would like to thank all professors from PUC-Rio for their contribution to my education. I am also very grateful for the opportunity to work at Apple Developer Academy - PUC-Rio. My sincere thanks to all my mentors and colleagues.

I extend my gratitude to my colleagues and friends from the OPUS Research Group. In particular, I would like to thank Diego Cedrim, Eduardo Fernandes, Leonardo Sousa, and Roberto Oliveira for all giving support and fellowship. A special thanks to my new friends Anderson Oliveira, Anderson Uchôa, Alexander López, Ana Carla Bibiano, Anne Benedicte Agbachi, Rafael de Mello, and Willian Oizumi.

I am very grateful for the collaboration made with the Federal University of Alagoas (UFAL). A special thanks to Baldoino Fonseca, Caio Barbosa, Daniel Tenório, Filipe Santos, Gabriel Nunes, Henrique Alves, João Lucas Correia, and Marcus Piancó.

Finally, I gratefully acknowledge the National Council for Scientific and Technological Development (CNPq), and PUC–Rio for the financial support.

Abstract

Vieira Ferreira, Isabella; Garcia, Alessandro (Advisor). **Assessing the Bug-Proneness of Refactored Code: Longitudinal Multi-Project Studies**. Rio de Janeiro, 2018. 90p. Dissertação de mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Programs often change along the system evolution, which implies an eventual code structure degradation. Recurring symptoms of such degradation are code smells. Studies suggest that the more frequently code smells affect a system, the higher becomes the bug-proneness of the code elements. To tackle code structural quality degradation, developers often apply refactorings on smelly program elements. However, applying refactorings might not suffice to reduce the bug-proneness of such degraded program elements. Previous empirical studies do not systematically analyze the bug-proneness of refactored code. Even though a recent study suggests that refactoring induces bugs frequently, the authors do not analyze to what extent refactored code is indeed closely related to the bug occurrence. Thus, in this dissertation, we conducted two longitudinal multi-project studies to assess the bug-proneness of refactored code. Our methodology aimed to address various limitations of previous studies. For instance, we have defined two complementary properties of the bug-proneness of refactored code, i.e., *frequency* and *distance*. While the former quantifies how often a refactored code is related to emerging bugs, the latter quantifies how close a bug emerges after a refactoring has been applied. The quantitative analysis of such properties was complemented by a manual analysis of refactorings closely related to the bug occurrence. Our first study aims at assessing the bug-proneness of code refactored through isolated refactorings, i.e., a single refactoring operation not performed in conjunction with other refactoring operations. This study reveals that 80% of the smelly elements that became buggy were not previously refactored. This result suggests the refactored code is much less bug-prone than non-refactored code. Moreover, in 75% of the times, a bug emerges in 7 changes far from the refactoring operation; this amount of changes usually corresponds to 3 months in the analyzed projects. Our second study aims at assessing the bug-proneness of code elements refactored through batch refactorings, i.e., a sequence of inter-related refactoring operations. Our results show that code refactored through batches is often more resilient to the introduction of bugs as compared to code refactored through isolated refactorings.

Keywords

Code Degradation; Code Refactoring; Bug-Proneness; Software Maintenance; Empirical Study.

Resumo

Vieira Ferreira, Isabella; Garcia, Alessandro. **Avaliando a Propensão a Bugs do Código Refatorado: Estudos Longitudinais Multiprojetos**. Rio de Janeiro, 2018. 90p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Os elementos de código geralmente mudam ao longo da evolução do sistema, o que implica em uma eventual degradação estrutural do código fonte. Sintomas recorrentes de tal degradação são chamados anomalias de código. Estudos sugerem que quanto mais anomalias de código afetam um sistema, mais alta se torna a propensão a bugs dos elementos de código. Para lidar com tal degradação da qualidade estrutural do código, desenvolvedores geralmente aplicam refatorações no código fonte. No entanto, aplicar refatorações pode não ser suficiente para reduzir a propensão a bugs dos elementos de código degradados. Um estudo recente sugere que refatorações induzem bugs frequentemente. No entanto, os autores não analisam se o código refatorado está, de fato, diretamente relacionado à introdução de bugs. Com isso, nesta dissertação, realizamos dois estudos longitudinais de múltiplos projetos para avaliar a propensão a bugs do código refatorado. Nossa metodologia teve como objetivo abordar várias limitações de estudos anteriores. Por exemplo, definimos duas propriedades complementares da propensão a bugs do código refatorado, sendo elas, *frequência* e *distância*. Enquanto a primeira propriedade quantifica a frequência com que um código refatorado está relacionado a bugs que emergiram no código fonte, a distância quantifica o quão próximo um bug surge depois que uma refatoração é aplicada. Nosso primeiro estudo tem como objetivo avaliar a propensão a bugs de refatorações isoladas. Primeiro, nossos resultados mostram que 80% dos elementos degradados que se tornaram bugs não foram previamente refatorados. Este resultado implica que um código refatorado é menos propenso a bugs do que um código não refatorado. Em segundo lugar, em 75% das vezes um bug surge depois de 7 mudanças feitas a partir da operação de refatoração, o que geralmente corresponde à 3 meses nos projetos analisados. Nosso segundo estudo tem como objetivo avaliar a propensão a bugs de refatorações em lote, ou seja, refatorações aplicadas em sequência. Nossos resultados mostram que, na maioria dos casos, o código refatorado em lotes é mais resiliente à introdução de bugs do que o código refatorado por meio de refatorações isoladas.

Palavras-chave

Degradação Estrutural do Código-Fonte; Refatoração; Propensão a Bugs; Manutenção de Software; Estudo Empírico.

Table of contents

1 Introduction **14**
1.1 The Bug-Proneness of Refactored Code: A Motivating Example 15
1.2 Problem Statement 17
1.3 Studies on Refactorings and Bugs 18
1.4 Proposed Approach and Evaluation 19
1.4.1 The Bug-Proneness of Code Refactored through Isolated Refactoring 21
1.4.2 The Bug-Proneness of Code Refactored through Batch Refactoring 23
1.5 Contributions 24
1.6 Dissertation Outline 25

2 Background and Related Work **26**
2.1 Basic Concepts 27
2.1.1 Code Degradation and Code Refactoring 27
2.1.2 Bugs and Bug-Proneness of Refactored Code 28
2.2 Literature Review 31
2.2.1 Studies on Code Degradation and Bugs 31
2.2.2 Studies on Refactorings and Bugs 32
2.3 Final Remarks 34

3 Research Methodology **35**
3.1 Research Goal 35
3.2 Software Projects Selection 36
3.3 Code Smell Detection 37
3.4 Refactoring Detection and Manual Validation 38
3.5 Bug Detection and Manual Validation 40
3.6 Bug-Fix Commit and Bug-Fix Elements Detection 42
3.7 Bug-Introducing Commit Detection 42
3.8 Changes Detection 43
3.9 Final Remarks 43

4 The Bug-Proneness of Code Refactored through Isolated Refactoring **44**
4.1 Study Settings and Procedures 44
4.1.1 Research Questions 45
4.1.2 Identification of Isolated Refactorings 48
4.1.3 Manual Classification of Refactoring Tactics 48
4.1.4 Measuring the Properties of the Bug-Proneness of Refactored Code 49
4.1.4.1 Measuring the *Frequency* Property 49
4.1.4.2 Measuring the *Distance* Property 50
4.1.5 Manual Validation of the *Distance* Results 51
4.2 The Bug-Proneness of Code Refactored through Isolated Refactorings 51
4.2.1 Frequency of Isolated Refactorings Performed in Code Elements that Are Bug-Prone 51
4.2.2 Bug-Proneness of Refactored Code *versus* Non-Refactored Code 54
4.2.3 Distance Between the Refactoring and the Bug 57

4.2.4 Bug-Proneness According to Refactoring Types 59
4.2.5 Bug-Proneness According to Refactoring Tactics 61
4.3 Threats to Validity 63
4.4 Final Remarks 64

5 The Bug-Proneness of Code Refactored through Batch Refactoring 66
5.1 Study Settings and Procedures 67
5.1.1 Research Questions 67
5.1.2 Identification of Batch Refactorings 69
5.1.3 Measuring the Properties of the Bug-Proneness of Refactored Code 70
5.1.4 Measuring the *Frequency* Property 70
5.1.5 Measuring the *Distance* Property 71
5.2 The Bug-Proneness of Code Refactored through Batch Refactoring 71
5.2.1 Frequency of Batch Refactorings Performed in Code Elements that Are Bug-Prone 71
5.2.2 Bug-Proneness of Refactored Code *versus* Non-Refactored Code 73
5.2.3 Distance Between the Refactoring and the Bug 75
5.3 Threats to Validity 77
5.4 Final Remarks 78

6 Conclusion and Future Work 80

Bibliography 84

A Published Papers 90

List of figures

Figure 1.1 Motivating example 16

Figure 2.1 Batch refactoring example 29

Figure 3.1 Study Phases 37

Figure 4.1 Evaluation of the bug-proneness of code refactored through isolated refactorings 49

Figure 4.2 Frequency of isolated refactorings according to the structure degradation of the code elements touched by these refactorings 53

Figure 4.3 Frequency of the bug-proneness of refactored code vs. non-refactored code for isolated refactorings 56

Figure 4.4 Distance of the most frequent refactoring types related to bugs 60

Figure 5.1 Evaluation of the bug-proneness of code refactored through batch refactoring 70

Figure 5.2 Frequency of the bug-proneness of refactored code vs. non-refactored code (batch refactorings) 74

List of tables

Table 1.1 List of specific research questions for the first study 21
Table 1.2 List of specific research questions for the second study 23

Table 2.1 Overview of previous studies *versus* our study 34

Table 3.1 General data of the analyzed software projects 38
Table 3.2 Analyzed Smell Types 39
Table 3.3 Analyzed Refactoring Types 40
Table 3.4 Refactored Elements 41
Table 3.5 Bug Patterns of FindBugs 42

Table 4.1 Study hypotheses derived from SRQ_2 46
Table 4.2 Study hypotheses derived from SRQ_3 47
Table 4.3 Study hypotheses derived from SRQ_5 48
Table 4.4 Frequency of isolated refactorings performed in code elements that are bug-prone per software project 52
Table 4.5 Frequency of the bug-proneness of refactored code vs. non refactored code per software project (isolated refactorings) 55
Table 4.6 Bug-proneness in distance for isolated refactorings considering bug reports 57
Table 4.7 Bug-proneness in distance for isolated refactorings considering bugs collected via static analysis 58
Table 4.8 Frequency of each refactoring type 60
Table 4.9 Bug-proneness of code refactored through root-canal refactoring 62
Table 4.10 Bug-proneness of code refactored through floss refactoring 62

Table 5.1 Study hypotheses derived from SRQ_7 68
Table 5.2 Study hypotheses derived from SRQ_8 68
Table 5.3 Frequency of bug-prone batch refactorings 72
Table 5.4 Frequency of the bug-proneness of refactored code vs. non refactored code per software project (batch refactorings) 74
Table 5.5 Bug-proneness in distance for batch refactorings 76

Sapere aude

Horace, *Epistularum liber primus.*

1
Introduction

Software maintenance requires the application of various changes to the source code. Undisciplined changes often lead to the degradation of the code structure quality (46, 50). Such degradation potentially hinders the software maintainability. One way to observe the code structure degradation is through *code smells*, which are microstructures in the program that represent symptoms of a structural problem. To address these symptoms, developers often apply code *refactorings* (6, 19). Code refactoring is a program transformation used for improving the structure of a program while preserving its observable behavior (19). Examples of commonly applied refactoring types include *Extract Method*, *Inline Method*, and *Move Method* (18).

Code refactoring is quite complex in practice. First, developers apply refactoring with different purposes (33, 34), such as reducing maintenance effort, facilitating feature additions, improving program testability, or even supporting bug fixes (33, 34, 35). Second, refactoring is very challenging (40, 57). That is, it is hard to perform refactoring on large code bases, which usually have many inter-component dependencies (40). Third, developers think that it is difficult to ensure program correctness after refactoring (40). In a study conducted at Microsoft, 76% of developers mentioned that refactoring comes with a risk of introducing bugs and functionality regression. Third, refactoring is applied with two significantly different tactics, namely root-canal refactoring and floss refactoring (18, 56). Developers apply *root-canal refactoring* when they aim to exclusively improve the code structure quality. On the contrary, developers apply *floss refactoring* when they aim to refactor the code together with non-structural changes as a means to reach other goals, such as adding features or fixing bugs.

Given the complexity of refactoring in practice, its application can impact positively or negatively on the *bug-proneness* of refactored code elements. Bug-proneness is the susceptibility of a code element containing a bug. A code element is *buggy* when it indeed contains a bug. A *bug* consists of a mistake made by the developer during the software development and maintenance, which violates the expected behavior of the software system (42, 58, 59). Then, every change in the program, including a refactoring operation, may induce the introduction of bugs in the modified code elements.

Hence, a refactored code element is *bug-prone* if it is susceptible to contain bugs after being refactored. In the case that the refactored code indeed contains a bug, we say that the code element became buggy. The susceptibility of a refactored code element may depend on particular characteristics observed on it. For instance, one could argue that some *refactoring types* make the refactored code element more bug-prone than other refactoring types. Additionally, depending on the *refactoring tactic* applied by the developer, the refactored code is more susceptible to contain bugs.

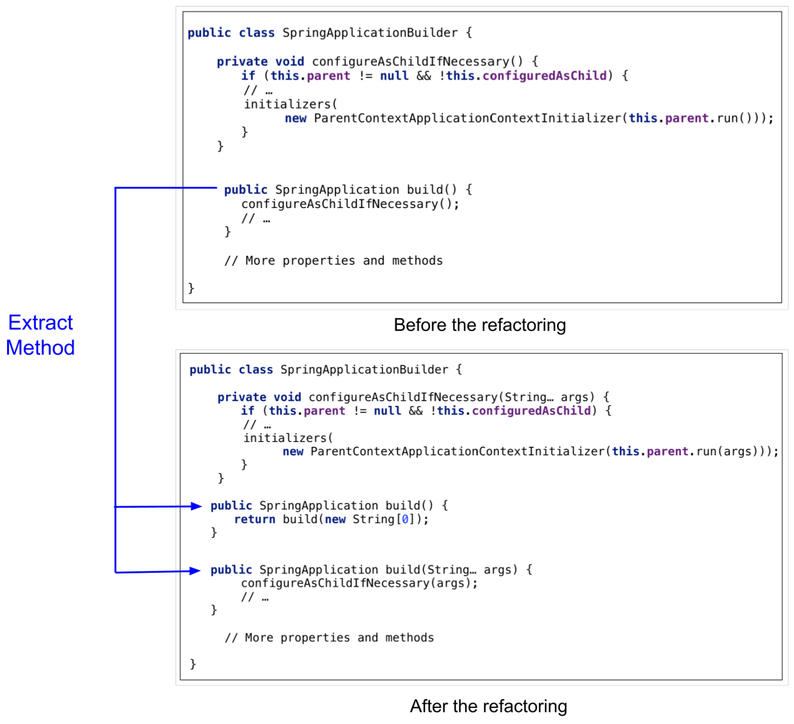
Unfortunately, the bug-proneness of refactored code is barely investigated in-depth. Only a few studies (1, 3) analyze the bug-proneness of refactored code. They provide evidence that refactored code is often susceptible to contain bugs. However, the provided evidence is not sufficient to blame refactorings for bugs. After analyzing their results, more relevant questions about the bug-proneness of refactored code still arise. For instance, it is unknown how long it takes to the refactored code element become buggy, and what are the characteristics of the bug-proneness of refactored code. With that in mind, we claim that it is necessary a more fine-grained and systematic analysis to better assess the bug-proneness of refactored code. Consequently, we will be able to confirm or refute if refactoring can actually be blamed for bugs, as previous studies suggest (1, 3).

1.1 The Bug-Proneness of Refactored Code: A Motivating Example

We motivate our study by illustrating to what extent refactored code could be bug-prone, as previous studies suggest (1, 3). This example is taken from the Spring Boot[1] project at GitHub. To provide this example, first, we analyzed the bug report #5103[2] entitled *command line args are not sent to parent SpringApplicationBuilder.configureAsChildIfNecessary in 1.3.2.* The description of the aforementioned bug report mentions that after upgrading Spring Boot from version 1.2.6 to 1.3.2, the configuration file is not loaded. The person who reported this bug suggested a way to fix it. That is, in version 1.3.2, there is a method called `configureAsChildIfNecessary` that calls `ParentContextApplicationContextInitializer` without passing the command line arguments. Thus, the solution would be to send the arguments to `configureAsChildIfNecessary`, and to `SpringApplicationBuilder.run` to bring back the behavior of version 1.2.6.

[1]Available at https://github.com/spring-projects/spring-boot.git
[2]Available at https://github.com/spring-projects/spring-boot/issues/5103

After understanding the reported bug, we identified the code elements involved with the fix of this bug. Then, we verified if these code elements had been refactored before the bug was reported. In this case, an *Extract Method* refactoring type was applied to the code element. This refactoring type is often applied when part of the code should be gathered in a single method (19). To solve this problem, developers create a new method with the extracted code (19). Figure 1.1 presents the source code before and after the refactoring. In this case, the developer extracted part of the `build` method (without any parameter) to another `build` method that receives the command line arguments as a parameter. The developer created the method `build` with different signatures to preserve backward compatibility of Spring Boot.



```
public class SpringApplicationBuilder {

    private void configureAsChildIfNecessary() {
       if (this.parent != null && !this.configuredAsChild) {
        // …
        initializers(
              new ParentContextApplicationContextInitializer(this.parent.run()));
       }
    }


     public SpringApplication build() {
       configureAsChildIfNecessary();
       // …
     }

     // More properties and methods

}
```

Before the refactoring

```
public class SpringApplicationBuilder {

    private void configureAsChildIfNecessary(String… args) {
       if (this.parent != null && !this.configuredAsChild) {
        // …
        initializers(
              new ParentContextApplicationContextInitializer(this.parent.run(args)));
       }
    }

   public SpringApplication build() {
      return build(new String[0]);
   }


   public SpringApplication build(String… args) {
       configureAsChildIfNecessary(args);
       // …
   }

     // More properties and methods

}
```

After the refactoring

Figure 1.1: Motivating example

In the example presented above, we can see that a floss refactoring was applied. In other words, the developer applied the Extract Method refactoring type for the purpose of fixing the bug described in the bug report identified as #5103. The analysis performed by previous studies (1, 3) is limited to assess whether a refactored code has a bug after being refactored. As a result, previous studies would conclude that refactored code is susceptible to contain bugs. However, as aforementioned, the refactoring was made to fix the bug. Therefore, the refactoring cannot be blamed for the bug.

Given the above circumstances, we claim that a more fine-grained and systematic analysis to better assess the bug-proneness of refactored code should be proposed. That is, one could say that if we measure how close or how far a refactored code element becomes buggy, we would have an insight of whether the refactored code is indeed bug-prone. In the case of the example presented above, as the bug was too close from the refactoring, one could argue that the refactoring either made the code element bug-prone or the refactoring was performed as a means to fix the bug. For that, only a manual analysis of the cases in which the refactoring is very close to the bug would actually give us an insight of whether the refactored code is bug-prone or not. As a result, we would see if refactoring can actually be blamed for bugs as previous studies suggest (1, 3). Furthermore, analyzing the bug-proneness of code elements refactored by different tactics would give us a better understanding of the bug-proneness of refactored code.

1.2 Problem Statement

There is an explicit assumption that code refactoring improves the structural quality of a program, thereby reducing the susceptibility of bugs in refactored code elements. However, this assumption might not always hold. Developers might unconsciously make the source code more susceptible to contain bugs depending on the complexity of certain refactorings. Kim et al. (40) found that there is no safe way of checking refactoring correctness, mainly when a regression test suit is insufficient. As a result, regression bugs might be introduced to the source code. Consequently, developers prefer to avoid refactoring the source code as a means to do not unintentionally introduce bugs (40). Furthermore, developers feel discouraged from applying refactorings due to the challenge of maintaining backward compatibility (40). Moreover, one could expect that when developers apply root-canal refactoring, i.e., when developers are intended at improving the code structure quality, the refactored code element is less susceptible to contain bugs than when developers apply floss refactoring. In the example presented in Section 1.1, the developer performed a floss refactoring to fix the bug. Furthermore, besides refactoring the source code to fix the bug, the developer was concerned about maintaining backward compatibility of Spring Boot. Thus, one could argue that a bug could have been introduced when the developer was performing such refactoring.

Unfortunately, there is limited understanding about the bug-proneness of refactored code. Existing studies (1, 3) provide evidence that refactored code is often susceptible to contain bugs. However, none of the existing studies (1, 3) analyze the characteristics of refactorings that are bug-prone. For instance, in the example presented in Section 1.1, the floss refactoring performed by the developer could have made the code element bug-prone if compared to when the developer applies a root-canal refactoring. Furthermore, some refactoring types might be more complex than others, therefore, increasing the susceptibility of bugs in the refactored code. The lack of knowledge about the characteristics of refactorings that make code elements bug-prone prevent researchers and practitioners from having better practices when applying refactorings. The general problem is described as follows.

General Problem. The characteristics of refactorings that make refactored code elements bug-prone remain unknown.

1.3 Studies on Refactorings and Bugs

Only a few studies have investigated the bug-proneness of refactored code (1, 3). Bavota et al. (1) analyze if refactoring induces bugs. In other words, the authors analyze if refactored code elements contain a bug in further commits. They focus on analyzing: (i) the percentage of refactored code elements that become buggy, and (ii) the refactoring types that are more likely to make the refactored code bug-prone. Their results show that refactored code is often bug-prone. Furthermore, they have found that some refactoring types are more harmful than others, such as refactorings involving hierarchies (e.g., Pull up Method). Conversely, Weißgerber and Diehl (3) investigate if the number of bug reports opened in the next five days after the refactoring increases or decreases. That is, the authors investigate whether the bug-proneness of refactored code increases or decreases according to the number of refactorings and opened bug reports. They found that a high ratio of refactorings is often followed by an increasing ratio of bug reports.

However, existing studies (1, 3) do not assess whether various recurring characteristics of refactorings make code elements bug-prone. For instance, the authors overlook the impact of different refactoring tactics on the bug-proneness of refactored code. The bug-proneness of code elements refactored by the different refactoring tactics might differ. Furthermore, the authors overlook the fact that developers very often apply a sequence of refactorings (also named batches) to improve the code structure quality (18, 40). Applying a sequence of refactorings might be more or less bug-prone if compared to the application of only one refactoring.

Moreover, existing studies (1, 3) performed coarse-grained analyses that might overshadow the results of the bug-proneness of refactored code. First, existing studies focus on analyzing the percentage of refactorings that make code elements bug-prone. For that, they only take into consideration if a refactored code contains at least one bug in further commits. This analysis might not suffice to measure the bug-proneness of refactored code because a bug might have emerged in the same code element many commits away from the refactoring. Second, they do not manually analyze if the refactoring indeed made the source code bug-prone in the analyzed software systems. As presented in the motivating example, a refactoring was performed to fix a bug. In this way, the refactoring should not be blamed for the bug. A manual analysis is, therefore, essential to guarantee the reliability of the results. Third, these studies only analyze a few software projects, and only consider major releases instead of making a more fine-grained analysis (commit by commit). As there are many changes between two major releases, many refactorings and bugs are hidden or unidentifiable in their analyses. Given the aforementioned limitations of previous studies, we propose a different approach for evaluating the bug-proneness of refactored code.

1.4
Proposed Approach and Evaluation

While existing studies (1, 3) focus on analyzing whether refactored code elements have bugs in further commits, we measure the bug-proneness of refactored code according to two complementary properties. These two properties serve to address various methodological limitations of previous studies (Section 1.3). First, the *frequency* in which degraded code elements that were refactored become buggy, and the *frequency* of refactorings performed in code elements that are bug-prone. That is, the more frequently degraded code elements are refactored and bug-prone, the higher the bug-proneness of refactored code elements. Similarly, the more frequently refactorings are performed in code elements that are bug-prone, the higher the bug-proneness of refactored code elements. Second, the *distance* in number of changes between the commit in which developers have applied the refactoring on degraded code elements, and the commit in which the bug emerged. A bug can emerge in two different commits. First, the commit in which the bug was indeed introduced in the code element. Second, the commit related to when a developer or a user opened a bug report. We consider both events of bugs, i.e., insert and report, due to the fact that a bug can be perceived and reported by developers or users after a long time that the bug was indeed introduced in the code element. Thus, the smaller the number of changes between the refactoring and the bug, the higher the bug-proneness of refactored code elements.

The frequency and distance properties allow us to better characterize how bug-prone is a refactored code. For instance, in the example presented in Section 1.1, the distance property would be equal to one. In other words, only one change was performed between the refactoring and the bug. In this case, the change is the refactoring itself. As the bug was too close from the refactoring, one could say that the refactoring is susceptible to contain bugs. In the cases that the bug is too close from the refactoring, a manual analysis is required to confirm if the refactoring can be blamed for the bug.

Besides proposing two complementary properties, in this dissertation, we divided the refactorings into two disjoint sets. The first set is composed of *batch refactorings* (or, simply, *batches*). Hence, if more than one refactoring is applied to the source code by the same developer, and those refactorings were applied to the same code element, then it is a batch. The second set is composed of *isolated refactorings*. In other words, if a refactoring is not part of a batch, then we call it an isolated refactoring. The goal to segregate the refactorings is to assess if refactored code elements are more or less bug-prone when developers apply multiple refactorings (batches) than when developers apply only one refactoring (isolated refactoring).

To assess the bug-proneness of refactored code regarding the aforementioned complementary properties, we propose to evaluate it in the context of degraded code elements. We focus on the analysis of degraded code elements because (i) developers apply different refactoring tactics, i.e., root-canal and floss refactorings, targeting at restructuring degraded code elements, and (ii) developers much more often choose to apply refactorings in degraded code elements (6, 32). We describe our study goal as follows: *analyze* refactorings applied to structurally degraded code; *for the purpose of* assessing the bug-proneness of refactored code; *with respect to* the frequency of refactored code that became buggy, the frequency of refactorings performed in code elements that are bug-prone, and the distance between the refactored code and the code element that contain bug; *from the point of view of* researchers; *in the context of* Java open source projects with bug reports available. In summary, we address a general research question described as follows.

General Research Question. What are the characteristics of refactorings that make refactored code elements bug-prone?

To reach our goal and answer our general research question, we performed two longitudinal multi-project studies. In the first study, we assessed the bug-proneness of code elements refactored through isolated refactorings. Furthermore, in the second study, we assessed the bug-proneness of code elements refactored through batch refactorings. Our studies involved 12 Java open source projects, 39,750 refactorings, including 21,217 isolated refactorings and 7,828 batch refactorings, 2,119 refactorings manually validated by refactoring tactic, 6,051 bug reports, and 49,250 bugs via static analysis. We describe each study with its the specific research questions, and its findings as follows.

1.4.1 The Bug-Proneness of Code Refactored through Isolated Refactoring

The goal of the first study is to assess the bug-proneness of code elements refactored through isolated refactorings. Table 1.1 presents the specific research questions (SRQ_s) of this study. First, we assess the complementary properties of the bug-proneness of refactored code, i.e., frequency and distance (SRQ_1, SRQ_2, SRQ_3). Then, we assess the different characteristics of the bug-proneness of code refactored through isolated refactoring, i.e., we investigate the bug-proneness regarding the refactoring types (SRQ_4), and the refactoring tactics (SRQ_5). Hereafter, we present a summary of the findings concerning each specific research question.

Table 1.1: List of specific research questions for the first study

$\mathbf{SRQ_s}$	**Description**
SRQ_1	How many isolated refactorings were performed in code elements that are bug-prone?
SRQ_2	Are refactored code elements less bug-prone than non-refactored code elements?
SRQ_3	How many times a degraded code element that was refactored has to change to become buggy?
SRQ_4	How frequent degraded code elements are bug-prone per refactoring type?
SRQ_5	How many times a degraded code element that was refactored has to change to become buggy per refactoring tactic?

Summary for SRQ_1. Only 5.38% of the isolated refactorings were performed in code elements that are bug-prone against 94.62% of isolated refactorings performed in code elements that are not bug-prone at all. Additionally, 62.71% of the refactorings performed in code elements that are not bug-prone touched in code elements without any code smell. Similarly, 63.14% of the refactorings performed in code elements that are bug-prone touched in code elements with either a single smell or multiple smells. This result shows that the refactoring might not have sufficed to fully overcome the degradation in the source code, and, as a result, bug(s) emerged in future commits.

Summary for SRQ$_2$. 79.67% of the smelly elements that became buggy were not previously refactored. This result shows that degraded, non-refactored code tends to be more bug-prone than degraded, refactored code.

Summary for SRQ$_3$. We analyzed the code elements that became buggy after the application of isolated refactorings. In 75% of these cases, at least seven additional changes were performed between the refactoring and the bug. Additionally, these seven changes were performed in approximately three months between the isolated refactoring and the bug. As a result, code elements refactored through isolated refactorings are often not susceptible to immediately contain bugs. Then, we manually analyzed the remaining 25% of the refactored code elements that became buggy; these elements represent the cases where the distance of refactoring to bugs was lower than 9 changes. Surprisingly, in these cases, we could not find any explicit case of refactored code that indeed induced the bug, as previous work suggests (1). On the contrary, we found that the changes performed between isolated refactorings and bugs are more likely to make code elements bug-prone.

Summary for SRQ$_4$. We analyzed which refactoring types (used in isolated refactorings) are often related to the occurrence of bugs. Bugs were more frequently found in code refactored by Extract Method and Inline Method. Furthermore, few changes are necessary after the application of these refactoring types so that the refactored code become buggy. Thus, developers should be aware when applying Extract Method and Inline Method.

Summary for SRQ$_5$. In 75% of the times that a code element affected by a floss refactoring became buggy, it is necessary at least eight changes after the floss refactoring so that the code element becomes buggy. Surprisingly, when considering 100% of the times that a code element affected by a root-canal refactoring becomes buggy, fewer changes are necessary after the application of a root-canal refactoring so that the code element becomes buggy compared to floss refactoring. Thus, root-canal refactoring is at least as bug-prone as floss refactoring.

1.4.2 The Bug-Proneness of Code Refactored through Batch Refactoring

The goal of the second study is to assess the bug-proneness of batch refactoring. After studying the bug-proneness of code refactored through isolated refactoring, there is a need to empirically investigate the bug-proneness of code elements refactored through batch refactoring. As refactoring might be risky as any other change in the source code, one could say that if more refactorings are applied to a code element, then more bug-prone the code element becomes. On the other hand, others could claim that the more a code element is refactored, the less it becomes susceptible to contain bugs. However, neither of these claims are tested since the literature does not assess the bug-proneness of code refactored through batch refactorings.

Thus, in this study, we investigate batches as one of the characteristics of refactorings. For that, we analyze the bug-proneness properties of code refactored through batch refactorings. Furthermore, we compare the results of this study with results of the study of isolated refactorings. The goal to compare the bug-proneness of code refactored through batch refactoring and isolated refactoring is to see if code refactored through batches are more bug-prone than code refactored through isolated refactoring. Table 1.2 presents the specific research questions (SRQ_s) of this study. We also present below the summary of each specific research question.

Table 1.2: List of specific research questions for the second study

SRQ_s	Description
SRQ_6	How many batch refactorings were performed in code elements that are bug-prone?
SRQ_7	Are code elements refactored through batch refactorings less bug-prone than non-refactored code elements
SRQ_8	Does the bug-proneness of degraded code elements decrease when applying batch refactoring?

Summary for SRQ_6. Our results show only 1.75% of the batch refactorings were performed in code elements that became buggy, against 98.25% of batch refactorings that were performed in code elements that not became buggy at all. By comparing these results with the ones found in RQ_1, we can see that code refactored through isolated refactorings is more bug-prone than code refactored through batch refactorings.

Summary for SRQ$_7$. 87.75% of the degraded code elements that became buggy were not previously refactored by a batch refactoring. On the contrary, only 12.25% of degraded code elements that became buggy were refactored through a batch refactoring. As a conclusion, a batch refactoring could have been applied to the degraded code elements in order to make them less bug-prone.

Summary for SRQ$_8$. In 75% of the times that code elements refactored through batch refactorings are bug-prone, a code element needs least eight changes after a batch refactoring so that the code element becomes buggy. However, when comparing isolated and batch refactoring, in 75% of the times that refactored code elements are bug-prone, it is necessary more changes so that the code element becomes buggy after a batch refactoring than after an isolated refactoring.

1.5 Contributions

This section summarizes the contributions of our studies on deriving new knowledge about the bug-proneness of refactored code. These contributions provide insights for both researchers and practitioners. We discuss each contribution as follows.

- This dissertation proposes two complementary properties of the bug-proneness of refactored code: frequency and distance. By assessing both properties, we were able to better characterize the bug-proneness of refactored code. As a result, we found that we cannot blame refactorings for bugs, as previous work suggest (1, 3).
- This dissertation also compares the findings of our empirical study with previous findings of another study (1). Our in-depth analyses put the investigation about the bug-proneness of refactored code from previous work (1, 3) into another perspective. That is, only analyzing if a refactored code has a bug in further commits is not enough to understand the relation between refactored code and bugs. It is also necessary to look to other characteristics of refactorings to reveal the circumstances that make refactored code bug-prone.
- Furthermore, the properties proposed in this dissertation allowed us to contradict previous study (1). For instance, Bavota et al. found that refactoring types related to hierarchy are the most harmful ones. However, we found that a refactored code is never bug-prone after the

application of these refactoring types. As a conclusion, refactoring types related to hierarchy cannot be directly blamed for bugs, as previous study suggests.

– Moreover, this dissertation investigates the effects of refactoring tactics and batch refactorings, which are unexplored by previous work (1, 3). Our results show that more changes are necessary after a batch refactoring (than after an isolated refactoring) so that the code element becomes buggy. Furthermore, surprisingly, root-canal refactoring is at least as bug-prone as floss refactoring.

– Finally, we provide recommendations for developers based on the characteristics found in the studies performed in this dissertation.

1.6 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 provides background information to help the readers understanding this dissertation as well as the literature review. Chapter 3 describes the study design, including the study goal. Chapter 4 discusses the first study conducted in this dissertation where we investigate the bug-proneness of code refactored through isolated refactorings. Chapter 5 discusses the second study where we investigate the bug-proneness of code refactored through batch refactorings. Finally, Chapter 6 concludes the dissertation with a summary of our contributions and suggestions for future research.

2
Background and Related Work

Code refactoring is a program transformation used to improve the structure of a program while preserving its observable behavior (19). Refactoring, in practice, consists of two significantly different tactics, namely root-canal and floss refactoring (18, 56). Developers apply *root-canal refactoring* when they aim to exclusively improve the code structure quality. On the contrary, developers apply *floss refactoring* when they aim to refactor the code together with non-structural changes as a means to reach other goals, such as adding features or fixing bugs. There is an explicit assumption that code refactoring improves the structural quality of a program, thereby reducing the likelihood of bugs on the refactored code elements. However, this assumption might not always hold. Developers might unconsciously make the source code more susceptible to contain bugs depending on the complexity of the refactoring.

Previous studies (1, 3) neither investigate the bug-proneness of refactored code in depth, nor the influence of the refactoring tactics on bugs. Only a few studies (1, 3) solely analyze the bug-proneness of refactored code. They provide evidence that refactored code is often susceptible to contain bugs. However, these studies (1, 3) are limited in only assessing whether the refactored code contains a bug in further commits. This analysis might not suffice to blame refactorings for bugs since the refactored code element could have become buggy very far in terms of commits from the refactoring.

To address the limitation of previous studies, this dissertation presents two longitudinal multi-project studies aimed at assessing the bug-proneness of refactored code. This chapter provides background information, i.e., the basic terminology which underlies all next chapters. The remainder of this chapter is organized as follows. Section 2.1 presents the basic concepts related to code degradation, refactoring, and bugs. In section 2.2 we discuss related work aimed at contextualizing our research from different perspectives in the literature. For that, we present a literature review about previous studies that are closely related to ours. Hence, we present studies that investigate the relationship between (i) code structural quality and bugs, and (ii) refactorings and bugs.

2.1 Basic Concepts

We introduce the basic concepts in two parts. First, we present concepts related to code degradation and refactoring. Second, we present the concepts related to bugs and the bug-proneness of refactored code.

2.1.1 Code Degradation and Code Refactoring

Code refactoring consists of transformations used for improving the structure of a program while preserving its observable behavior (19). Developers apply refactoring with different purposes (33, 34), such as removing design problems, reducing maintenance effort, facilitating feature additions, improving program testability, or even supporting bug fixes (33, 34, 35). Likewise, even though refactoring is intended to improve the code structure, developers still need to know in which code elements they should apply it. Previous work (6, 32) suggest that the presence of code smells should be used as a symptom of the need for applying refactoring. A *code smell* is a microstructure in the program that represents a symptom of a structural problem. It is expected that developers refactor their code as soon as code smells start to affect the code elements (38, 39). An example of a code smell is when a method is more interested in data from another class than the data in its own class (39). In this case, developers can apply a refactoring to move the method to the class in which it is interested.

Refactoring Type. It refers to the kind of transformation applied to one or more code elements. An example of a refactoring is when the developer moves a method from one class to another in order to remove excessive dependencies between classes. This refactoring type is called *Move Method*. There are other refactoring types with different purposes, such as extracting new code elements from excessively complex elements and managing class inheritances. Therefore, given a refactoring r, $type(r)$ is a function that returns the type of the refactoring r. We chose to study the 13 most commonly investigated refactoring types in the literature (18). These refactoring types are defined in the Fowler's catalog (19). We present a description of each refactoring type on Table 3.3 in Chapter 3.

Refactored Code Elements. In this dissertation, we consider as refactored elements all those directly affected by a refactoring operation. For instance, let us consider the *Move Method* refactoring. In this refactoring type, a method

m is moved from class A to B. Hence, the refactored elements, in this case, are $\{m, A, B\}$. All callers of m are indirectly affected by this refactoring, but we do not consider them as refactored elements due to the fact that the broader the scope of a refactoring, there are more chances to mislead the conclusions of the studies performed in this dissertation. Similar reasoning applies to the other refactoring types; thus, for each refactoring type, a different set of refactored elements is used. In this way, given a refactoring operation r, $el(r)$ is a function that returns the set of refactored elements of r. We present the refactored elements of each refactoring type on Table 3.4 in Chapter 3.

Refactoring Tactic. Developers apply two refactoring tactics: root-canal refactoring and floss refactoring (18, 56). Developers apply *root-canal refactoring* when they aim to exclusively improve the code structure quality. On the contrary, developers apply *floss refactoring* when they aim to refactor the code together with non-structural changes as a means to reach other goals, such as adding features or fixing bugs.

Isolated and Batch Refactoring. In the context of this dissertation, a sequence of refactorings is considered a batch refactoring (or, simply, a *batch*) if some constraints are satisfied. First, a batch must have more than one refactoring. Second, if $b = [r_1, r_2, \cdots, r_n]$ is a batch of size n, then $[r_1, r_2, \cdots, r_n]$ is a list of refactoring operations ordered chronologically. Third, all refactorings must have at least one code element in common (18, 40). Thus, $el(r_1) \cap el(r_2) \cap \cdots \cap r_n \neq \emptyset$. Finally, all refactorings in a batch must have been performed by the same developer. Figure 2.1 presents two examples of batches. The developer performed two refactorings in the *UserCtrl* class in the first commit. In this way, the batch b_1 is composed of $\{$*Extract Method*, *Inline Method*$\}$. The b_2 batch occurs in the class *User*. The developer applied two *Move Method* refactorings. In this way, $b_2 = \{$*Move Method*, *Move Method*$\}$. Besides the batch, a refactoring operation can be considered an *isolated refactoring*. That is, if a refactoring is not part of a batch, i.e., if a refactoring does not satisfy the aforementioned constraints, then this refactoring is an isolated refactoring. In the projects analyzed in this dissertation, we found 21,217 isolated refactorings and 7,828 batches.

2.1.2 Bugs and Bug-Proneness of Refactored Code

A bug consists of a mistake made by the developer during the software development and maintenance, which violates the expected behavior of the

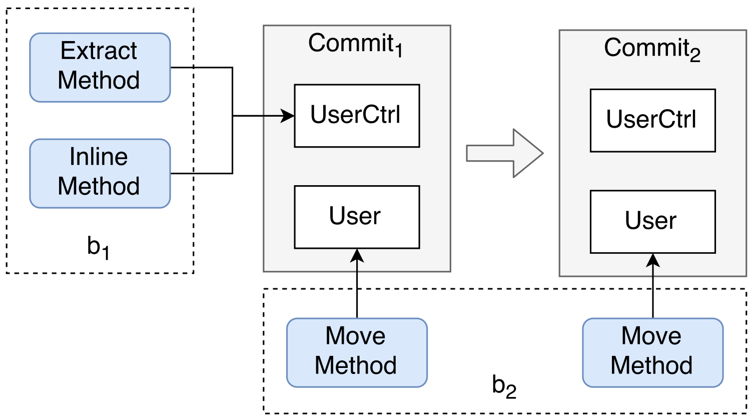


Figure 2.1: Batch refactoring example

software system (42, 58, 59). Software users and developers are encouraged to report the identified bugs through bug tracking systems, such as Bugzilla and Jira (42). Then, a bug report is created. A bug report is a detailed description of the failure (unexpected behavior observed by a user or a developer), and it occasionally has a hint at the location of the fault (the actual mistake in the program) in the code, e.g., patches or stack traces (63). Overall, bug reports are generated along the life cycle of a software system, and often accumulate enriching information about bugs that affect the system.

Another way of discovering bugs in software systems is by using static analysis tools. Such tools look for violations of programming practices without running the source code (64). In this study, we used FindBugs for this purpose, which is very popular tool (65). The tool implements several bug patterns that are commonly used to find a significant number of bugs in software projects (65). For study purposes, we focus on categories of bug patterns identified by the tool, which possibly relates to code refactoring (see Table 3.5). We aimed at choosing bug patterns that may affect the code structure as well as those that may be related to refactorings.

Bug-Introducing Changes and Buggy Code Elements. Developers change the source code to add or to remove functionalities, to refactor, or to fix a bug. Thus, they might be inadvertently introducing bugs through a change of the source code. This change is called *bug-introducing change*. Later on, the bug may manifest during the program execution when the unexpected behavior is observed by a user. This behavior is recorded in a bug tracking system (i.e., bug reports) as soon as it is revealed (12). Afterwards, developers perform a change in the source code to fix the bugs described in bug reports. When they commit this change, they mention in the commit message that the change was made to fix a bug. Thus, we call it *bug-fix change*. Thus, we define that b_e is a buggy element only if b_e is involved with the fix of a bug described in a bug report, or if b_e is considered buggy by a tool that detects bugs via static analysis.

Bug-Proneness of Refactored Code. Bug-proneness is the susceptibility of a code element containing a bug. Thus, a refactored code element is bug-prone if it is susceptible to contain a bug after being refactored. In the case that the refactored code indeed contains a bug, we say that the code element became buggy. The susceptibility of a refactored code element may depend on particular characteristics observed on it. For instance, one could argue that some refactoring types make the refactored code element more bug-prone than other refactoring types. Additionally, depending on the refactoring tactic applied by the developer, the refactored code is more susceptible to contain bugs. Furthermore, the bug-proneness of refactored code might differ if a developer applied an isolated refactoring or a batch refactoring.

We measure the bug-proneness of a code element by computing two properties: frequency and distance. First, the *frequency* in which degraded code elements that were refactored become buggy, and the *frequency* of refactorings performed in code elements that are bug-prone. In the first analysis, the frequency property can be measured by considering either when a degraded code element was refactored or not refactored. Hence, the more frequently degraded code elements become buggy, the higher the bug-proneness of the code element. Additionally, in the latter analysis, the frequency property can be measured by considering either when a refactoring was performed in a code element that is bug-prone or not bug-prone at all. Thus, the more frequently refactorings are performed in code elements that are bug-prone, the higher the bug-proneness of refactored code elements. Second, the *distance* in number of changes between the commit in which developers have applied the refactoring on degraded code elements (refactoring commit) and the commit of the bug. The commit of the bug can be (i) the commit in which the bug was introduced by a developer (bug-introduction commit), or (ii) the commit in which a user or a developer reported the bug (bug-report commit). We consider both events of bugs, i.e., bug-introduction commit and bug-report commit, due to the fact that a bug can be perceived and reported by developers or users after a long time that the bug was indeed introduced in the code element. Thus, the smaller the number of changes between the refactoring and the bug, the higher the bug-proneness of the code elements. The distance property enables us to understand how close or how far a refactored code element become buggy after being refactored.

Bug-Proneness of Code Refactored through Batch Refactoring. The bug-proneness of a code refactored through batch refactoring is measured in the same way as for an isolated refactoring. However, as a batch is composed of multiple refactorings, we consider only the first refactoring in the batch to compute the distance. Thus, given $batch = \{r_1, r_2, \cdots, r_n\}$, we say that the *distance* property will be computed from the first refactoring commit v_{r_1} (where r_1 is the first refactoring operation in the batch) to the commit in which the code element became buggy v_{b_e}.

2.2 Literature Review

This section shows how our study significantly differs from others while sharing some goals. Section 2.2.1 presents studies about code degradation and bugs and how our study differ from literature. Section 2.2.2 presents studies about refactorings and bugs, and a comparison about the study settings.

2.2.1 Studies on Code Degradation and Bugs

Previous work (43, 44, 45, 46) have studied the relationship between the degradation of the code structure quality and bugs. Khomh et al. (43, 45), for instance, have shown that degraded code elements are more change and bug-prone than others. Similarly, D'Ambros, Bacchelli, and Lanza (44) evaluated whether an increase in the number of code smells can induce bugs or not. That is, the authors assess if more degraded code elements are, more susceptible to contain bugs the code elements are. As a result, they have found that an increase in the number of code smells is likely to induce bugs in the software systems. Furthermore, a recent study conducted by Rahman and Roy (46) revealed that code clones might introduce bugs in the software systems if such code elements are not changed consistently during software evolution.

In summary, all these studies have investigated the fact that degraded code elements may be somehow related to bugs. Their findings potentially help in identifying classes that need to be refactored. However, these studies (43, 44, 45, 46) overlook the fact that developers very often apply refactorings to remove the structural problem (6, 19). Hence, in this dissertation, we aim at filling the literature's gap by analyzing the frequency that refactored code is bug-prone regarding its degradation level and the frequency of refactorings performed in code elements that are bug-prone.

2.2.2 Studies on Refactorings and Bugs

Some studies have investigated the bug-proneness of refactored code (1, 3). Bavota et al. (1) analyze if refactoring induces bugs. In other words, the authors analyze if refactored code elements contain bugs in further commits. They focus on analyzing (i) the percentage of refactored code elements that are bug-prone, and (ii) the refactoring types that are more likely to make the refactored code bug-prone. Their results show that refactored code is often bug-prone. Furthermore, they have found that some refactoring types are more harmful than others, such as refactorings involving hierarchies (e.g., Pull up Method). Conversely, Weißgerber and Diehl (3) investigate if the number of bug reports opened in the next five days after the refactoring increases or decreases. That is, the authors investigate whether the bug-proneness of refactored code increases or decreases according to the number of refactorings and opened bug reports. They found that a high ratio of refactoring is often followed by an increasing ratio of bug reports.

However, existing studies (1, 3) do not assess the characteristics of refactorings that make code elements bug-prone. That is, the authors overlook the impact of different refactoring tactics on the bug-proneness of refactored code. The bug-proneness code elements refactored by the different refactoring tactics might differ. Furthermore, the authors overlook the fact that developers very often apply batch refactorings (18, 40). Applying batches might be more or less bug-prone if compared to the application of isolated refactorings.

Moreover, existing studies (1, 3) performed coarse-grained analyses that might overshadow the results of the bug-proneness of refactored code. First, existing studies do not analyze the complementary properties of the bug-proneness of refactored code, i.e., frequency and distance (see Section 2.1.2). These properties allow us to better characterize how bug-prone is a refactored code. For instance, the distance property allow us to know if a code element became buggy very close or very far from the refactoring. Instead, existing studies (1, 3) focus on analyzing the percentage of refactorings that make code elements bug-prone. For that, they only take into consideration if a refactored code contains at least one bug in further commits. This analysis might not suffice to measure the bug-proneness of refactored code because a bug might have emerged in the same code element many commits away from the refactoring. Second, they do not manually analyze if the refactoring indeed made the source code bug-prone in the analyzed software systems. A manual analysis is, therefore, essential to guarantee the reliability of the results. Third, these studies only analyze a few software projects, and only consider major releases instead of making a more fine-grained analysis (commit by commit). As there are many changes between two major releases, many refactorings and bugs are hidden or unidentifiable in their analyses.

In order to fill the gap in the literature, we perform a more fine-grained analysis by considering the (i) the characteristics of refactorings that make code elements bug-prone, (ii) the complementary properties of the bug-proneness of refactored code, (iii) the entire history of commits in 12 software projects against major releases in only 3 software projects from previous work (1). Our goal is to verify whether refactoring is bug-prone, as previous work suggests (1, 3).

Comparison among Study Settings. Table 2.1 presents studies that somehow investigate the relationship between refactorings and bugs. The first column presents each related work. The second column presents the goal of each related work. The third column shows the tools that each study relied on to collect data. The fourth column presents the approaches adopted to collect the refactorings. The fifth column presents the analyzed software projects. The sixth column presents the approach used to collect bugs. Finally, the seventh column summarizes the procedures for data analysis. We discuss the main limitations of each related work listed in the table and also present how our empirical study addresses these limitations as follows.

Regarding the software projects, we analyzed 12 well-known projects from different domains. In fact, most related studies (1, 3, 28, 48) analyze only two software projects. Concerning the bug detection process, we carefully evaluate both bugs explicitly described in reports and bugs detected via static analysis, which we validated with 14 researchers against only two researchers of the only study that also validates bugs (28). Previous work (1, 3, 28, 48) consider all bug reports as real bugs. However, Kim et al. (23) mentions that bug report classifications are unreliable, and there might be a bug report for enhancement rather than an actual bug. We emphasize that it is important to identify bugs from different sources since there might be cases where bugs could not have been identified and reported by users or developers in bug reports.

Concerning the refactoring computation, in our work, we automatically detected refactorings by using the Refactoring Miner tool (20, 66), while the other studies (41, 48) do not systematically distinguish changes that are actual refactoring operations. Thus, they misidentified some refactorings by considering changes like variable replacements, varying logic of statements, and others non-explicitly related to refactoring. Finally, they analyze the refactoring operations throughout the project releases rather than commits.

In our work, we consider the entire set of commits for all software projects analyzed in this study.

Table 2.1: Overview of previous studies *versus* our study

Study	**Study Goal**	**Tool Support**	**Change Detection**	**Software Projects**	**Bug Detection**	**Data Analysis**
Bavota et al., SCAM'12 (1)	Understand to what extent refactorings induce bugs	Ref-finder; SZZ	All Refactorings, Per refactoring type; Per release	3 Projects: Ant, ArgoUML, Xerces	Bug report	Refactorings inducing bugs
Kim et al., ASE'06 (28)	Propose algorithms to automatically identify bug-introducing changes	Adapted SZZ	N/A	2 Projects: Columba, Eclipse IDE	Bug report; SCM	Changes only
Ratzinger et al., MSR'08 (41)	Analyze the influence of refactoring on bugs	CVS for changes	Historical change analysis only	5 Projects: ArgoUML, JBoss Cache, Liferay Portal, Spring Framework, XFramework	Bug report; CVS	Analyze refactorings to predict bugs in the near future
Śliwerski et al., MSR'05 (24)	Analyze when changes induce bug fixes	CVS for changes	Bug-fixes only	2 Projects: Mozilla, Eclipse IDE	Bug report	Fix-inducing changes
Weißgerber et al., MSR'06 (3)	Analyze whether refactoring is less error-prone than other changes	CVS for changes; RefVis for validating refactoring candidates	Per refactoring type; Per release	3 Projects: ArgoUML, JEdit, JUnit	Bug report	Analyze the percentage of refactorings per day according to the ratio of bugs opened within the next 5 days.
Wu et al., FSE'11 (48)	Recover links between bugs and committed changes	Proposed the ReLink tool that links changes and bugs	Changes among commits	2 Projects: ZXing, OpenIntents	Bug report; CVS; SVN	Change logs
Our Study	Assess the bug-proneness of refactored code	Refactoring Miner, heuristic to detect batches, adapted version of SZZ	Isolated Refactoring, Batch Refactoring, Per refactoring tactic, per refactoring type, the entire history of commits	12 Projects	Bug reports, bugs via static analysis	The bug-proneness of refactored code in terms of frequency and distance. Furthermore, analyzing the characteristics of refactorings that make code elements bug-prone

2.3 Final Remarks

This chapter provided the required background to support the understanding of this dissertation. We presented the basic concepts used throughout the next chapters. Besides the basic concepts, we discussed related work as follows. First, we presented empirical studies on the relationship between code structural quality and bugs. Second, we discuss studies that investigate the relationship between refactorings and bugs. Based on the discussion presented in this chapter, we claim that it is necessary a more fine-grained and systematic analysis to better assess the bug-proneness of refactored code. For this purpose, the next chapter presents our research methodology.

3
Research Methodology

This chapter presents our research methodology for assessing the bug-proneness of refactored code. We applied this methodology in the context of two longitudinal studies in different contexts. In other words, our research methodology was intended to support the assessment of bug-proneness in the context of isolated refactorings (Chapter 4) and batch refactorings (Chapter 5). These studies allow us to have a better understanding of the influence of key refactoring characteristics on the bug-proneness.

The goal is the same for the two longitudinal studies, while some study procedures might differ from one study to another. Thus, this chapter provides the common procedures for both studies described in Chapter 4 and Chapter 5. The specific research questions of each study are described in each chapter.

Thus, the remainder of this chapter is organized as follows. Section 3.1 presents the research goal. Section 3.2 presents the procedure to select software projects. Section 3.3 presents the procedure to detect code smells. Section 3.4 presents the procedure to identify refactorings. Section 3.5 presents how we identify bugs as well as the bug report commit. Sections 3.6 and 3.7 presents the procedure to identify the bug-fix commit and bug-fix elements, and the bug-introducing commit, respectively. Section 3.8 presents how we detect changes. Finally, Section 3.9 presents the final remarks of this chapter and introduces the next chapter.

3.1
Research Goal

The empirical studies presented in this dissertation aim at evaluating the bug-proneness of refactored code. By relying on the guidelines provided by Wohlin et al. (2), we refined and structured the study goal as follows:

– **Analyze** refactorings applied to structurally degraded code
– **For the purpose of** assessing the bug-proneness of refactored code
– **With respect to** the frequency of refactored code that became buggy, the frequency of refactorings performed in code elements that are bug-prone, and the distance between the refactored code and the code element that contain bug

- **From the viewpoint of** researchers
- **In the context of** Java open source projects with bug reports available.

Figure 3.1 illustrates the common study phases for the two studies conducted in this dissertation. First, we select software projects to be analyzed in the two studies. Second, we detect code smells since our analyses are performed in the context of degraded code elements. Third, we identify refactorings for all commits of all analyzed software projects. Additionally, we perform a manual validation to guarantee the reliability of our results. Fourth, we detect bugs from two different sources, i.e., bugs reported by a user or a developer, and bugs identified via static analysis tools. We chose to study both sources of bugs since not all bugs in a system are identified and reported by developers and users in bug reports. Furthermore, we detect all code elements and commits that are fundamental for the analyses performed in the two studies. Finally, we detect changes performed in each commit of each software project. We describe in detail each phase in the next sections.

3.2 Software Projects Selection

To conduct our studies, we selected a set of software projects from GitHub repository. We analyze 12 software projects selected using the following quality criteria. First, Java projects, a very popular programming language[1]. Second, open source projects to allow the studies replication. Third, highly popular projects and from different domains. Fourth, users actively use their issue tracking systems such as Bugzilla and the GitHub issue management system for bug reporting. Fifth, software projects in which their developers have the culture to describe in the commit message the bug report being fixed in such commit. Furthermore, we focused the data analysis on open source projects to support the studies replication and extension. In addition, it is difficult to have access to the proprietary code for analysis, but there are several open source projects with many contributors, e.g., Elasticsearch (1,021 contributors). Moreover, by analyzing open source projects, we can compare our results with previous work (1, 3) that also analyze open source projects.

Table 3.1 provides general data about the analyzed projects. The first column presents the name of the software project. The second column presents the number of lines of code. The third column presents the number of classes. The fourth column presents the analyzed period. The fifth column presents the number of commits.

[1]`https://www.tiobe.com/tiobe-index/`

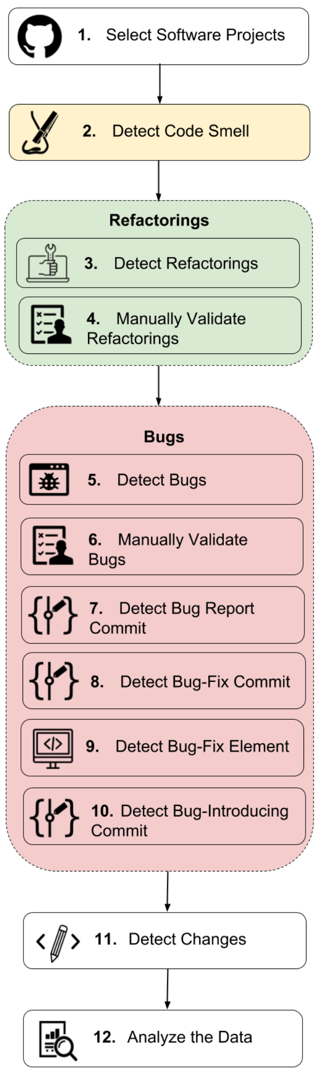

1. Select Software Projects
2. Detect Code Smell
Refactorings
3. Detect Refactorings
4. Manually Validate Refactorings
Bugs
5. Detect Bugs
6. Manually Validate Bugs
7. Detect Bug Report Commit
8. Detect Bug-Fix Commit
9. Detect Bug-Fix Element
10. Detect Bug-Introducing Commit
11. Detect Changes
12. Analyze the Data

Figure 3.1: Study Phases

3.3 Code Smell Detection

Code smells are often identified with rule-based strategies (15). Each strategy defines a set of software metrics and thresholds. Thus, to apply such strategies, it is necessary to compute software metrics for all projects. After collecting the selected metrics, we applied a set of previously defined rules (16, 17) to identify code smells per software project. The specific metrics and thresholds for supporting the identification of code smells were defined in a previous work (17). We used these rules because they are refinements of the well-known rules proposed by Lanza and Marinescu (16), and have a precision of 72% and recall of 81% on average (17).

Table 3.2 presents the 17 types of code smells analyzed in our studies (19). We selected these code smells since they are very common, and they

Table 3.1: General data of the analyzed software projects

Software Project	**LOC**	**#Classes**	**Analyzed Period**	**#Commits**
Ant	137,314	1,784	2000-01 to 2016-07	13,331
Derby	1,760,766	3,741	2004-08 to 2017-08	7,865
Elasticsearch	578,561	8,845	2010-08 to 2016-08	23,597
Elasticsearch-hadoop	34,640	599	2013-11 to 2017-11	1,718
ExoPlayer	87,321	1,218	2014-06 to 2017-10	3,081
Fresco	50,779	860	2015-03 to 2017-11	1,535
Material-dialogs	7,584	98	2014-05 to 2017-10	1,330
Netty	244,694	4,316	2008-08 to 2017-11	8,357
OkHttp	49,739	642	2011-05 to 2016-08	2,645
Presto	350,976	4,146	2012-08 to 2016-08	8,056
Spring-boot	178,752	6,513	2012-10 to 2016-08	8,529
Tomcat	668,720	2,275	2006-03 to 2016-12	17,732
Sum	4,149,846.00	33,702.00	-	97,776.00
Mean	345,820.50	2,808.50	-	8,148.00
Stdev	494,967.03	2,550.72	-	7,040.05

are directly related to the most frequent refactoring types (19, 21, 31). For instance, *Feature Envy* is a code smell type that refers to a method that is more interested in other classes than in its own class (19). To remove this code smell, developers apply a refactoring type called *Move Method* to move the “envious method" from its current class to the class to which it is interested. The first column shows the smell type, and the second column presents the description of the smell.

3.4 Refactoring Detection and Manual Validation

Let $S = \{s_1, \cdots, s_n\}$ be a set of software projects. Each software s has a set of commits $V(s) = \{v_1, \cdots, v_m\}$. Each commit v_i has a set of elements $E(v_i) = \{e_1, \cdots, e_k\}$ representing all methods, classes and fields belonging to it. To detect refactorings, we must analyze transformations between each subsequent pair of commits. Thus, we assume that R is a refactoring detection function where $R(v_i, v_{i+1}) = \{r_1(rt_1, e_1), \cdots, r_k(rt_k, e_k)\}$ gives us a set of tuples composed of two elements: the refactoring type (rt_i) and the set of refactored elements represented by e_i. So, the function R returns the set of all refactorings detected in a pair of commits.

We used the Refactoring Miner tool (20, 66) to identify refactorings in the selected projects. Tsantalis et al. (20, 66) have reported that Refactoring Miner has a precision of 98%, and a recall of 87%. We choose to study the 13 most commonly investigated refactoring types in the literature (18). These refactoring types are defined in the Fowler’s catalog (19). Refactoring Miner detects all 13 refactoring types investigated in our studies (11). We identified

Table 3.2: Analyzed Smell Types

Smell Type	**Description**
Brain Class	Long and complex class that centralizes the intelligence of the system
Brain Method	Long and complex method that centralizes the intelligence of a class
Class Data Should Be Private	A class exposing its fields, violating the principle of data hiding
Complex Class	A class having at least one method having a high cyclomatic complexity
Data Class	These are classes that have only fields and accessors methods
Dispersed Coupling	A method that accesses many code elements, and the accessed code elements are dispersed among many classes
Feature Envy	A method that is more interested in a class other than the one it actually is in
God Class	When a class centralizes the system functionality
Intensive Coupling	A method that has tight coupling with other methods, and these coupled methods are defined in the context of few classes
Lazy Class	A class having very small dimension, few methods and with low complexity
Long Method	A method that is unduly long in terms of lines of code
Long Parameter List	A method having a long list of parameters, some of which avoidable
Message Chain	A long chain of method invocations is performed to implement a class functionality
Refused Bequest	A class redefining most of the inherited methods, thus signaling a wrong hierarchy
Shotgun Surgery	When a change performed on it demands a lot of little changes to several different classes
Spaghetti Code	A class implementing complex methods interacting between them, with no parameters, using global variables
Speculative Generality	A class declared as abstract having very few children classes using its methods.

39,750 refactoring operations in total. Table 3.3 presents the refactoring types analyzed in our studies. The first column presents the refactoring type. The second column describes the problem that is intended to be addressed by each refactoring type. The third column describes the solution intended by applying each refactoring type. Furthermore, Table 3.4 presents the elements considered as refatored according to each refactoring type.

Manually validate refactoring. We conducted a manual validation of the refactorings identified by the Refactoring Miner tool to ensure the reliability of our data. Such validation covered a random set of refactoring operations from different refactoring types since the precision of the Refactoring Miner tool could vary due to the rules implemented to detect each refactoring type. We recruited ten undergraduate students to analyze the samples. The samples were divided into ten disjointed sets, and each student validated a different one. After applying a statistical test with a confidence level of 95%, we observed

Table 3.3: Analyzed Refactoring Types

Refactoring Type	Problem	Solution
Extract Method	A code fragment can be grouped together	Turn the fragment into a method whose name explains the purpose of the method
Extract Interface	Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common	Extract the subset into an interface
Extract Superclass	There are two classes with similar features	Create a superclass and move the common features to the superclass
Inline Method	When a method body is more obvious than the method	Replace calls to the method with the method's content and delete the method itself
Move Class	Your class belongs to a package that other classes unrelated to it	Move the class to a related package or create a new package if required for further use
Move Field	A field is, or will be, used by another class more than the class on which it is defined	Create a new field in the target class, and change all its users
Move Method	A method is, or will be, using or used by more features of another class than the class in which it is deffined	Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether
Rename Class	The name of the class does not reveal its purpose	Change the name of the class and update all callers
Rename Method	The name of a method does not reveal its purpose	Change the name of the method and update all callers
Pull up Field	Two subclasses have the same field	Move the field to the superclass
Pull up Method	There are methods with identical results on subclasses	Move them to the superclass
Push down Field	A field is used only by some subclasses	Move the field to those subclasses
Push down Method	The behavior on a superclass is relevant only for some of its subclasses	Move it to those subclasses

a high precision of the tool for each refactoring, with a median of 88.36%. By applying the Grubb outlier test (22) ($\alpha = 0.05$), we could not find any outliers, indicating that no refactoring type strongly influences the median precision found. Thus, the obtained results represent a key factor to trust on the results reported in this dissertation.

3.5 Bug Detection and Manual Validation

We identify bugs for the selected software projects in two different ways: (i) bugs reported in bug reports, and (ii) bugs detected via static analysis tools. We chose to analyze bugs from different sources since bugs reported in issue tracking systems are the ones that users/developers have found during testing or production time, for instance. However, there might exist dormant bugs in the source code that have not been reported or identified yet. Therefore, these bugs can be identified through static analysis tools.

We selected bug reports with status *resolved fixed*, *verified fixed*, *closed*, or *closed fixed* for analysis. Furthermore, we chose to analyze only bugs labeled as “bug” or “defect” in the issue tracking system. We collected 6,051 bug reports. Besides collecting bug reports, we detected the commit in which the bug was reported. For that, we consider the commit in the date before the bug report was opened.

Table 3.4: Refactored Elements

Refactoring Type	**Refactored Elements**
Extract Interface	classes implementing the new interface
Extract Method	(i) method created; (ii) method from where the new method was extracted; and (iii) class containing both methods
Extract Superclass	(i) classes extending the new class; and (ii) new class created
Inline Method	(i) the method which received the new code; and (ii) class containing the method
Move Field	the two classes affected by the change: the class which the field used to reside and the class which received the field
Move Class	the two packages affected by the change: the package which the class used to reside and the package which received the class
Move Method	the two classes affected by the change: the class which the method used to reside and the class which received the method
Pull Up Field	the two classes affected by the change: the class which the field used to reside and the class which received the field
Pull Up Method	the two classes affected by the change: the class which the method used to reside and the class which received the method
Push Down Field	the two classes affected by the change: the class which the field used to reside and the class which received the field
Push Down Method	the two classes affected by the change: the class which the method used to reside and the class which received the method
Rename Class	the renamed class and the package that contains it.
Rename Method	the renamed method and the class that contains it.

Additionally, we used the FindBugs tool (version 3.0.1) (51) to detect bugs via static analysis for each commit of two software projects. We have identified 49,250 bugs using FindBugs for the projects Apache Derby and Apache Tomcat. Table 3.5 presents the description of each bug pattern selected for the studies performed in this dissertation. The first column presents the bug pattern, and the second column presents the description of each bug pattern according to the FindBugs description[2]. We consider as report commit, the first commit in which Findbugs reported the bug in a code element.

Manually validate bugs. Previous research (23) mentions that bug report classifications are unreliable. Thus, we performed a bug report manual classification to identify which bug reports actually represent bugs in the projects of Apache Tomcat, Apache Derby, and Apache Ant. This classification was performed in pairs by 14 researchers. Each person of the pair was responsible for manually classify the same bug report as “bug” or “not bug”. When there was a divergence in opinion, the pair should talk and define the final classification of such bug. In the final analysis, we considered only bug reports that represent bugs in such projects. We manually validated 1,477 bug reports, in which 516 (34.94%) were classified as “bug” and 961 (65.06%) as “not bug”. We also performed a manual validation of the bugs collected via static analy-

[2]`http://findbugs.sourceforge.net/bugDescriptions.html`

Table 3.5: Bug Patterns of FindBugs

Bug Pattern	**Description**
Malicious Code	Variables or fields exposed to classes that should not be using them
Multithreaded	Thread synchronization issues
Performance	Inefficient memory usage/buffer allocation, usage of non-static classes
Security	Similar to malicious code vulnerability
Correctness	Apparent coding mistakes

sis. This procedure has the purpose of reducing the number of false positives of the tool. To do that, we followed the same procedure described above for the bug reports. We manually validated 198 bugs, in which 168 (84.85%) was classified as “bug” and 30 (15.15%) as “not bug”.

3.6 Bug-Fix Commit and Bug-Fix Elements Detection

To identify the bug-inducing commit (described below), it is necessary to know the bug-fix commit and the code elements involved with the fix of the bug. This procedure is necessary only for bug reports since bugs detected via static analysis already output the commit and the elements associated with the fix of the identified bug. Thus, a common practice among developers is to include the bug report number in the commit comment whenever they fix a bug associated with it (24). In this way, to map a bug report with its fix commit, we automatically search log messages for references to bug reports such as “bug 23442” or “fix for bug 23442” as proposed by Dallmeier and Zimmermann (25). We ignored bug reports that we could not find the commit of the fix because, without the fix commit, we cannot find the fixed files. Thus, these bug reports are considered not functional (26). We consider as buggy elements, all code elements that were modified in the fix commit. Furthermore, we discarded from our analysis buggy elements involved in test classes.

3.7 Bug-Introducing Commit Detection

Given the bug-fix commit and the bug-fix elements previously identified, we used the bug-introducing change identification algorithm proposed by Śliwerski et al. (24) (the SZZ algorithm) to identify when the bug was introduced in the project. SZZ is currently the most used algorithm for automatically identify fix-inducing commits (27). SZZ aims at identifying the lines modified in a bug-fixing commit, and then it identifies the fix-inducing change immediately before each line of the bug-fixing commit. As the original version of SZZ may have false positives and false negatives, we have used a combination of heuristics proposed by Kim et. al (28) and Williams and Spacco (29). Kim et. al (28) mention two limitations of the original SZZ: (i) not all changes are fixes, i.e., even if a file change is defined as a bug-fix by developers, not all hunks in the change are bug-fixes; (ii) there is not enough information in bug tracking systems, and because of this an incorrect bug-inducing commit may be chosen. Using their approach, we can remove 38-51% of false positives and 14% of false negatives as compared to the original implementation of SZZ (24). SZZ outputs a list of commits related to the introduction of the bug in the software system. For analysis purposes, we considered only the newest commit reported by SZZ.

3.8 Changes Detection

In the studies performed in this dissertation, we considered code elements that have changed during the software history. For that, we collect all classes changed in each commit of each software project analyzed in this dissertation. Thus, to view the changed files on each commit of a given project, we used the following command from GitHub: `git log -pretty="commit=%H:%ae" -name-status`[3]. This command returns the list of all changed files in all commits of such project.

3.9 Final Remarks

This chapter presented the study design of the work. First, we discussed our goals. For this purpose, we select a set of 12 open source project of GitHub repositories. First, we detected code smells. Then, we used the Refactoring Miner tool to detect refactoring operations over our project sample. After, we detected bugs and identified the report commit, the introducing commit, the fix commit, and the code elements involved with the fix of the bug. Finally, we detected the changed files over the software project history. The next chapter presents and discusses the results of the first study conducted in this dissertation.

[3]`https://git-scm.com/docs/git-log`

4
The Bug-Proneness of Code Refactored through Isolated Refactoring

According to the literature (1, 3), refactored code elements are often bug-prone. However, these studies tend to solely analyze if a refactored code contains a bug in the same or further commits. In spite of this limited analysis, such studies overlook the impact of different refactoring tactics on the bug-proneness of refactored code. For instance, one could say that pure refactorings, i.e., root-canal refactorings, would be less bug-prone than floss refactorings since its goal is only to improve the code structural quality. Therefore, when studying the bug-proneness of refactored code elements, the analysis of different refactoring tactics is also important.

This chapter presents a longitudinal study focused on investigating the bug-proneness of code refactored through isolated refactoring. First, we perform an overall analysis by assessing the frequency of isolated refactorings performed in code elements that are bug-prone and not bug-prone at all. Second, we assess the frequency regarding the number of degraded code elements that became buggy regarding being refactored or not. Third, we measure the distance between the isolated refactoring and the bug. Finally, we investigate the characteristics of refactorings that make the source code the refactored code element bug-prone, i.e., refactoring types and refactoring tactics.

Part of the study presented in this chapter was presented as a poster at the 40th International Conference on Software Engineering (5). The remainder of this chapter is organized as follows. Section 4.1 describes the study settings, including the study goal and research questions. Section 4.2 presents the results of our empirical study regarding the bug-proneness of code refactored through isolated refactoring. Section 4.3 discusses threats to the study validity. Section 4.4 summarizes this chapter and introduces the following chapter.

4.1
Study Settings and Procedures

General study procedures are described in Chapter 3. This section describes the specific settings of this study aimed at understanding the bug-proneness of code refactored through isolated refactorings. The remainder of this section is organized as follows. Section 4.1.1 presents the research questions and associated hypotheses. Section 4.1.2 presents the procedure for identifying isolated refactorings. Section 4.1.3 presents the procedure to manually classify refactoring tactics. Section 4.1.4 presents the procedure to measure the two complementary properties of the bug-proneness of refactored code: *frequency* (Section 4.1.4.1), and *distance* (Section 4.1.4.2). Finally, Section 4.1.5 presents the procedure to manually validate the results for the distance property.

4.1.1 Research Questions

The empirical study presented in this chapter aims at evaluating the bug-proneness of code refactored through isolated refactoring. Our study goal is defined in Chapter 3. From our study goal, we designed the following specific research questions (SRQs).

SRQ₁. How many isolated refactorings were performed in code elements that are bug-prone?

As our goal is to measure the bug-proneness of refactored code elements, then we should know how many isolated refactorings were performed in code elements that are bug-prone. Additionally, we investigate the number of code smells in code elements touched by isolated refactorings. Hence, one could assume that if a refactored code element had multiple code smells before the refactoring was applied, then those code elements are more bug-prone. We do this analysis for isolated refactorings performed in code elements that are bug-prone and isolated refactorings performed in code elements that are not bug-prone at all. The result of this SRQ serves as a starting point to analyze the characteristics of refactorings that make the source code bug-prone.

SRQ₂. Are refactored code elements less bug-prone than non-refactored code elements?

As a result of SRQ$_1$, we know how many isolated refactorings were performed in code elements that are bug-prone. Thus, we can now perform further investigation on refactorings performed in code elements that are bug-prone. Then, SRQ$_2$ aims at investigating whether refactored code elements are less bug-prone than non-refactored code elements. For that, we analyze code elements that contain code smells before the refactoring operation. That is, such code elements tend to degrade as developers constantly apply changes along the software maintenance. Thus, aimed at reducing the negative effects of code structural quality degradation, developers often apply refactorings on the degraded code elements (6). By definition, refactoring improves the structural quality of the refactored code elements. It implies improving the software maintenance as a whole, which means that changing the source code becomes easier, e.g., to add new software functionalities.

Regardless the purpose of developers while refactoring code, the general assumption is that the source code will have better structural quality. Consequently, one could assume that refactoring degraded code elements make them less bug-prone. That is because previous studies show that most changes applied to the code elements are due to changing software functionalities, and bugs are mostly functionality-related. However, all these assumptions are limitedly addressed by previous studies (see Section 2.2). We address such limitation by answering SRQ_2. Thus, SRQ_2 serves as a starting point to understand to what extent we could blame refactorings for bugs. We derived our null (H_0) and alternative hypotheses (HA_1) from SRQ_2 as presented in Table 4.1.

Table 4.1: Study hypotheses derived from SRQ_2

Hypotheses	**Description**
H_0	There is no difference in the bug-proneness of refactored and non-refactored code.
HA_1	There is a difference in the bug-proneness of refactored and non-refactored code.

SRQ_3. How many times a degraded code element that was refactored has to change to become buggy?

Understanding if applying refactorings to degraded code elements reduces the bug-proneness of code elements (SRQ_2) is a valid starting point. However, some tricky questions about the bug-proneness of refactored code might emerge. A major question regards how close is the applied refactoring from either the bug introduction or the bug report. As for how close, we mean how many times a degraded code element that was refactored had to change to become buggy. In fact, previous studies suggest that there is a relationship between refactorings and bugs. It does not drastically differ from our investigation with SRQ_2. However, all these studies fall short in showing that the refactoring is bug-prone. Thus, their findings might mislead developers to refactor degraded code elements. With SRQ_3, we aim to understand how many changes developers have to apply on a degraded code element in order to make it buggy. Our goal is to understand how reliable is stating that refactorings are bug-prone, as reported by previous studies. We derived our null (H_0) and alternative hypotheses (HA_1) from SRQ_3 as presented in Table 4.2.

Table 4.2: Study hypotheses derived from SRQ_3

Hypotheses	**Description**
H_0	There is no difference in the bug-proneness regarding distance when considering the insertion commit and report commit.
HA_1	There is a difference in the bug-proneness regarding distance when considering the insertion commit and report commit.

SRQ₄. How frequent degraded code elements are bug-prone per refactoring type?

By assessing how many times a degraded code element has to change to become buggy (SRQ_3), we promote a better understanding about to what extent blaming refactorings for bugs makes sense. However, refactorings vary per type. Recent studies provide evidence that each refactoring type has a different effect on the code structural quality (6). Thus, one could assume that these different refactoring types contribute differently to the bug-proneness of refactored code elements. We designed SRQ_4 for investigating such a reasonable assumption. Our goal is to identify the refactoring types that developers apply at most to degraded code elements. After that, we investigate the bug-proneness of code elements that were refactored through each type. By doing that, we can better understand if the bug-proneness of refactored code significantly varies among different types.

SRQ₅. How many times a degraded code element that was refactored has to change to become buggy per refactoring tactic?

Similar to SRQ_4, we also want to have a better understanding of how different refactoring tactics can impact in the bug-proneness of refactored code elements. The refactoring tactics should be investigated since someone can argue that the results presented in SRQ_3 can be different when we consider refactoring tactics. Since root-canal and floss refactoring have different purposes, we can expect that the bug-proneness of refactored code per refactoring tactic may differ in each one. We derived our null (H_0) and alternative hypotheses (HA_1) from SRQ_5 as presented in Table 4.3.

Table 4.3: Study hypotheses derived from SRQ_5

Hypotheses	Description
H_0	There is no difference in the bug-proneness of refactored code per refactoring tactic.
HA_1	There is a difference in the bug-proneness of refactored code per refactoring tactic.

4.1.2 Identification of Isolated Refactorings

As described in Section 3.4, we collect all refactorings performed in all software projects analyzed in this study. After that, to answer the SRQs designed for this study, we divide the set R of refactorings into two disjoint sets. The first set is composed of isolated refactorings, i.e., refactorings that are not performed in conjunction with other refactoring operations. The second set is composed by batch refactorings, i.e., a sequence of inter-related refactoring operations. We formally define isolated refactorings as follows.

Definition 1.
Let r be a single refactoring.
Let $b = (r_1, r_2, ..., r_n)$ be a sorted list of refactorings, which we call a batch a refactoring, composed of n arbitrary refactorings.
r is an *isolated refactoring* if r is not in b. That is, $r \neq r_i \ \forall i = \{1, .., n\}$.

4.1.3 Manual Classification of Refactoring Tactics

To classify the isolated refactorings into *root-canal refactorings* and *floss refactorings*, we conducted a manual inspection of a randomly selected sample composed of 2,119 refactorings. We manually analyzed whether the changes performed during the refactoring do not modify the behavior. We classify a change as floss refactoring when there are behavioral changes, such as an addition of methods or changes in the method body that are not related to refactoring transformations. When we did not identify behavioral changes, the refactoring was classified as root-canal. This inspection was performed by three researchers. Two of them are very experienced refactoring researchers. The most experienced one solved the conflicts. As a result, we found that developers apply root-canal refactoring in 31.5% of the cases. The confidence level for this number is 95% with a confidence interval of 5%.

4.1.4 Measuring the Properties of the Bug-Proneness of Refactored Code

Figure 4.1 illustrates how we evaluate the bug-proneness of refactored code for isolated refactorings. The figure presents the timeline in terms of commits of a software project. For each commit, we represent the key event of such commit. For the sake of simplicity, we consider only events performed in the same code element, in this case, the *method X*.

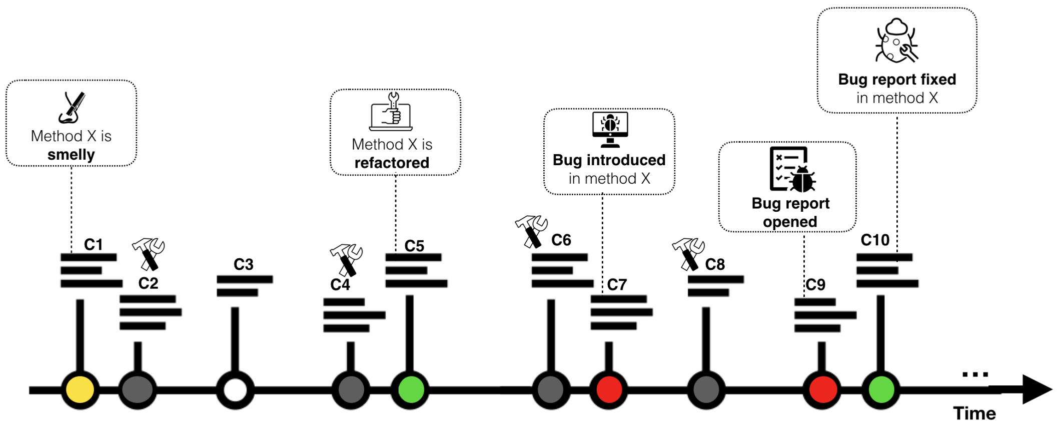


Figure 4.1: Evaluation of the bug-proneness of code refactored through isolated refactorings

By analyzing the timeline presented in Figure 4.1, we can see that in the first commit (represented by C1) the method X was *smelly*, i.e., method X was structurally degraded. Then, two *changes* were performed in method X: one change in C2, and another one in C4. After these changes, a *refactoring* operation of any type was applied to the degraded code element (method X) in C5. After being refactored, another *change* was performed in method X in C6. Then, a *bug* was *introduced* in method X, although it has only be perceived and *reported* by users in C9. Between the bug introduction and the bug report, another change was performed in method X (C8). Finally, the bug was *fixed* in method X in C10. Sections 4.1.4.1 and 4.1.4.2 explain how we measure the two complementary properties, frequency and distance, for all analyses performed in this study.

4.1.4.1 Measuring the *Frequency* Property

We performed different analyses regarding the *frequency* property. First, to answer SRQ$_1$, we compute how many isolated refactorings were performed in code elements that are bug-prone. For that, let us consider the example of Figure 4.1. A refactoring operation was performed in C5, and the refactored code element had a bug in further commits. Thus, as in this example there was just one refactoring, and this refactored code element became buggy in further commits, then we say that 100% of the refactorings were performed in code elements that are bug-prone. We perform this analysis by counting how many isolated refactorings were performed in code elements that are bug-prone for all software systems analyzed in this study. Furthermore, we count how many isolated refactorings touched in code elements without any code smells, with a single code smell, and with multiple code smells. We perform the latter analysis for isolated refactorings performed in code elements that are bug-prone, and isolated refactorings performed in code elements that are not bug-prone at all.

Second, to answer SRQ_2, we measure the frequency by computing how many degraded code elements have been refactored or not before they become buggy. Considering the Figure 4.1, in this case, we measure the *frequency* property by counting the number of buggy code elements that were refactored in the past according to their the number of code smells on it. For instance, we consider as buggy code elements all those elements involved with the fix of the bug (method X, in this case). Then, we check whether this code element has been refactored before the introduction and the report of the bug. In our case, method X was refactored in C5. Next, we assess the number of code smells in the refactored code. That is if it had no smell, a single smell, or multiple smells before the refactoring operation. In the case of method X, it had only one smell. Thus, for this example, one buggy code element was refactored and had a single smell. We carry out this analysis for every buggy element for each software project analyzed in this study. Finally, to answer SRQ_4, we measure how many code elements are bug-prone per refactoring type. For that, we divide the isolated refactorings by refactoring type.

4.1.4.2 Measuring the *Distance* Property

We measure the *distance* property by considering all isolated refactorings, per refactoring type and per refactoring tactic. For that, we count the number of changes between the commit in which the refactoring was applied and the commit in which the bug was either introduced or reported, by considering all code elements involved with the fix of the bug (method X). In the example of Figure 4.1, it was necessary only one change (performed in C6) after the refactoring operation (C5) to introduce the bug (C7). Similarly, it was necessary two changes (C6 and C8) after the refactoring operation (C5) to the report of the bug (C9). Thus, we say that the distance is one and two, respectively. We consider both events of bugs, i.e., insert and report, due to the fact that a bug can be perceived and reported by developers or users after a long time that the bug was indeed introduced in the code element. Thus, it is important to verify if the results change by considering different events of bugs.

4.1.5 Manual Validation of the *Distance* Results

To analyze if a refactored code is indeed bug-prone, we manually analyzed 25% of the cases in which the refactoring was very close to the bug. That is, we manually analyzed 25% of the results found for the distance property in which it was necessary few changes after the refactoring so that the code element became buggy. For that, we first analyzed the bug report. Then, we checked which code elements were fixed for such bug report. After, we analyzed the commit in which the refactoring was applied, and the bug was reported. Finally, we tried to identify any relationship between the refactored code elements and the buggy code elements. The validation was performed by three researchers.

4.2 The Bug-Proneness of Code Refactored through Isolated Refactorings

This section presents the study results that answer the SRQs of this study. The following subsection presents the results for each one of our specific RQs.

4.2.1 Frequency of Isolated Refactorings Performed in Code Elements that Are Bug-Prone

In this section, we aim to assess the number of isolated refactorings performed in code elements that are bug-prone, by answering the following *SRQ_1: How many isolated refactorings were performed in code elements that are bug-prone?*. Table 4.4 presents the results for all analyzed projects. The first column presents the name of the software project. The second column presents the total number of isolated refactorings per software project. The third column presents the number of isolated refactorings performed in code elements that are bug-prone. The fourth column presents the number of isolated refactorings performed in code elements that are not bug-prone at all. Furthermore, highlighted cells in the table indicate that we found a non-ignorable frequency (more than 10%) of refactorings performed in code elements that are bug-prone.

Table 4.4: Frequency of isolated refactorings performed in code elements that are bug-prone per software project

Software project	**#Isolated refactorings**	**#Isolated refactorings (bug-prone)**	**#Isolated refactorings (not bug-prone)**
Ant	1,276	11 (0,86%)	1,265 (99,14%)
Derby	3,093	22 (0,71%)	3,071 (99,29%)
Elasticsearch	5,597	736 (13,15%)	4,861 (86,85%)
Elaticsearch-hadoop	265	3 (1,13%)	262 (98,87%)
ExoPlayer	1,198	21 (1,75%)	1,177 (98,25%)
Fresco	564	5 (0,89%)	559 (99,11%)
Material-dialogs	78	12 (15,38%)	66 (84,62%)
Netty	2,782	31 (1,11%)	2,751 (98,89%)
Okhttp	698	47 (6,73%)	651 (93,27%)
Presto	1,526	32 (2,10%)	1,494 (97,90%)
Spring-boot	1,152	176 (15,28%)	976 (84,72%)
Tomcat	1,393	36 (2,58%)	1,357 (97,42%)
Total	**21,217**	**1,142** (**5.38%**)	**20,075** (**94.62%**)

As we can see in the table, more than 13% of the isolated refactorings were performed in code elements that are bug-prone considering the projects *Elasticsearch*, *Material-dialogs*, and *Spring-boot.* On the contrary, the other projects have few isolated refactorings performed in code elements that are bug-prone (varying from 0.86% to 6.73%). Overall, only 5.38% of the isolated refactorings were performed in code elements that are bug-prone. Conversely, most isolated refactorings (94.62%) were performed in code elements that are not bug-prone at all. This result is interesting because, to the best of our knowledge, none of the previous studies (1, 3) explicitly quantify the number of refactorings performed in code elements that are bug-prone (see Section 2.2).

After analyzing the number of refactorings performed in code elements that are bug-prone, we want to assess the number of code smells in code elements touched by these isolated refactorings. Figure 4.2 presents the frequency of isolated refactorings performed in code elements that are bug-prone (red bars) *versus* the frequency of isolated refactorings performed in code elements that are not bug-prone at all (green bar) according to the number of code smells in the refactored code element before the refactoring, i.e., no smell, single smell or multiple smells.

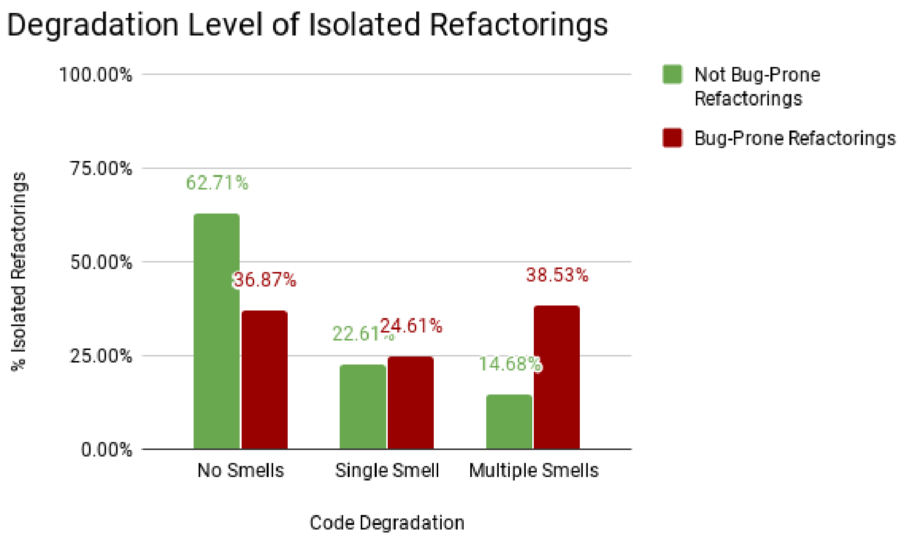


Figure 4.2: Frequency of isolated refactorings according to the structure degradation of the code elements touched by these refactorings

Our results show that 62.71% of the isolated refactorings performed in code elements that are not bug-prone touched in code elements without any code smell. That is, most of the refactored code elements that did not contain any bugs in further commits were not structurally degraded. By analyzing the isolated refactorings performed in code elements that are bug-prone, we can see that 63.14% of them touched in code elements with either a single smell or multiple smells. Thus, the refactoring might not have sufficed to combat the side effects of the source code degradation and, thus, the code elements became buggy in further commits.

To get a better understanding of the code structure degradation of the code elements touched by isolated refactorings, we analyzed the code smell types affecting such code elements. As a result, we found that 68% of the isolated refactorings performed in code elements that are bug-prone touched in code elements with smells that affect only one class (e.g., *Brain Class*, *Brain Method*, *Class Data Should Be Private*, *Complex Class*, *Data Class*, *Lazy Class*, *Message Chain*, *God Class*, *Long Method*, *Long Parameter List*, *Spaghetti Code*, and *Speculative Generality*), against 78.42% of the isolated refactorings performed in code elements that are not bug-prone. However, many of the isolated refactorings performed in code elements that are bug-prone (32%) touched in code elements affecting more than one class (e.g. *Dispersed Coupling*, *Feature Envy*, *Intensive Coupling*, *Refused Bequest*, and *Shotgun Surgery*) against only 21.58% of isolated refactorings performed in code elements that are not bug-prone. This result suggests that isolated refactorings performed in code elements that are bug-prone touched in code elements affected by critical code smells, i.e., smells affecting more than one class.

Summary for SRQ$_1$. Only 5.38% of the isolated refactorings were performed in code elements that are bug-prone against 94.62% of isolated refactorings performed in code elements that are not bug-prone at all. Additionally, 62.71% of the refactorings performed in code elements that are not bug-prone touched in code elements without any code smell. Similarly, 63.14% of the isolated refactorings performed in code elements that are bug-prone touched in code elements with either a single smell or multiple smells. This result shows that the refactoring might not have sufficed to fully overcome the degradation in the source code, and, as a result, bug(s) emerged in future commits.

4.2.2 Bug-Proneness of Refactored Code *versus* Non-Refactored Code

In this section, we aim to assess the frequency that refactored code is bug-prone against non-refactored code. To do that, we computed the *frequency* by analyzing how many buggy elements had been refactored or not according to the number of code smells affecting such code elements before the refactoring. Previous work (7, 8) mention that the higher the number of code smells in an element, the higher its degradation. Thus, we consider the degradation of a code element according to two categories as follows: (i) single smell, and (ii) multiple smells. Thus, this section addresses SRQ$_2$, which asks *Are refactored elements less bug-prone than non-refactored elements when such elements are degraded?*. To answer this question, we compute the frequency of degraded code elements that have been refactored or not before they become buggy. Table 4.5 presents the results per software project. The first column presents the software projects. The second and third columns present the frequency of code elements (i) hosting either single or multiple smells, (ii) weren't refactored, and (iii) became buggy. The fourth and fifth columns show the frequency of code elements (i) hosting either single or multiple smells, (ii) were refactored and (iii) became buggy. Finally, the last column presents the proportion of degraded code elements that were refactored and became buggy (summing up the values of the fourth and fifth columns), and degraded code elements that became buggy and weren't refactored (summing up the values of the second and third columns). By measuring the proportion, we are able to see the percentage of degraded code elements that were refactored and became buggy. The equation below computes such proportion. Furthermore, highlighted cells in the table indicate that we found a non-ignorable frequency (more than 10%) for such cases of analysis.

$$proportion = \frac{\#degraded\ \&\ refactored\ \&\ buggy\ code}{\#degraded\ \&\ non{-}refactored\ \&\ buggy\ code}$$

Table 4.5: Frequency of the bug-proneness of refactored code vs. non refactored code per software project (isolated refactorings)

	No Refactoring		**Refactoring**		
Projects	**Single smell & Buggy**	**Multiple smells & Buggy**	**Single smell & Buggy**	**Multiple smells & Buggy**	**Proportion**
Ant	29.63%	37.04%	0.00%	33.33%	50.00%
Derby	7.52%	91.23%	0.00%	1.25%	1.27%
Elasticsearch	39.95%	31.09%	2.55%	26.42%	40.78%
Elasticsearch-hadoop	0.00%	0.00%	0.00%	0.00%	-
ExoPlayer	71.43%	0.00%	28.57%	0.00%	40.00%
Fresco	82.64%	17.36%	0.00%	0.00%	0.00%
Material-dialogs	0.00%	1.03%	0.00%	98.97%	9600.00%
Netty	60.26%	38.46%	1.28%	0.00%	1.30%
Okhttp	7.69%	80.77%	0.00%	11.54%	13.04%
Presto	32.08%	39.62%	7.55%	20.75%	39.47%
Spring-boot	17.18%	30.53%	3.05%	49.24%	109.60%
Tomcat	74.84%	24.74%	0.00%	0.42%	0.42%
Total	**45.83%**	**33.84%**	**1.58%**	**18.75%**	**25.52%**

As we can see in the table, most software projects have a non-ignorable frequency of code elements that weren't refactored, considering either single smell or multiple smells (second and third column, respectively). Some projects have a high frequency of code elements containing multiple code smells, that weren't refactored and became buggy, i.e., Derby (91.23%), and Okhttp (80.77%). Furthermore, some projects have a higher frequency of code elements containing a single smell, i.e., Tomcat (74.84%) and Fresco (82.64%). On the contrary, most software projects do not have a considerable value of frequency for code elements containing a single smell that was refactored and became buggy. Moreover, 6 out of 12 software projects contain a non-ignorable frequency of code elements containing multiple smells that were refactored and became buggy. Regarding the proportion of degraded code elements that were refactored and became buggy and degraded code elements that weren't refactored and became buggy, we can see that 50% of the projects presented a non-ignorable value of proportion. Overall, only 25.52% of the degraded code elements were refactored before becoming buggy. That is, 74.48% of the degraded code elements weren't refactored before becoming buggy. This result shows that degraded and non-refactored code is more bug-prone than degraded and refactored code.

Figure 4.3 presents a bar graph that shows the frequency of buggy elements that were refactored or not according to the number of code smells affecting the code elements before the refactoring.

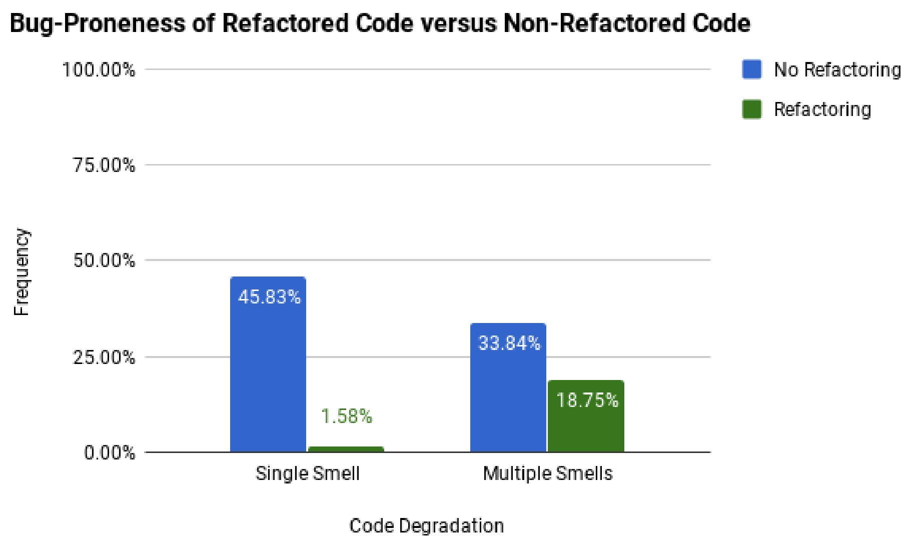


Figure 4.3: Frequency of the bug-proneness of refactored code vs. non-refactored code for isolated refactorings

When analyzing the number of code smells affecting the code elements before the refactoring regardless if a refactoring operation was applied, i.e., summing up the results for single smells (left side of the bar graph) and multiple smells (right side of the bar graph), our results show that 52.59% of the buggy code elements contain multiple smells against 47.41% that contain single smells. This result is interesting because it shows that buggy code elements had high code structure degradation. Furthermore, if we consider the refactoring operation regardless the number of code smells affecting the code elements, i.e., summing up both blue and green bars, our results suggest that 79.67% of the buggy code elements have not been refactored before they become buggy. That is, no refactoring operation was applied before the code element becomes buggy in the software system. In this case, applying refactoring on smelly elements could have avoided bugs in these elements. On the contrary, 20.33% of the buggy elements were refactored before they become buggy.

We applied the Fisher's test to compute the strength of the relation between refactoring and degraded code with bugs (9). Furthermore, we used the Odds Ratio (10) to compute the possibility of the presence or absence of a phenomenon (i.e., refactoring) to be associated with the presence or absence of the other phenomenon (i.e., degraded code with bugs). Considering all projects analyzed, we found a p-value less than 0.05, and Odds Ratio equals to

0.2894. Thus, our results show that the possibility of a refactoring be related to a degraded code with bugs is 0.28 if compared to non-refactoring related to degraded code with bugs. In summary, our results lead us to reject the null hypothesis H_0 and accept the alternative hypothesis HA_1.

Summary for SRQ$_2$. 79.67% of the smelly elements that became buggy were not previously refactored. This result shows that degraded, non-refactored code tends to be more bug-prone than degraded and refactored code.

4.2.3 Distance Between the Refactoring and the Bug

This section aims to analyze the bug-proneness of refactored code elements based on the property *distance*. After computing the distance, we analyze the quartiles distribution of the data to see whether refactored code elements are bug-prone or not. That is, if there are several changes between the isolated refactoring and the bug, then the refactoring might have made the code element less bug-prone. Similarly, if there are few changes between the refactoring and the bug, then the refactoring might have made the code element more bug-prone. The result of this analysis will allow us to answer SRQ$_3$, which asks *How many times a degraded code element that was refactored has to change to become buggy?*.

Table 4.6 presents the results for isolated refactorings, considering the insertion and report commits for all projects analyzed in this study. The first column shows the considered commit of the bug (insertion or report). The second column presents the number of times that a refactored code became buggy. The following columns present the minimum value of distance, the quartiles (25%, 50%, 75%), and the maximum value of distance found. We applied the Grubb outlier test ($\alpha = 0.05$), and we removed outliers for both insertion and report commit data. Thus, these results represent a key factor to provide confidence in the findings reported in this study.

Table 4.6: Bug-proneness in distance for isolated refactorings considering bug reports

Bug	N	Min	Q1	Q2	Q3	Max
Insertion	786	1.00	7.00	17.00	32.00	271.00
Report	2677	0.00	9.00	21.00	43.00	364.00

Our results for the bug report analysis show that in 75% of the times, a code element needs at least seven changes to become buggy after being refactored. In the analyzed software projects, seven changes happened in approximately three months (considering the mode in the distance values of Q1). That is, the refactored code element became buggy in the software systems three months after the refactoring operation. To actually assess if refactoring is bug-prone, we also analyzed bugs via static analysis. For that, we collected bugs via static analysis for two software projects analyzed in this study, Apache Tomcat e Apache Derby. The goal of analyzing the distance property for bug reports and bugs via static analysis is to see if there is a difference in the results considering both sources of bugs. Table 4.7 presents the results of the analysis of bugs via static analysis.

Table 4.7: Bug-proneness in distance for isolated refactorings considering bugs collected via static analysis

Bug	**N**	**Min**	**Q1**	**Q2**	**Q3**	**Max**
Insertion	186	1.00	8.25	26.00	37.00	58.00

As we can see, when analyzing bugs via static analysis, a code element needs at least nine changes to became buggy after being refactored. However, if we consider the number of months that the refactored code took to became buggy, it was necessary 21 months for these nine changes happen. Both analyses of bug reports and bugs via static analysis suggest that the other changes occurred in the period between the refactoring and the bug might be related to the bug. In other words, the refactoring is unlikely to be bug-prone due to the fact that lots of changes are performed in between the isolated refactoring and the bug. We performed a manual analysis of the results of bug reports considering distances up to the ones found in the second quartile (Q2) (see Section 4.1.5). However, we could not find any explicit evidence of refactorings that are bug-prone. That is, by definition, refactoring improves the code structural quality without changing the external behavior. On the other hand, the bug is associated with changing the expected behavior of the software system. Thus, it is hard to find an obvious relationship between refactorings and bugs.

To analyze whether there is any statistically significant difference between the insertion and the report commits of bug reports, we carried out the Mann-Whitney Wilcoxon (MWW) test, considering a significance level of 0.05. We found a p-value < 0.05, showing that there is a statistical difference between (i) the results of bug-proneness of refactored code when considering the bug insertion commit and (ii) the results of bug-proneness of refactored code when considering the bug report commit. In summary, our results lead us to reject the null hypothesis H_0 and accept the alternative hypothesis HA_1.

Previous work (1, 11) only analyze whether the refactored code contains a bug after being refactored or not. Bavota et al. (1) show that refactorings tend to induce bugs frequently. However, their analysis does not support the fact that the code element could have become buggy very far from the refactoring operation. In their case, the distance between the refactoring and the bug could be even further because the authors only analyze major releases instead of a more fine-grained analysis (commit by commit). That is, usually, between two major releases, developers perform significant changes in the source code. Thus, they probably missed refactorings and bugs when they followed this procedure since refactorings and bugs might be hidden or unidentifiable when considering only major releases. In our study, we mitigate this threat by collecting refactorings commit by commit. Although Bavota et al. (1) mention that they could find examples of refactorings inducing bugs between two major releases, we could not find any example when considering the analysis commit by commit. Our running example presented in Section 1.1 illustrates how we could not find any cases of refactoring inducing bugs. In addition, the study performed by Weißgerber and Diehl (11) analyzes bug reports opened within the next five days after the refactoring operation. This analysis might not suffice to show that refactored code is bug-prone, as we have demonstrated that it takes around 3 months for bugs reports, and 21 months considering bugs for static analysis so that the code element become buggy after the refactoring.

Summary for SRQ$_3$. In 75% of the times that refactored code elements are bug-prone when analyzing isolated refactorings, a code element needs at least nine changes after isolated refactorings so that the code element becomes buggy. As a result, refactored code elements are often not susceptible to immediately contain bugs.

4.2.4 Bug-Proneness According to Refactoring Types

Given the analysis of the *distance* property performed in SRQ$_3$, we want to understand which refactoring types are applied to buggy code elements according to the distance values. Thus, this section aims to answer SRQ$_4$ that asks *How frequent degraded code elements is bug-prone per refactoring type?*. First, we analyze the frequency that each refactoring type is further associated with a bug, considering bug reports. Table 4.8 shows the frequency of each refactoring type is further associated with a bug. Refactoring types not listed in the table did not have any value of frequency.

Table 4.8: Frequency of each refactoring type

Refactoring Type	**Frequency**
Extract Method	54.00%
Inline Method	25.00%
Rename Method	7.00%
Move Attribute	6.00%
Move Method	5.00%
Extract Superclass	3.00%

As we can see in the table above, *Extract Method*, and *Inline Method* are the most frequent refactoring types associated with bugs. Previous work (1) shows that such refactoring types are considered harmful for the system. We could confirm it in our analysis. Thus, to get a better understanding of this phenomenon, we analyzed how many changes a code element needs to contain a bug after one of these most frequent refactoring types. Figure 4.4 presents a histogram for each one of most frequent refactoring types. The first graph corresponds to the Extract Method refactoring type, and the second graph corresponds to the Inline Method refactoring type. The horizontal axis presents the distance values when considering such refactoring types. The vertical axis shows the frequency of each distance values, i.e., how many times such distance value occurred for such refactoring type.

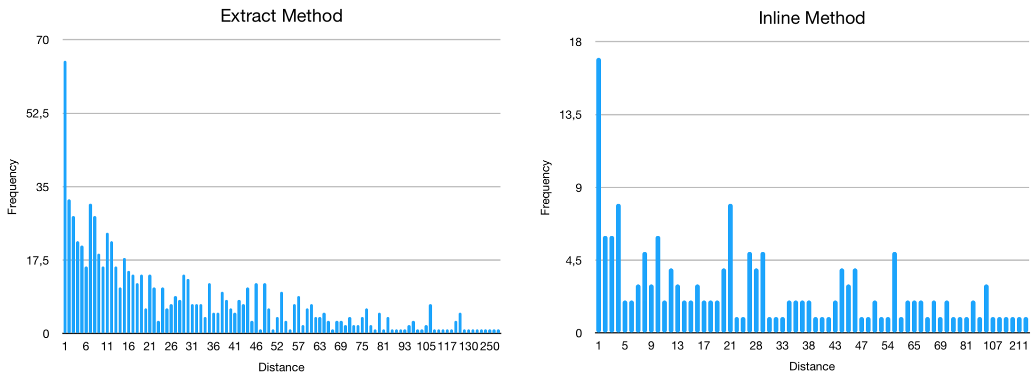


Figure 4.4: Distance of the most frequent refactoring types related to bugs

The graph shows that lower distance values have a higher frequency. That is, for the Extract Method refactoring type, most bugs appear one change away from the refactoring operation. Furthermore, distance values ranging from 2 to 5, and from 7 to 10 also have a considerable frequency. Similarly, for the Inline Method, most bugs appear one change away from the refactoring operation. We could confirm the results found by Bavota et al. (1). The authors state that the Extract Method and the Inline Method refactoring types are harmful and make the refactored code more bug-prone. As we found in our analysis, few changes are necessary so that bugs appear after the application of one of these refactoring types. Thus, we reinforce the results found by Bavota et al. (1) from another perspective. That is, we performed a more fine-grained and systematic analysis that allowed us to show that both properties (frequency and distance) should be used complementary, i.e., if a phenomenon occurs very frequently, then we should measure how close or how far a bug appears after the application of the most frequent refactoring types. Thus, this result gives us a hint that developers should be aware of the application of the Extract Method and the Inline Method refactoring types.

We have also analyzed refactoring types of isolated refactorings that never make code elements bug-prone. As a result, we found that code elements refactored by *Extract Interface*, *Move Class*, *Pull up Attribute*, *Pull down Attribute*, *Pull up Method*, *Push down Method*, and *Rename Class* are never bug-prone. Surprisingly, this result contradicts Bavota et. al (1) since they have found that those refactoring types are one of the most harmful, especially refactoring types involving hierarchies (e.g., Pull up Method, and Pull down Method). Our results show that developers should not worry when applying these refactoring types. Furthermore, the results for *Extract Interface*, *Push down Attribute*, and *Push down Method* confirm the results found by Bavota et al. (1) when the authors mention that these refactoring types are not harmful at all. The authors do not provide any evidence for *Move Class*, and *Rename Class*.

Summary for SRQ$_4$. Extract Method and Inline Method are the most frequent refactoring types when analyzing the bug-proneness of code refactored through isolated refactorings. Furthermore, few changes are necessary after the application of these refactoring types so that the refactored code become buggy. Thus, developers should be aware when applying Extract Method and Inline Method.

4.2.5 Bug-Proneness According to Refactoring Tactics

The goal of this section is to assess the bug-proneness of refactored code according to the refactoring tactics: root-canal and floss refactoring. Thus, we aim to answer *SRQ$_4$. How many times a degraded code element that was refactored has to change to become buggy per refactoring tactic?* to know if there is a difference in the results presented in SRQ$_3$, when considering the refactoring tactics for the analysis of bug reports. Thus, to assess SRQ$_4$, we will measure the property *distance* in the manual validated sample (see

Section 4.1.3). Table 4.9 and 4.10 present the results for root-canal refactoring and floss refactoring, respectively. The first column presents the considered commit of the bug (insertion or report). The second column presents the number of times that a refactored code became buggy per refactoring tactic. The following columns present the minimum value of distance, the quartiles (25%, 50%, and 75%) represented by Q1, Q2, and Q3, respectively, and the maximum value of distance.

Table 4.9: Bug-proneness of code refactored through root-canal refactoring

Bug	**N**	**Min**	**Q1**	**Q2**	**Q3**	**Max**
Insertion	23	1	11.2	20	27	61
Report	74	0	7	19	42.1	87

Table 4.10: Bug-proneness of code refactored through floss refactoring

Bug	**N**	**Min**	**Q1**	**Q2**	**Q3**	**Max**
Insertion	723	1	7	16.5	32	271
Report	2203	0	8	19	38	364

By analyzing the results of floss refactoring, we can see that up to Q2, i.e., 50% of the times that a refactored code is bug-prone, it is necessary less changes than root-canal refactoring so that the refactored code element became buggy. Hence, when developers refactor with other goals, and changes are applied in conjunction with the refactoring, then the code element is more bug-prone than root-canal refactoring in 50% of the times. However, when considering Q3 and the maximum value of distance, we can observe that it is necessary more changes so that the refactored code element became buggy after a floss refactoring than after a root-canal refactoring. Surprisingly, even when developers have the goal to improve the code structure quality, refactored code elements are bug-prone. Our results suggest that depending on the refactoring tactic, the bug-proneness of refactored code differ.

To analyze whether there is any statistically significant difference between the root-canal and floss refactoring, we carried out the Mann-Whitney Wilcoxon (MWW) test, considering a significance level of 0.05. We found a p-value < 0.05, showing that there is a statistical difference between root-canal and floss refactoring, when considering both insertion and report commits. In summary, our results lead us to reject the null hypothesis H_0 and accept the alternative hypothesis HA_1.

Summary for SRQ_5. In 75% of the times that a code element affected by a floss refactoring is bug-prone, it is necessary at least eight changes after the floss refactoring so that the code element become buggy. Surprisingly, when considering 100% of the times that a code element affected by a root-canal refactoring is bug-prone, it is necessary less changes after the root-canal refactoring so that the code element become buggy compared to floss refactoring. Thus, root-canal refactoring is at least as bug-prone as floss refactoring.

4.3 Threats to Validity

We discuss the threats to the study validity (60), with the respective minimizations, as follows.

Construct and Internal Validity.

The code smell types analyzed in this study might not be representative of all the existing variety of code smells. To mitigate this threat, we selected the most common code smell types. Furthermore, the selected code smell types are directly related to refactoring types analyzed in this dissertation (19, 21). The analyses performed in this study are very sensitive to code smell detection rules. Such rules are based on arbitrary thresholds. The risk is that different thresholds can lead to results completely distinct. Therefore, choices of thresholds can pose a threat to this study. To mitigate this risk, we used thresholds previously validated by others researchers (14, 33, 61, 62).

The variety of refactoring types analyzed in our study may not be representative. To mitigate this threat, we selected refactoring types that have been widely applied and investigated by previous studies (1, 6, 19). Furthermore, Refactoring Miner, the tool used to detect refactorings, represents a threat in our study because it may lead to false positives. To mitigate this threat, we choose this tool because previous study (6) has reported high precision. Furthermore, we performed a manual validation to guarantee the reliability of our results (see Section 3.4).

Regarding the refactoring tactics, we could not reach the developers to ask their intentions (root-canal or floss) in all detected refactorings. Therefore, we performed a manual validation to see whether the refactoring is root-canal or floss (see Section 4.1.3). Such analysis is limited to two versions of the source code directed related to the refactoring, not considering all commits in the repository. Moreover, the manual analysis only considers behavior preservation in refactored code elements. Another threat to validity is that we do not consider callers as refactored elements. To mitigate this threat, we consider as refactored code elements all those elements directly affected by a refactoring operation.

The bug detection also represents a threat to validity. We collected bug reports for open source repositories with the tag “bug” or “defect”. However, bug reports might be incorrectly tagged. To minimize this threat, we performed a manual validation of a sample (see Section 3.5). Furthermore, the use of the FindBugs tool represents a threat to validity, because it may have false positive and false negatives on identifying bugs in the source code. To mitigate this threat, we manually validated a sample of the bugs reported by the tool (see Section 3.5). The false positives of the SZZ heuristic may represent a threat to internal validity. To minimize this threat, we used a combination of improved heuristics proposed by (12) and (13).

Conclusion and External Validity.

Regarding the generalization of our findings, there are threats which could limit our study findings of being applied in different software development contexts. In fact, our study relies on 12 Java open source software projects only, which may be a low number of projects for an empirical study of this nature. We minimize possible threats to validity by analyzing software projects from different domains, with varying sizes, and supported by active issue tracking systems. In addition, the low number of samples may reduce the ability to reveal patterns in the data. To mitigate we carefully conducted our quantitative data analysis by validating the analyzed data whenever possible. We also used well-known statistical measures, such as *quartiles*, which have been used by previous study (31) in the context of code refactoring.

4.4 Final Remarks

In this chapter, we investigated the bug-proneness of code refactored through isolated refactoring. For this purpose, we presented a longitudinal study aimed at investigating the characteristics of refactorings that make the source code bug-prone. For that, we assessed the two complementary properties of the bug-proneness of refactored code, i.e., frequency and distance.

As a result, we found that the refactorings performed in code elements that are bug-prone touched mostly in code elements with either a single smell or multiple code smells. In addition, we found that it is necessary at least nine changes (performed in around 3 months) after the refactoring so that the refactored code element become buggy. Furthermore, our fine-grained and systematic analysis allowed us to show that it is necessary a considerable number of changes after the application of the most frequent refactoring types performed in code elements that are bug-prone. Finally, our results suggest that, surprisingly, code elements refactored through root-canal refactoring can be also bug-prone. The results found in this study contradicts the literature (1, 3) in many different ways. The next chapter presents and discusses the results of the second study conducted in this dissertation.

5
The Bug-Proneness of Code Refactored through Batch Refactoring

Due to the diversity of purposes behind refactoring, researchers have been trying to understand how and why developers perform refactoring (18, 35, 52). These studies report the most common refactoring types applied by developers (18), the typical reasons underlying refactoring types (35), and how developers use tools that help to refactor (52). Among other findings, these studies briefly shed light upon the existence of *batch refactorings*, i.e., when developers apply a sequence of refactoring operations.

After studying the bug-proneness of code refactored through isolated refactoring (presented in Chapter 4), there is a need to empirically investigate the bug-proneness of code elements refactored through batch refactoring. Previous work (1, 3) overlook the effects of batch refactorings on the bug-proneness of refactored code elements. As refactoring might be risky as any other change in the source code, one could say that if more refactorings are applied to a code element, then more bug-prone the code elements are. However, this is unknown since the literature does not explore this phenomenon. Thus, this chapter has the goal to sweep an unexplored terrain between batch refactorings and bugs. For that, we consider batches as one of the characteristics of refactorings that make refactored code bug-prone. Hence, we analyze the complementary properties of the bug-proneness of refactored code, i.e., *frequency* and *distance*, in the context of batch refactorings.

The remainder of this chapter is organized as follows. Section 5.1 describes the study settings, including the study goal and research questions. Section 5.2 presents the results of our empirical study regarding the bug-proneness of code refactored through batch refactoring. Section 5.3 discusses threats to the validity. Section 5.4 summarizes this chapter and introduces the following chapter.

5.1
Study Settings and Procedures

The common study procedures for all studies performed in this dissertation is described in Chapter 3. This section describes the specific settings and procedures of the study aimed at understanding the bug-proneness of code refactored through batch refactoring. The remainder of this section is organized as follows. Section 5.1.1 presents the research questions, and associated hypotheses. Section 5.1.2 describes how we identify batch refactorings. Finally, Section 5.1.3 presents how we measure the bug-proneness of code refactored through batch refactoring regarding the frequency and the distance properties.

5.1.1
Research Questions

The study presented in this chapter intends to investigate the bug-proneness of code refactored through batch refactoring. From our study goal, we designed the following specific research questions (SRQs).

SRQ_6. How many batch refactorings were performed in code elements that are bug-prone?

Our goal is to measure the bug-proneness of code refactored through batch refactoring. Thus, it is essential to understand how many batch refactorings were performed in code elements that are bug-prone. As a result of this research question, we can then perform further analysis to assess how this phenomenon occurs.

SRQ_7. Are code elements refactored through batch refactorings less bug-prone than non-refactored code elements?

As a result of SRQ_6, we know how many batch refactorings were performed in code elements that are bug-prone. Thus, this research question aims at investigating whether code elements refactored through batch refactorings are less bug-prone than non-refactored code elements. For that, we analyze one of the complementary properties of the bug-proneness of refactored code elements, i.e., *frequency*. Similar to SRQ_2, we only consider degraded code elements, since developers often apply refactorings on degraded code elements (6). Thus, with the results of SRQ_6, we will be able also to compare the *frequency* property between isolated refactorings and batch refactorings. We derived our null (H_0) and alternative hypotheses (HA_1) from SRQ_7 as presented in Table 5.1.

Table 5.1: Study hypotheses derived from SRQ_7

Hypotheses	Description
H_0	There is no difference in the bug-proneness of code elements that have undergone batch refactorings and non-refactored code elements.
HA_1	There is a difference in the bug-proneness of code elements that have undergone batch refactorings and non-refactored code elements.

SRQ₈. Does the bug-proneness of degraded code elements decrease when applying batch refactoring?

High levels of code degradation require the application of batch refactorings. Otherwise, it might not be possible to fully eliminate the code degradation and, therefore, reduce its bug proneness. Recent studies (18, 35, 52) suggest that applying batch refactoring is a more recurring phenomenon that researchers could expect. Due to the aforementioned assumption that every isolated refactoring might decrease the bug-proneness of degraded code elements, one could assume the following: the more refactorings developers apply on a degraded code element, the less bug-prone such code element becomes compared to only one refactoring. We confirm or refute this expectation through SRQ_8 by comparing the distance values of batch refactorings versus isolated refactorings. We derived our null (H_0) and alternative hypotheses (HA_1) from SRQ_8 as presented in Table 5.2.

Table 5.2: Study hypotheses derived from SRQ_8

Hypotheses	Description
H_0	There is no difference in the bug-proneness regarding distance when applying batch refactorings or isolated refactorings.
HA_1	There is a difference in the bug-proneness regarding distance when applying batch refactorings or isolated refactorings.

5.1.2 Identification of Batch Refactorings

After collecting all refactorings performed in all analyzed software projects (see Section 2.1.1), we need to identify batch refactorings to assess the bug-proneness of code refactored through batch refactorings. Murphy Hill et al. (18) mention that developers often refactor multiple pieces of code since several code elements need to be refactored in order to perform a complete refactoring. This result suggests that batches have more than one refactoring operation and that they should share a code element. Furthermore, the authors (18) mine CSV. As CVS does not record which file revisions were committed in a single transaction, the authors proposed an approach for finding revisions committed by the same developer with the same commit message. This indicates that batches should be performed by the same developer. Then, we developed a heuristic to identify batch refactorings. In this study, we define as a batch if (i) the refactorings have at least one code element in common (18, 40); (ii) all refactorings in the batch must have been performed by the same developer; (iii) the batch must have more than one refactoring operation. After identifying each batch refactoring, we sort the refactoring operations chronologically. The sorting depends if the batch was identified in the same commit or in different commits. In the case that we have a batch in the same commit (see batch b_1 in Figure 2.1), the order is arbitrary. When a batch is identified in different commit (see batch b_2 in Figure 2.1), we sort the refactorings according to the commit order.

Definition 2.
A batch refactoring $b = [r_1, r_2, ..., r_n]$ is a sequence of *size* $n \geq 2$ refactorings r_i, for $1 \leq i \leq n$, such that:

- $Order(r_j) \leq Order(r_{j+1})$, for $1 \leq j < n$, where $Order(r_j)$ is the *order*, in the program commit history, of the commit that includes r_j
- $\forall j(Programmer(r_j) = u)$, for $1 \leq j \leq n$, where $Programmer(r_j)$ is the *programmer* that has performed r_j and u is a specific programmer
- $\forall j(Element(r_j) = e)$, for $1 \leq j \leq n$, where $Element(r_j)$ is the *program element* affected by r_j and e is a specific element

5.1.3 Measuring the Properties of the Bug-Proneness of Refactored Code

Figure 5.1 presents how we measure the bug-proneness of code refactored through batch refactoring. The figure represents a timeline regarding the commit history of a software project. For illustration purpose, we will only show events that happened on *method X*. As we can see, a *batch* composed of two refactorings happened in $\{C3, C5\}$. Furthermore, method X was *smelly* in C1, a bug was *introduced* on it in C7, and a *bug report* was opened in C9. Additionally, changes were performed in method X on C2, C4, C6, and C8. Finally, the bug was *fixed* in method X in C10. Sections 5.1.4 and 5.1.5 present how we measure the frequency and distance properties.

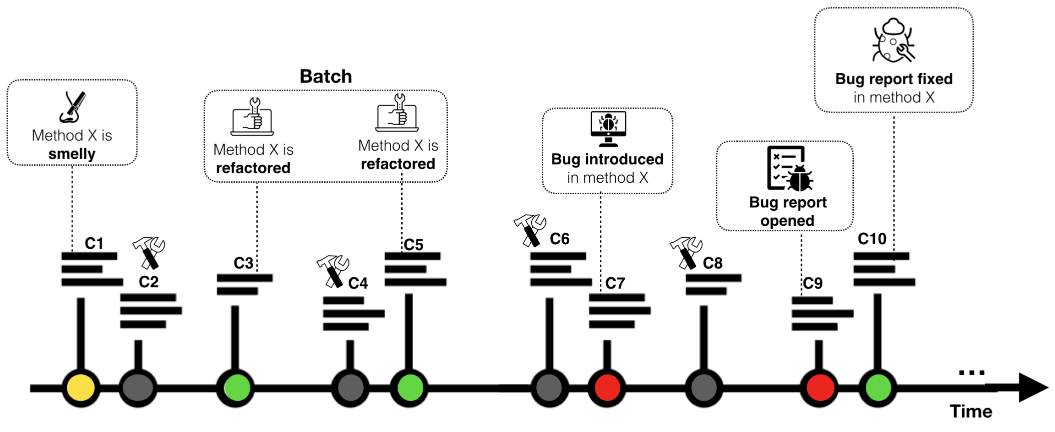


Figure 5.1: Evaluation of the bug-proneness of code refactored through batch refactoring

5.1.4 Measuring the *Frequency* Property

To answer SRQ_6, we count how many batch refactorings were performed in code elements that are bug-prone. For that, we consider the first refactoring that is part of the batch. We chose the first refactoring due to the fact that the subsequent refactorings that are part of the batch could have made the code element bug-prone. For instance, in Figure 5.1, we consider the first refactoring of the batch, i.e., the refactoring operation performed in C3, and then we verify whether a bug has been either introduced or reported in further commits. Observe that if we had considered the last refactoring operation of the batch, i.e., the refactoring operation performed in C5, we would have missed a bug that could have been inserted and reported in C4.

To compare the frequency that code elements refactored through batch refactorings are less bug-prone than non-refactored code elements (SRQ_7), we count the number of buggy code elements that belong to a batch refactoring according to the number of code smells on it. For that, we look back from the first refactoring operation part of the batch that had a bug in further commits and count how many code smells the refactored code element had. Thus, in Figure 5.1, method X had one code smells in C1, before the first refactoring operation part of the batch (performed in C3).

5.1.5 Measuring the *Distance* Property

To measure the bug-proneness of code refactored through batch refactoring considering the *distance* property (SRQ_8), we count the number of changes between the commit of the first refactoring part of the batch and the commit in which the bug was either introduced or reported. That is, in Figure 5.1, two changes were performed in method X in $\{C4, C6\}$ from the first refactoring part of the batch (C3) to the bug introduction (C7). Furthermore, if we consider the commit in which the bug was reported, the example shows that three changes were performed ($\{C4, C6, C8\}$) from the commit in which the first refactoring part of the batch was performed (C3) to the commit in which the bug was reported (C9). Note that if we have not considered the first refactoring part of the batch, we would have missed the change performed in C4, which could be the one responsible for the bug occurrence.

5.2 The Bug-Proneness of Code Refactored through Batch Refactoring

This section shows the results that answer SRQ_6 (Section 5.2.1), SRQ_7 (Section 5.2.2), and SRQ_8 (Section 5.2.3).

5.2.1 Frequency of Batch Refactorings Performed in Code Elements that Are Bug-Prone

In this section, we assess the number of batch refactorings performed in code elements that are bug-prone, by answering SRQ_6, which asks *How many batch refactorings were performed in code elements that are bug-prone?*. Table 5.3 presents the frequency of batch refactorings performed in code elements that are bug-prone. The first column lists the software projects analyzed in this study. The second column presents the total number of batch refactorings found for each software project. The third column shows the number of batch refactorings performed in code elements that are bug-prone. Finally, the fourth column presents the number of batch refactorings performed in code elements that are not bug-prone at all. Furthermore, highlighted cells in the table indicate that we found a non-ignorable frequency (more than 10%) of batch refactorings performed in code elements that are bug-prone.

Table 5.3: Frequency of bug-prone batch refactorings

Software Project	**# Batch Refactorings**	**# Batch Refactorings with Bugs in the Future**	**# Batch Refactorings without Bugs in the Future**
Ant	331	2 (0.60%)	329 (99.40%)
Derby	4,407	3 (0.07%)	4,404 (99.93%)
Elasticsearch	816	72 (8.82%)	744 (91.18%)
Elasticsearch-hadoop	242	0 (0.00%)	242 (100%)
ExoPlayer	410	6 (1.46%)	404 (98.54%)
Fresco	18	1 (5.56%)	17 (94.44%)
Material-dialogs	18	6 (33.33%)	12 (66.67%)
Netty	766	12 (1.57%)	754 (98.43%)
Okhttp	66	5 (7.58%)	61 (92.42%)
Presto	250	4 (1.60%)	246 (98.40%)
Spring-boot	130	21 (16.15%)	109 (83.85%)
Tomcat	374	5 (1.34%)	369 (98.66%)
Total	7,828	137 (1.75%)	7,691 (98.25%)

The results presented in Table 5.3 show that more than 15% of the batches performed in the projects *Spring-boot* and *Material-dialogs* were performed in code elements that are bug-prone. However, in the other analyzed projects, only a few batch refactorings were performed in code elements that are bug-prone (varying from 0.07% to 8.82%). In the overall analysis for all software projects, only 1.75% of the batch refactorings were performed in code elements that are bug-prone, against 98.25% of the batch refactorings performed in code elements that are not bug-prone at all. By comparing these results with the ones found in SRQ$_1$, we can see that code refactored through isolated refactorings are more bug-prone than code refactored through batch refactorings.

Summary for SRQ$_6$. Our results show only 1.75% of the batch refactorings were performed in code elements that are bug-prone, against 98.25% of batch refactorings that were performed in code elements that are not bug-prone at all.

5.2.2 Bug-Proneness of Refactored Code *versus* Non-Refactored Code

In this section, we aim to answer SRQ$_7$, which asks *Are code elements refactored through batch refactorings less bug-prone than non-refactored code elements?* by assessing the frequency that code elements refactored through batches are bug-prone against non-refactored code elements. For that, we considered degraded code elements according to two categories: (i) single smell, and (ii) multiple smells. Table 5.4 presents the results per software project. The first column presents the software projects. The second and third columns present the frequency of code elements (i) hosting either single or multiple smells, (ii) weren't refactored by a batch, and (iii) became buggy. The fourth and fifth columns show the frequency of code elements (i) hosting either single or multiple smells, (ii) were refactored by a batch and (iii) became buggy. Finally, the last column presents the proportion of degraded code elements that were refactored and became buggy (summing up the values of the fourth and fifth columns), and degraded code elements that became buggy and weren't refactored (summing up the values of the second and third columns). By measuring the proportion, we are able to see the percentage of degraded code elements that were refactored and became buggy. The equation below computes such proportion. Furthermore, highlighted cells in the table indicate that we found a non-ignorable frequency (more than 10%) for such cases of analysis.

$$proportion = \frac{\#degraded\ \&\ refactored\ \&\ buggy\ code}{\#degraded\ \&\ non{-}refactored\ \&\ buggy\ code}$$

As we can see in the table, most software projects have a non-ignorable frequency of code elements that weren't refactored by a batch, considering either single smell or multiple smells (second and third column, respectively). Some projects have a high frequency of code elements containing multiple code smells, that weren't refactored by a batch refactoring and became buggy, i.e., Derby (84.07%), and Fresco (98.29%). Furthermore, some projects have a higher frequency of code elements containing a single smell, i.e., Tomcat (75.00%). On the contrary, most software projects do not have a considerable value of frequency for code elements containing a single smell that were refactored by a batch and became buggy. Moreover, 5 out of 12 software projects contain a non-ignorable frequency of code elements containing multiple smells that were refactored by a batch and became buggy.

Table 5.4: Frequency of the bug-proneness of refactored code vs. non refactored code per software project (batch refactorings)

	No Refactoring		Refactoring		
Projects	**Single smell & Buggy**	**Multiple smells & Buggy**	**Single smell & Buggy**	**Multiple smells & Buggy**	**Proportion**
Ant	6.25%	71.88%	0.00%	21.88%	28.00%
Derby	14.29%	84.07%	0.00%	1.65%	1.68%
Elasticsearch	0.00%	0.00%	0.00%	0.00%	-
Elasticsearch-hadoop	0.00%	0.00%	0.00%	0.00%	-
ExoPlayer	0.00%	0.00%	0.00%	0.00%	-
Fresco	0.00%	98.29%	1.71%	0.00%	1.74%
Material-dialogs	0.00%	100%	0.00%	00.00%	00.00%
Netty	49.30%	11.27%	2.82%	36.62%	65.12%
Okhttp	11.76%	64.71%	0.00%	23.53%	30.77%
Presto	37.50%	52.50%	0.00%	10.00%	11.11%
Spring-boot	11.92%	43.05%	0.00%	45.03%	81.93%
Tomcat	75.00%	24.56%	0.00%	0.44%	0.44%
Total	**41.88%**	**45.87%**	**0.38%**	**11.87%**	**14.00%**

Regarding the proportion of degraded code elements that were refactored and became buggy, and degraded code elements that weren't refactored and became buggy, we can see that in 5 out of 12 software projects presented a non-ignorable value of proportion. Overall, only 14.00% of the degraded code elements were refactored before becoming buggy. That is, 86.00% of the degraded code elements weren't refactored before becoming buggy. This result shows that degraded and non-refactored code is more bug-prone than degraded and refactored code. Figure 5.2 presents a bar graph that shows the frequency of buggy elements that were refactored multiple times or not according to the number of code smells in code elements before the refactoring operation.

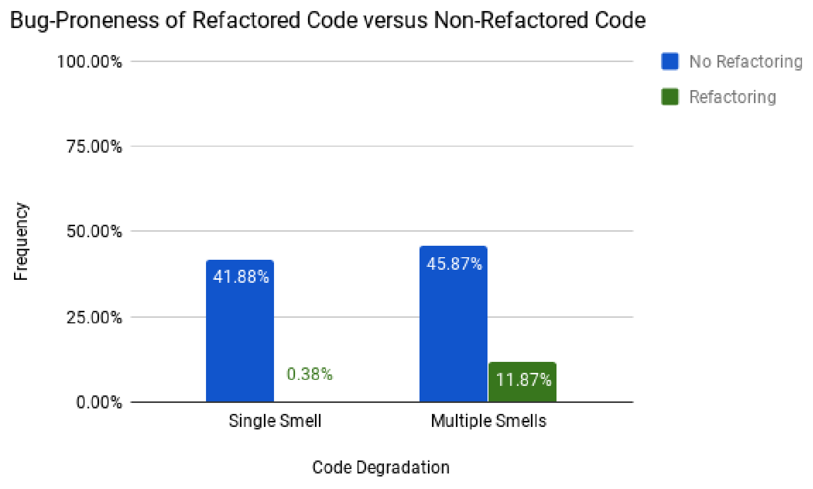


Figure 5.2: Frequency of the bug-proneness of refactored code vs. non-refactored code (batch refactorings)

When analyzing the code degradation regardless the refactoring operation, i.e., summing up the results for single smells (left side of the bar graph) and multiple smells (right side of the bar graph), our results show that 57.74% of the buggy code elements contain multiple smells against 42.26% that contain single smells before the first refactoring part of the batch. This result is interesting because it shows that buggy code elements were highly structurally degraded. Furthermore, if we consider the refactoring operation regardless the code degradation, i.e., summing up both blue bars and green bars, our results suggest that 87.75% of the buggy code elements have not been refactored before they become buggy, regardless the code degradation. That is, no refactoring operation was applied before the code element becomes buggy in the software system. In this case, applying a batch refactoring on smelly elements could have avoided bugs in these elements. On the contrary, 12.25% of the buggy elements were refactored before they become buggy.

We applied the Fisher's test to compute the strength of the relation between batch refactoring and degraded code with bugs (9). Furthermore, we used the Odds Ratio (10) to compute the possibility of the presence or absence of a phenomenon (i.e., batch refactoring) to be associated with the presence or absence of the other phenomenon (i.e., degraded code with bugs). Considering all projects analyzed, we found a p-value less than 0.05, and Odds Ratio equals to 0.6881. Thus, our results show that the possibility of a batch refactoring be related to a degraded code with bugs is 0.68 if compared to non-refactoring related to degraded code with bugs. In summary, our results lead us to reject the null hypothesis H_0 and accept the alternative hypothesis HA_1.

Summary for SRQ_7. 87.75% of the degraded code elements that became buggy were not previously refactored by a batch refactoring. This result shows that degraded, non-refactored code elements tend to be more bug-prone than degraded, refactored code elements.

5.2.3 Distance Between the Refactoring and the Bug

In this section, we aim at analyzing the bug-proneness of code elements refactored through batch refactorings regarding the *distance* property. The result will allow us to answer *SRQ_8. Does the bug-proneness of degraded code elements decrease when applying batch refactoring?*. Table 5.5 shows the results of distance values for the insertion and report commit of the bug. We applied the Grubb outlier test ($\alpha = 0.05$), and we removed outliers for both insertion and report commit data. Thus, the results found to represent a key factor to provide confidence in the results reported in this work.

Table 5.5: Bug-proneness in distance for batch refactorings

Bug	N	Min	Q1	Q2	Q3	Max
Insertion	111	1.00	7.50	29.00	52.50	123.00
Report	120	0.00	8.00	27.00	51.25	191.00

As we can see in the table, in 75% of the times, a degraded code element needs at least eight changes to became buggy after a batch refactoring. One could say that if the code element is structurally degraded, and more refactorings are applied to it (batches), then the code is more bug-prone than when a code element is refactored only once. However, we surprisingly found that if more refactorings are applied to a degraded code element, then it is less bug-prone.

Similar to the analysis of isolated refactorings (Section 4.2.3), it took three months so that the refactored code element became buggy. If we compare the bug-proneness of code refactored through isolated refactorings and code refactored through batch refactorings, we can see that the distance starts to vary from Q2. For instance, 50% of times that isolated refactorings are bug-prone, it is necessary up to 21 changes so that the refactored code element become buggy. However, when considering batch refactoring, it is necessary up to 29 changes so that the refactored code element becomes buggy. This result suggests that applying batches might contribute to make the code element less bug-prone. That is, when applying batch refactorings, it is necessary more changes so that the code element become buggy if compared to isolated refactorings.

To analyze whether there is any statistically significant difference between the bug-proneness of code refactored through isolated refactoring and the bug-proneness of code refactored through batch refactoring, we carried out the Mann-Whitney Wilcoxon (MWW) test, considering a significance level of 0.05. Considering the insertion commit, we found a p-value < 0.05, showing that there is a statistical difference between the bug-proneness of code refactored through isolated refactoring and the bug-proneness of code refactored through batch refactoring. However, when considering the report commit, we found a p-value > 0.05. That is, there is no statistical difference between the bug-proneness of code refactored through isolated refactoring and batch refactoring for the report commit. In summary, our results lead us to reject the null hypothesis H_0 and accept the alternative hypothesis HA_1.

Summary for SRQ$_8$. In 75% of the times that code elements refactored through batch refactorings are bug-prone, a code element needs least eight changes after a batch refactoring so that the code element becomes buggy. However, when comparing isolated and batch refactoring, in 75% of the times that refactored code elements are bug-prone, it is necessary more changes so that the code element becomes buggy after a batch refactoring than after an isolated refactoring.

5.3 Threats to Validity

In this section, we discuss the threats to the study validity (60), with the respective minimizations. This study has some common threats to validity to the first study conducted in this dissertation (see Chapter 4). The mitigation of such threats was the same as the previous study. We describe them as follows.

Construct and Internal Validity.

The code smell types analyzed in this study might not be representative. Furthermore, the selected code smell types are directly related to refactoring types analyzed in this dissertation (10, 21). Also, the analyses performed in this study are very sensitive to code smell detection rules. Such rules are based on arbitrary thresholds. Therefore, choices of thresholds are threats to this study. The mitigation of such threats was the same as the previous study (see Chapter 4). That is, we selected the most common code smell types. Furthermore, we used thresholds previously validated by others researchers (14, 33, 61, 62).

The Refactoring Miner tool used to detect refactorings represents a threat in our study because it may lead to false positives. We mitigate this threat similar to the study performed in Chapter 4. That is, we choose this tool because previous study (6) has reported high precision. Furthermore, we performed a manual validation to guarantee the reliability of our results (see Section 3.4). Also, to measure the distance property, we take into account the first refactoring part of the batch. This is a treat to validity since the chosen refactoring will influence the results of the bug-proneness of code refactored through batch refactoring. To mitigate this threat, we also measured the distance property from the last refactoring that is part of the batch. We could not find any significant difference in the results of the distance property considering the first and the last refactoring part of the batch.

The bug detection also represents a threat to validity. Furthermore, the use of the FindBugs tool represents a threat to validity, because it may have false positive and false negatives on identifying bugs in the source code. Moreover, the false positives of the SZZ heuristic may represent a threat to internal validity. We mitigate this threat similar to the study performed in Chapter 4. That is, we performed a manual validation of a sample of bug reports collected in this study (see Section 3.5). We also manually validated a sample of the bugs reported by Findbugs (see Section 3.5). Furthermore, we used a combination of improved heuristics of SZZ proposed by (12) and (13).

Similar to Chapter 4, another threat to validity is the changes detected in our study. We identify changes at the file level. It might be the case that the change is not an actual change in the source code. It might be the case that the developer just commented something in the source code, or did a line break.

Conclusion and External Validity.

Similar to Chapter 4, there are threats which could limit our study findings of being applied in different software development contexts. Our study relies on 12 Java open source software projects only, which may be a low number of projects for an empirical study of this nature. We minimize possible threats to validity by analyzing software projects from different domains, with varying sizes, and supported by active issue tracking systems. In addition, the low number of samples may reduce the ability to reveal patterns in the data. To mitigate we carefully conducted our quantitative data analysis by validating the analyzed data whenever possible. We also used well-known statistical measures, such as *quartiles*, which have been used by previous study (31) in the context of software refactoring.

5.4 Final Remarks

In this chapter, we investigated the bug-proneness of code refactored through batch refactoring. We presented a study aimed at investigating an unexplored terrain of the bug-proneness of code refactored through batch refactoring. For that, we assessed batches as one of the characteristics of refactorings that make the source code bug-prone. Additionally, we assessed the complementary properties of the bug-proneness of refactored code. We found that only 1.75% of the batch refactorings were performed in code elements that became buggy. By comparing these results with the ones found in the previous study, we can see that code refactored through isolated refactorings are more bug-prone than code refactored through batch refactoring. Finally, we found that most of the times that code elements refactored through batches are bug-prone, it is necessary more changes so that the code element becomes buggy compared to isolated refactorings. The aforementioned findings were not previously found in the literature.

6
Conclusion and Future Work

Previous studies (1, 3) neither investigate the bug-proneness of refactored code in depth, nor the influence of refactoring tactics on bugs. In fact, these studies (1, 3) provided limited evidence that refactored code is often susceptible to contain bugs. That is, these studies (1, 3) only assess whether refactored code elements contain a bug in further commits. This analysis might not suffice to blame refactorings for bugs since the refactored code element could have become buggy very far from the refactoring.

To address the aforementioned limitations, this dissertation proposes a more fine-grained and systematic evaluation of the bug-proneness of refactored code. Hence, we performed two longitudinal multi-project studies to assess the bug-proneness of refactored code. The first study concerns assessing the bug-proneness of code refactored through isolated refactorings. Furthermore, the second study concerns assessing the bug-proneness of code refactored through batch refactorings. Besides conducting two longitudinal multi-project studies, we also proposed two complementary properties to assess the bug-proneness of refactored, namely frequency and distance. These properties allowed us to assess the different characteristics of refactorings that make refactored code elements bug-prone, i.e., refactoring types, refactoring tactics, isolated refactorings, and batch refactorings. Our studies involved 12 Java open source projects, 39,750 refactorings, including 21,217 isolated refactorings and 7,828 batch refactorings, 2,119 refactorings manually validated by refactoring tactic, 6,051 bug reports, and 49,250 bugs via static analysis.

We answer our general research question, which asks *What are the characteristics of refactorings that make code elements bug-prone?*, by relying on the findings of our studies as follows. First, we found that only 5.38% of isolated refactorings are bug-prone against 94.62% of isolated refactorings that are not bug-prone at all. Additionally, 62.71% of the isolated refactorings that are not bug-prone touched in code elements without any code smell. Similarly, 63.14% of the isolated refactorings that are bug-prone touched in code elements with either a single smell or multiple smells. This result shows that the refactoring might not have sufficed to fully overcome the degradation in the source code, and bug(s) emerged in future commits.

Second, 79.67% of the degraded code elements that became buggy were not previously refactored. This result shows that degraded, non-refactored code elements tend to be more bug-prone than degraded, refactored code elements. Third, in 75% of the times that refactored code elements are bug-prone when analyzing isolated refactorings, it is necessary at least seven changes after isolated refactorings so that the code element become buggy. Additionally, these seven changes were performed in approximately three months between the isolated refactoring and the bug. As a result, refactored code elements are often not susceptible to immediately contain bugs. We have manually analyzed 25% of the bug-prone refactored code elements, and we could not find any explicit case of bug-prone refactored code, as previous work suggests (1). On the contrary, the changes performed between isolated refactorings and bugs are more likely to be bug-prone.

Fourth, we found that the Extract Method and Inline Method are the most frequent refactoring types when analyzing the bug-proneness of code refactored through isolated refactorings. Furthermore, few changes are necessary after the application of these refactoring types so that the refactored code become buggy. Thus, developers should be aware when applying Extract Method and Inline Method in order to do not make refactored code elements bug-prone.

Fifth, in 75% of the times that a code element affected by a floss refactoring is bug-prone, it is necessary at least eight changes after the floss refactoring so that the code element become buggy. Surprisingly, when considering 100% of the times that a code element affected by a root-canal refactoring is bug-prone, it is necessarily fewer changes after the root-canal refactoring so that the code element become buggy compared to floss refactoring. Thus, root-canal refactoring is at least as bug-prone as floss refactoring.

Sixth, we found that 1.75% of the batch refactorings are bug-prone against 98.25% of batch refactorings that are not bug-prone at all. Seventh, 87.75% of the degraded code elements that became buggy were not previously refactored by a batch refactoring. This result shows that degraded and non-refactored code elements tend to be more bug-prone than degraded and refactored code elements.

Finally, in 75% of the times that refactored code elements are bug-prone when analyzing batch refactorings, it is necessary at least to eight changes after a batch refactoring so that the code element become buggy. However, when comparing isolated and batch refactoring, in 75% of the times that refactored code elements are bug-prone, it is necessary more changes so that the code element become buggy after a batch refactoring than after an isolated refactoring.

In conclusion, the conducted empirical evaluation demystify the literature results, by showing that refactored code is not often susceptible to contain bugs. In fact, if we had only analyzed if a refactored code contains a bug in further commits, as previous studies have done, we would confirm their results. However, by considering the distance property, we were able to see that a refactored code element is changed many times so that this code element become buggy. Our results allowed us to identify characteristics and make recommendations to developers, as follows. We conclude that an isolated refactoring applied to degraded code elements tend not to suffice to reduce their bug-proneness. This observation leads us to our first recommendation.

Recommendation 1. Developers should apply complementary refactorings to fully remove the code degradation and do not make refactored code elements bug-prone.

In addition, we observe that surprisingly, root-canal refactoring is at least as bug-prone as floss refactoring. This observation leads us to our second recommendation.

Recommendation 2. Developers should be aware of not making the refactored code bug-prone even when applying root-canal.

Furthermore, we observe that batch refactoring tends to postpone bugs in refactored code elements. This observation leads us to our third recommendation.

Recommendation 3. Batches can be recommended to reduce the bug-proneness of refactored code.

Moreover, we observe that bugs appear very close to the refactoring operation when the Extract Method and Inline Method refactoring types are applied. This observation leads us to our fourth recommendation.

Recommendation 4. Developers should be aware when applying Extract Method, and Inline Method refactoring types since code elements refactored by these refactoring types are often susceptible to contain bugs immediately.

Finally, we observe that the distance property indeed provide a means to measure the bug-proneness. This observation leads us to our fifth recommendation.

Recommendation 5. A recommender system should take into account the distance property when warning developers about the bug-proneness of refactored code.

As future work, we intend to assess the bug-proneness of regular changes, e.g., line additions. After that, we can compare the bug-proneness of refactored code against the bug-proneness of code elements that had regular changes. This analysis will allow us to see if other types of changes make the source code bug-prone. Furthermore, it is interesting to assess the bug-proneness of refactored code in proprietary software systems. Our studies focused only on the analysis of popular open source projects, which may have different structural degradation, different types of bugs and refactorings as compared to proprietary software systems. Finally, we intend to recommend practices to motivate software developers to refactor and improve state-of-the-art refactoring tools.

Bibliography

[1] BAVOTA, G.; DE CARLUCCIO, B.; DE LUCIA, A.; DI PENTA, M.; OLIVETO, R. ; STROLLO, O.. **When does a refactoring induce bugs? an empirical study.** In: 12TH SCAM, p. 104–113, 2012.

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

[3] WEISSGERBER, P.; DIEHL, S.. **Are refactorings less error-prone than other changes?** In: 3RD MSR, p. 112–118, 2006.

[5] FERREIRA, I.; FERNANDES, E.; CEDRIM, D.; UCHÔA, A.; BIBIANO, A. C.; GARCIA, A.; CORREIA, J. L.; SANTOS, F.; NUNES, G.; BARBOSA, C.; FONSECA, B. ; DE MELLO, R.. **The buggy side of code refactoring: Understanding the relationship between refactorings and bugs.** International Conference on Software Engineering, 2018.

[6] CEDRIM, D.; GARCIA, A.; MONGIOVI, M.; GHEYI, R.; SOUSA, L.; DE MELLO, R.; FONSECA, B.; RIBEIRO, M. ; CHÁVEZ, A.. **Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects.** In: 11TH FSE, p. 465–475, 2017.

[7] FERNANDES, E.; VALE, G.; SOUSA, L.; FIGUEIREDO, E.; GARCIA, A. ; LEE, J.. **No code anomaly is an island.** In: 16TH ICSR, p. 48–64, 2017.

[8] OIZUMI, W.; GARCIA, A.; DA SILVA SOUSA, L.; CAFEO, B. ; ZHAO, Y.. **Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems.** In: 38TH ICSE, p. 440–451, 2016.

[9] FISHER, R.. **On the interpretation of χ 2 from contingency tables, and the calculation of p.** J. Royal Stat. Soc., 85(1):87–94, 1922.

[10] CORNFIELD, J.. **A method of estimating comparative rates from clinical data. applications to cancer of the lung, breast, and cervix.** J. Nat. Cancer Inst., 11(6):1269–1275, 1951.

[11] WEISSGERBER, P.; DIEHL, S.. **Are refactorings less error-prone than other changes?** In: 3RD MSR, p. 112–118, 2006.

[12] KIM, S.; ZIMMERMANN, T.; PAN, K.; JAMES JR, E. ; OTHERS. **Automatic identification of bug-introducing changes.** In: 21ST ASE, p. 81–90, 2006.

[13] WILLIAMS, C.; SPACCO, J.. **Szz revisited: verifying when changes induce fixes.** In: 8TH DEFECTS, p. 32–36, 2008.

[14] LANZA, M.; MARINESCU, R.. **Object-oriented metrics in practice.** Springer Science & Business Media, 2007.

[15] ARCOVERDE, R.; MACIA, I.; GARCIA, A. ; VON STAA, A.. **Automatically detecting architecturally-relevant code anomalies.** In: 3RD RSSE, p. 90–91, 2012.

[16] LANZA, M.; MARINESCU, R.. **Object-oriented metrics in practice.** Springer Science & Business Media, 2007.

[17] MACIA, I.; ARCOVERDE, R.; GARCIA, A.; CHAVEZ, C. ; VON STAA, A.. **On the relevance of code anomalies for identifying architecture degradation symptoms.** In: 16TH CSMR, p. 277–286, 2012.

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

[19] FOWLER, M.. **Refactoring.** Addison-Wesley Professional, 1999.

[20] TSANTALIS, N.; GUANA, V.; STROULIA, E. ; HINDLE, A.. **A multidimensional empirical study on refactoring activity.** In: 13TH CONFERENCE OF THE CENTER FOR ADVANCED STUDIES ON COLLABORATIVE RESEARCH (CASCON), p. 132–146, 2013.

[21] MACIA, I.; ARCOVERDE, R.; GARCIA, A.; CHAVEZ, C. ; VON STAA, A.. **On the relevance of code anomalies for identifying architecture degradation symptoms.** In: CSMR12, p. 277–286, March 2012.

[22] GRUBBS, F.. **Procedures for detecting outlying observations in samples.** Technometrics, 11(1):1–21, 1969.

[23] HERZIG, K.; JUST, S. ; ZELLER, A.. **It's not a bug, it's a feature: how misclassification impacts bug prediction.** In: 35TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 392–401, 2013.

[24] ŚLIWERSKI, J.; ZIMMERMANN, T. ; ZELLER, A.. **When do changes induce fixes?** In: ACM SIGSOFT SOFTWARE ENGINEERING NOTES (SEN), volumen 30, p. 1–5, 2005.

[25] DALLMEIER, V.; ZIMMERMANN, T.. **Extraction of bug localization benchmarks from history.** In: 22ND INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), p. 433–436, 2007.

[26] YE, X.; BUNESCU, R. ; LIU, C.. **Learning to rank relevant files for bug reports using domain knowledge.** In: 22ND FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 689–699, 2014.

[27] DA COSTA, D. A.; MCINTOSH, S.; SHANG, W.; KULESZA, U.; COELHO, R. ; HASSAN, A. E.. **A framework for evaluating the results of the szz approach for identifying bug-introducing changes.** IEEE Transactions on Software Engineering (TSE), 43(7):641–657, 2017.

[28] KIM, S.; ZIMMERMANN, T.; PAN, K.; JAMES JR, E. ; OTHERS. **Automatic identification of bug-introducing changes.** In: 21ST INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE), p. 81–90, 2006.

[29] WILLIAMS, C.; SPACCO, J.. **Szz revisited: verifying when changes induce fixes.** In: 8TH WORKSHOP ON DEFECTS IN LARGE SOFTWARE SYSTEMS (DEFECTS), p. 32–36, 2008.

[31] CHAVEZ, A.; FERREIRA, I.; FERNANDES, E.; CEDRIM, D. ; GARCIA, A.. **How does refactoring affect internal quality attributes?** In: 31ST BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES), p. 1–10, 2017.

[32] KIM, S.; ERNST, M. D.. **Which warnings should i fix first?** In: PROCEEDINGS OF THE THE 6TH JOINT MEETING OF THE EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND THE ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, p. 45–54. ACM, 2007.

[33] BAVOTA, G.; DE LUCIA, A.; DI PENTA, M.; OLIVETO, R. ; PALOMBA, F.. **An experimental investigation on the innate relationship between quality and refactoring.** J. Syst. Softw (JSS), 107:1–14, 2015.

[34] KIM, M.; ZIMMERMANN, T. ; NAGAPPAN, N.. **A field study of refactoring challenges and benefits.** In: 20TH FSE, p. 50, 2012.

[35] SILVA, D.; TSANTALIS, N. ; VALENTE, M. T.. **Why we refactor? confessions of github contributors**. In: 24TH FSE, p. 858–870, 2016.

[38] BOURQUIN, F.; KELLER, R.. **High-impact refactoring based on architecture violations**. In: 11TH CSMR, p. 149–158, 2007.

[39] KIM, S.; ZIMMERMANN, T.; PAN, K.; JAMES JR, E. ; OTHERS. **Automatic identification of bug-introducing changes**. In: 21ST ASE, p. 81–90, 2006.

[40] KIM, M.; ZIMMERMANN, T. ; NAGAPPAN, N.. **An empirical study of refactoring challenges and benefits at microsoft**. IEEE Trans. Softw. Eng. (TSE), 40(7):633–649, 2014.

[41] RATZINGER, J.; SIGMUND, T. ; GALL, H. C.. **On the relation of refactorings and software defect prediction**. In: 5TH MSR, p. 35–38, 2008.

[42] LAMKANFI, A.; DEMEYER, S.; GIGER, E. ; GOETHALS, B.. **Predicting the severity of a reported bug**. In: 7TH MSR, p. 1–10, 2010.

[43] KHOMH, F.; DI PENTA, M. ; GUEHENEUC, Y.-G.. **An exploratory study of the impact of code smells on software change-proneness**. In: 16TH WCRE, p. 75–84, 2009.

[44] D'AMBROS, M.; BACCHELLI, A. ; LANZA, M.. **On the impact of design flaws on software defects**. In: 10TH QSIC, p. 23–31, 2010.

[45] KHOMH, F.; DI PENTA, M.; GUÉHÉNEUC, Y.-G. ; ANTONIOL, G.. **An exploratory study of the impact of antipatterns on class change- and fault-proneness**. Emp. Softw. Eng. (ESE), 17(3):243–275, 2012.

[46] RAHMAN, M.; ROY, C.. **On the relationships between stability and bug-proneness of code clones: An empirical study**. In: 17TH SCAM, p. 131–140, 2017.

[48] WU, R.; ZHANG, H.; KIM, S. ; CHEUNG, S.-C.. **Relink: recovering links between bugs and changes**. In: 11TH FOUNDATIONS OF SOFTWARE ENGINEERING (FSE), p. 15–25, 2011.

[50] ANTONIOL, G.. **Requiem for software evolution research: A few steps toward the creative age**. In: 9TH IWPSE CO-LOCATED WITH THE 6TH ESEC/FSE, p. 1–3, 2007.

[51] GÖRG, C.; WEISSGERBER, P.. **Error detection by refactoring reconstruction**. In: ACM SIGSOFT SOFTW. ENG. NOTES (SEN), volumen 30, p. 1–5, 2005.

[52] MURPHY, G. C.; KERSTEN, M. ; FINDLATER, L.. **How are java software developers using the elipse ide?** IEEE Software, 23(4):76–83, July 2006.

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

[57] KIM, M.; ZIMMERMANN, T. ; NAGAPPAN, N.. **A field study of refactoring challenges and benefits**. In: PROCEEDINGS OF THE ACM SIGSOFT 20TH INTERNATIONAL SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, p. 50. ACM, 2012.

[58] LAMKANFI, A.; DEMEYER, S.; GIGER, E. ; GOETHALS, B.. **Predicting the severity of a reported bug**. In: MINING SOFTWARE REPOSITORIES (MSR), 2010 7TH IEEE WORKING CONFERENCE ON, p. 1–10. IEEE, 2010.

[59] ZHANG, S.; ZHAO, J.. **On identifying bug patterns in aspect-oriented programs**. In: COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 2007. COMPSAC 2007. 31ST ANNUAL INTERNATIONAL, volumen 1, p. 431–438. IEEE, 2007.

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

[61] VALE, G.; FERNANDES, E. ; FIGUEIREDO, E.. **On the proposal and evaluation of a benchmark-based threshold derivation method**. Software Quality Journal, p. 1–32, 2018.

[62] OLIVEIRA, P.; VALENTE, M. T. ; LIMA, F. P.. **Extracting relative thresholds for source code metrics**. In: SOFTWARE MAINTENANCE, REENGINEERING AND REVERSE ENGINEERING (CSMR-WCRE), 2014 SOFTWARE EVOLUTION WEEK-IEEE CONFERENCE ON, p. 254–263. IEEE, 2014.

[63] BETTENBURG, N.; JUST, S.; SCHRÖTER, A.; WEISS, C.; PREMRAJ, R. ; ZIMMERMANN, T.. **What makes a good bug report?** In: PROCEEDINGS OF THE 16TH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, p. 308–318. ACM, 2008.

[64] AYEWAH, N.; HOVEMEYER, D.; MORGENTHALER, J.; PENIX, J. ; PUGH, W.. **Using static analysis to find bugs**. IEEE Software, 25(5), 2008.

[65] SHEN, H.; FANG, J. ; ZHAO, J.. **Efindbugs: Effective error ranking for findbugs**. In: 4TH INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION (ICST), p. 299–308, 2011.

[66] TSANTALIS, N.; MANSOURI, M.; ESHKEVARI, L. M.; MAZINANIAN, D. ; DIG, D.. **Accurate and efficient refactoring detection in commit history**. In: IEEE ICSE, volumen 2018, 2018.

A
Published Papers

Ferreira, I., Fernandes, E., Cedrim, D., Uchôa, A., Bibiano, A.C., Garcia, A., Correia, J.L., Santos, F., Nunes, G., Barbosa, C. and Fonseca, B., 2018, May. *The buggy side of code refactoring: understanding the relationship between refactorings and bugs.* In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (pp. 406-407). ACM.

Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D. and Garcia, A., 2017, September. *How does refactoring affect internal quality attributes?: A multi-project study.* In Proceedings of the 31st Brazilian Symposium on Software Engineering (pp. 74-83). ACM.

Garnier, M., Ferreira, I. and Garcia, A., 2017. *On the influence of program constructs on bug localization effectiveness: A study of 20 C# projects.* Journal of Software Engineering Research and Development, 5(1), p.6.